# BETAC: Bidirectional Encoder Transformer for Assembly Code Function Name Recovery

Guillaume Breyton, Mohd Saqib, Benjamin C. M. Fung
*McGill University*
Montreal, Canada
guillaume.breyton@mail.mcgill.ca, mohd.saqib@mail.mcgill.ca, ben.fung@mcgill.ca

Philippe Charland
*Defence R&D Canada*
Quebec, Canada
philippe.charland@drdc-rddc.gc.ca

***Abstract*—Recovering function names from stripped binaries is a crucial and time-consuming task for software reverse engineering, particularly in enhancing network reliability, resilience, and security. This paper tackles the challenge of recovering function names in stripped binaries, a fundamental step in reverse engineering. The absence of syntactic information and the possibility of different code producing identical behavior complicate this task. To overcome these challenges, we introduce a novel model, the Bidirectional Encoder Transformer for Assembly Code (BETAC), leveraging a transformer-based architecture known for effectively processing sequential data. BETAC utilizes self-attention mechanisms and feed-forward networks to discern complex relationships within assembly code for precise function name prediction. We evaluated BETAC against various existing encoder and decoder models in diverse binary datasets, including benign and malicious codes in multiple formats. Our model demonstrated superior performance over previous techniques in certain metrics and showed resilience against code obfuscation.**

***Index Terms*—Reverse engineering automation, binaries, assembly code, CodeBERT, Transformers, summarization**

## I. Introduction

Reverse engineering of executables has numerous practical applications, including improving and debugging legacy programs, understanding unknown binaries, filtering malware, and detecting illegally cloned proprietary code. However, due to the limited information that can be extracted from stripped binaries, the reverse engineering process can be time-consuming, expensive, and requires intensive training.

Despite significant progress in the development of disassemblers such as IDA Pro[1], and Ghidra[2], static analysis frameworks [1], [2], and similarity detectors [3], [4], the reverse engineering process remains predominantly manual. In recent years, notable advancements have been made in the prediction of variable and function names for high-level language source code. Various techniques, including code summarizing [5], [6], code retrieval [7], [8], code generation [9], [10], and name generation or suggestion [11], [12], have been proposed. Most of these approaches are based on simple models that are generated from a syntactic static analysis of the source code. These models are then fed into machine learning algorithms to provide some form of understanding of the source code. However, the traditional techniques for predicting variable and function names in high-level languages rely on syntactic analysis, which does not translate well to assembly code because of its lack of high-level syntactic information and variable types. The BETAC model, designed specifically for assembly code, overcomes these limitations by employing a transformer-based architecture that can understand complex relationships within assembly code without relying on high-level syntax, making it more suitable for this task.

Our research seeks to automate the renaming process of assembly functions to more informative names that describe the function's behavior or role. Unlike predicting variable names, predicting function names is a more feasible task as functions provide more significant insights into the code's purpose. Additionally, predicting variable names is even more challenging, due to the limited information available in the assembly code and the disassembler's limitations in predicting variable types. While disassemblers can efficiently separate the code into functions, they often struggle to predict the type of a variable. This is primarily because most high-level language types do not exist at the assembly level and only a limited number of types exist, similar to those found in the C language. Additionally, the disassembler may confuse between these types, such as representing an array of integers of length 3 as three pointers to integers, requiring the reverse engineer to identify the corresponding addresses as consecutive.

This paper makes the following contributions.

- The approach proposed in this study is a new representation model called *Bidirectional Encoder Transformer for Assembly Code* (*BETAC*). BETAC is the first model designed to produce generic features that are independent of the binary format or coding style. It employs transformers instead of Seq2Seq models, which is the first instance of this approach in this context. BETAC is also explicitly trained to predict generic features, a unique feature that sets it apart from previous approaches that use less diverse datasets to avoid the problem. Our experimental results reveal that BETAC achieves better BLEU scores than the state-of-the-art on all tested datasets [3], [13], including uncommon samples, such as obfuscated malware, and consistent accuracy, precision, and recall on all datasets. Given the nature of the task, BLEU scores are more relevant than accuracy, precision, and recall in this context.

This research is supported by NSERC Alliance Grants (ALLRP 561035-20), BlackBerry Limited, and Defence Research & Development Canada (DRDC).

[1]https://hex-rays.com/ida-pro/

[2]ttps://ghidra-sre.org/

Unlike previous research that focused on closed datasets of specific libraries in a specific binary format [3], [13], BETAC is trained on a diverse collection of assembly files and yields good and consistent results on diverse input data. BETAC represents a promising solution for automating the function name recovery process in reverse engineering, as it produces generic features and achieves superior performance across diverse input data.

- We address prior research limitations by not only developing and training our proposed model but also by using the largest and most diverse collection of assembly files to date. Our dataset consists of samples from four publicly accessible sources, including two existing datasets that were previously used for similar problems, a large dataset of ELF files from open-source libraries, and a small dataset of malware compiled with various obfuscations. Unlike prior research that focused on a single binary format, we train and test our model on both ELF and PE formats, enabling it to perform well on diverse input data. Our approach demonstrates the model's ability to predict function names accurately, even in the presence of obfuscation, and achieve promising results on diverse input data. This approach is a significant step towards automating the function name recovery process in reverse engineering, enabling practitioners to better understand and analyze complex codebases.

## II. Related Work

### A. Machine learning for source code

Machine learning approaches for predicting names in high-level programming languages have been studied extensively in previous work. Some works rely solely on syntax [14], while others employ semantic analysis [15], or a combination of both [16]. Syntax-based methods are advantageous for languages such as Java and JavaScript, which have a rich syntax but may not be applicable in the case of binary reverse engineering, where syntax information is lacking. Brockschmidt et al. [10] incorporated semantic analysis using Gated Graph Neural Networks, where the graph edges were relations discovered via semantic analysis. However, graph neural networks are not directly applicable to our approach, as our work focuses on instruction sequences rather than complex graph structures. In a recent study, distributed representations of C functions were learned based on control flow graphs (CFG) [17]. We also use CFGs, but operate on the more challenging domain of stripped binaries, instead of C code.

### B. Static analysis frameworks for reverse engineering

Some initial approaches [1] used statistical methods to infer subclass and superclass relationships and fill in any missing structural information in stripped binaries using behavioral information. Reps et al. [18] proposed a binary executable framework specifically designed for analyzing x86 executables. Their work focused on detecting and manipulating variables in the assembly code to obtain information about unknown variables and recover concealed data structures and patterns in the code that could indicate a hierarchy in the functions. While this framework provides valuable structural information, it lacks the semantics of each function, which reverse engineers must devote significant time to deciphering. Therefore, we concentrate on semantics in our research, specifically function names rather than the binary's structure.

### C. Relating source code to assembly instructions

The first approaches in reverse engineering attempted to decompile assembly instructions to source code, then perform a complete software analysis of the result [19]. This method is now being replaced by machine learning and artificial intelligence methods [20]. Therefore, it is often used as a baseline in more recent research.

Recent research has explored novel approaches to improve the accuracy and efficiency of reverse engineering. For instance, Qiu et al. [20] used a machine learning approach to create a weighted prefix tree of start bytes of instruction sequences for functions. Their approach yielded better results than previously tested methods, including popular disassembly tools. Additionally, Li et al. [21] proposed a Recurrent Neural Network architecture that takes the binary code as input and decides whether there is a function boundary or not. Their model outperforms previous approaches, in particular in terms of computation time. This research utilizes an architecture inspired by successful natural language processing solutions.

Another notable study by Hu et al. [22] proposed a model for aligning the source code and the binary code in order to fully relate the two. This approach uses two LSTM encoders to create representations of both codes, followed by a CNN that produces alignment scores when given the code pairs as input. While this study could be extended to summarize the assembly code, as well as the information already identified through the alignment process, it would be a harder problem than what we are trying to achieve in this study. Recently, Mikolov et al. [23] proposed an LSTM-based encoder-decoder model that translates the binary code into source code. Our work takes a similar direction, but we output natural language tokens instead of source code.

### D. Encoder-decoder for function name recovery

Several recent papers have made significant progress in the field of binary function name recovery. In 2018, He et al. [24] proposed a decision tree-based model that performed well on a closed dataset, but had limited scalability and showed a drop in performance when tested on a diverse dataset. This model did not leverage the sequential nature of the input or extract structural features from the CFG or call graph.

Lacomis et al. [13] incorporated both lexical and structural information by training a bidirectional LSTM on decompiled source code and a gated-graph encoder-decoder on the Abstract Syntax Tree. They achieved promising results for the prediction of function and variable names in a large data set. David et al. [3] proposed a new approach that produces a call-site graph by enriching the list of calls to external library functions, with the nature of the passed arguments. They used

```
.text:00401015                push    esi
.text:00401016                push    edi
.text:00401017                mov     edi, [ebp+8]
.text:0040101A                lea     eax, [esp+5Ch]
.text:0040101E                xor     ebx, ebx
```

Fig. 1. Code instructions

this enriched list as input to an encoder-decoder paradigm and achieved good results, though this method is only suitable for functions that call multiple external library functions. Artuso et al. [25] emphasized the effectiveness of natural language processing techniques for tasks such as naming functions in stripped binaries and used Seq2Seq and Transformer models to achieve good results when fine-tuned for a specific domain with a single binary format and a set of given libraries. Our research aims to provide a tool that gives useful results on diverse binaries, and we chose an encoder-decoder paradigm for this purpose. In Section V, we compare our solution with three of the four aforementioned models.

## III. The Problem of Function Name Recovery

In this section, we formally present our problem and discuss the three primary challenges that we encountered, along with our solutions to address them.

### A. Problem definition

Let $V$ be a vocabulary of tokens (tokenized words and the NULL token) with a distance $s$ representing the similarity of tokens in $V$. If $v, v'$ are tokens of $V$, $s(v, v')$ is close to 0 if $v$ and $v'$ are related tokens, or tokens with a close meaning. In particular, $s(v, v') = 0$ means $v = v'$. Let $f$ be a function obtained from a disassembled stripped binary file. $f$ has a name composed of up to $k$ words that we can tokenize and describe as $(t_1, \ldots, t_k)$. We consider the last tokens to be NULL if the function $f$'s name includes less than $k$ tokens. We suppose that $(t_1, \ldots, t_k)$ are in $V$. Let $A$ be a model that takes $f$, $V$ and $(t_1, ..., t_k)$ as input, and outputs $k$ tokens $(t'_1, \ldots, t'_k)$ taken from $V$. The *problem of function name recovery* is to find the optimal model $A$ such that for any function $f$, $(t_1, \ldots, t_k) = (t'_1, \ldots, t'_k)$, i.e., produce a model $A$ that gives a minimal distance $s(t_i, t'_i)$ for $1 \leq i \leq k$.

### B. Challenge 1: extracting syntactic information is hard

The use of binary format as input for machine learning models poses significant challenges. Therefore, we limit our input to assembly code, as illustrated in Figure 1, which is obtained through a decompiler, such as Ghidra or IDA Pro. Assembly code is a human-readable format of binary code that facilitates pre-processing and feeding to machine learning models. However, we cannot directly apply natural language or source code models to assembly code. This is due to the assembly code's limited vocabulary of tokens, each of which has little meaning outside its context. The context in this case is the current instruction of the token and any instructions that were executed before or after the current instruction. However, since jumps occur in the assembly code during function execution, the instructions executed before or after the current instruction may not be immediately adjacent.

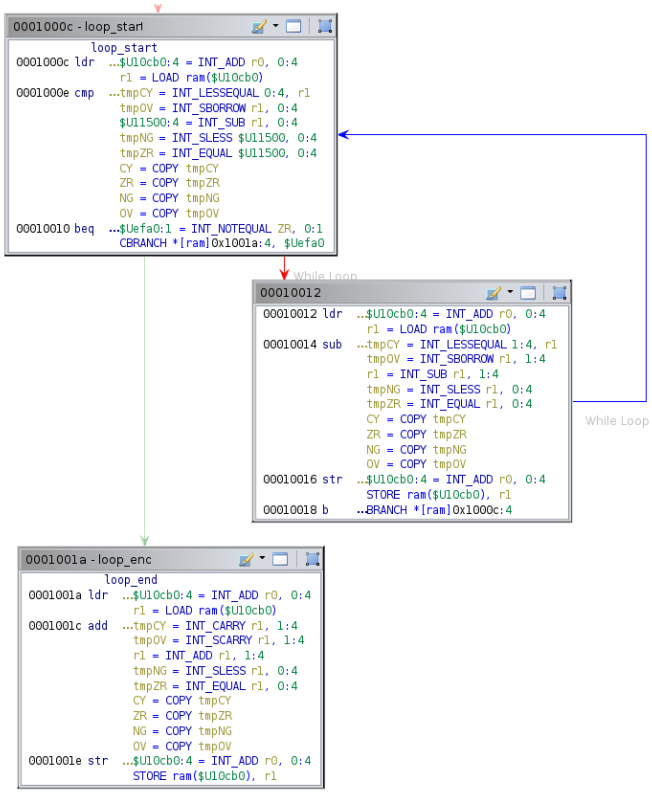


Fig. 2. Partial CFG

To address the vocabulary limitation, we treat the instruction as the basic input unit instead of the word. Typically, an instruction comprises at most three words, a mnemonic, and zero to two operands. We tokenize the words and represent the instruction as a list of three tokens.

To capture the instruction's context, we create the CFG of the function, where each node is a list of instructions executed sequentially, regardless of the input. The edges of the graph are directed and represent the loops and conditions of the function, with an example in Figure 2. The different paths in the graph represent the possible instruction sequences executed by the function, depending on the input. Instead of having one instruction sequence as input, we have several sequences, based on the number of paths in the CFG.

### C. Challenge 2: navigating structural information variability

Two functions with different structures can have the exact same behavior. Therefore, multiple CFGs can represent functions that have the same role. The structural information provided by the analysis of the CFG depends on the coding style, the libraries used, etc. Moreover, the CFG of a function can be profoundly modified by obfuscation. We want to get rid of this kind of phenotypical information that is not interesting for a reverse engineering process because we only care about the role and the behavior of the function.

We use some simple deobfuscation techniques to clean our Control Flow Graph before the input is fed to the machine learning model. We remove dead or useless branches in the graph: some obfuscation techniques add useless pieces in the

code to make it bigger and harder to reverse engineer. We also simplify data structures. To obfuscate the code, data structures that are easy to understand, such as arrays, can be replaced by individual pointers to addresses in memory.

### D. Challenge 3: obtaining consistent labelled data is hard

To produce a list of tokens for a function's name, labelled data is required. However, obtaining such data is challenging, even if the corresponding source code is available, as it requires manual matching of each assembly function with its corresponding source code function. Additionally, due to variations in naming styles, the same function may have different names that describe different aspects of its behavior, leading to inconsistent labelling in the dataset. Furthermore, we assume that each function performs a single task described by its name, but a function may perform multiple tasks sequentially or have behavior conditioned by its input.

To automate the process of matching source code names with assembly functions, we added a debug symbol at the beginning of each function that remains after compilation, if it was not already present.

## IV. The Solution

Our proposed model, *BETAC*, follows an encoder-decoder paradigm, as illustrated in Figure 3. After preprocessing the words and instructions, multiple sequences of assembly instructions are created for each function, where each sequence represents a possible path of execution for that function. These sequences are used as inputs and fed into the BETAC encoder. As demonstrated in Figure 4, multiple possible sequences of code execution exist for each assembly function, and for each sequence, the encoder-decoder model generates a list of five tokens. These lists are then combined using a simple set of rules, resulting in one final list of five tokens that describe the generated name of the assembly function.

### A. Data representation

*a) Tokenizing the function names.:* The process has four steps. First, we split function names into words. It can be automated, since all the function names in our datasets follow either the camel case, e.g., `getElementsByTagName`, or snake notations, e.g., `convert_to_int`. Second, we use stemming to make tokens out of the words, which reduces the vocabulary. Different forms of the token are mapped into the same base form. For example, the tokens "shared" and "sharing" are both mapped to "share". Third, we build the vocabulary by removing useless and meaningless tokens. We first assign a score to each token and then retain only tokens whose score is above a certain threshold $\tau$. The score for each token $t$ is the project frequency, i.e. the number of different packages in which a token appears. This allows us to exclude tokens that appear only in a few packages, even if they appear with a high frequency in these packages. This way, we avoid assigning a large score to tokens that are not semantically relevant. Finally, we exclude from the vocabulary all tokens that have no meaning ending up with a vocabulary of 2,088 tokens. Fourth, we rename the functions with only the tokens taken from their name and present in the vocabulary. Some functions have no name after this last step. We remove them from the data.

*b) Tokenizing the assembly words.:* The operands of each instruction refer to registers and addresses in memory. We use one token per type of register, which gives us 8 tokens: `AX` (used for RAX, EAX, and AX), `BX`, `CX`, `DX`, `SP`, `BP`, `SI`, `BI`. All other registers are tokenized into `REG`. We also use `GLOBAL` and `CONST` tokens for global and constant variables respectively. As an example, the instruction `mov edx, [ebp+0C]` is tokenized into `MOV, DX, BP`. To these tokens, we add all the mnemonics as such. Then we apply the same method of scoring and keeping tokens above a chosen threshold as used for tokenizing the function names. All tokens that have been removed from the vocabulary are replaced by `UNK`.

*c) Producing the CFGs.:* We use the CFGs generated by Ghidra. We remove the branches going upwards in the graph to avoid infinite loops and create a tree. We consider that each instruction will be read at most one time during execution. In other words, loops run at most one time. We then perform a breadth-first exploration of the tree and create a new file for each execution path.

*d) Preparing the assembly instructions.:* For each assembly instruction, we keep four tokens: one for the mnemonic, two for the operands that are filled by the `NULL` token if the instruction has less than two operands, and one `COMA` token that signals the end of the instruction. We put a `BEGIN` token at the beginning of the sequence of instructions and an `END` token at the end.

### B. Encoding the Assembly Instruction Sequence

Although the basic process is the same, different neural network models can be used as the encoder. Before coming up with BETAC as the solution for our encoder, we tried other simpler models. The encoder outputs a context vector, which is used as the input for the decoder. We compared our transformer-based model BETAC with LSTM, GRU, the two best-performing Seq2Seq models, as well as their bimodal versions, Bi-LSTM and Bi-GRU. The advantage of LSTM neural networks is that they prevent the loss of important information through the training process, by preserving information from past inputs in its hidden state on the long term. GRU neural networks also work this way, but they reduce the number of gates used and remove the cell state.

Our proposed model, *BETAC*, is a Transformer-based architecture, which is renowned for its effectiveness in processing sequential data. It is designed with layers of self-attention mechanisms and feed-forward networks, allowing it to learn complex relationships within assembly code for function name prediction. Similar to *CodeBERT* [26], it is trained with a hybrid objective function that incorporates the pre-training task of replaced token detection, which is to detect plausible alternatives sampled from generators. We train BETAC by *Masked Language Modeling* (*MLM*) and *Replaced Token*

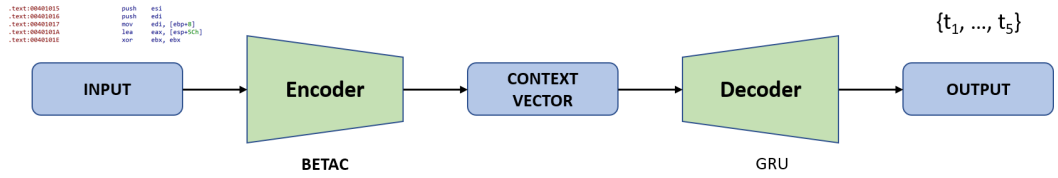


Fig. 3. Pipeline of the model for single sequence. Each sequence of instructions is fed into an encoder-decoder that produces a list of tokens as outputs.

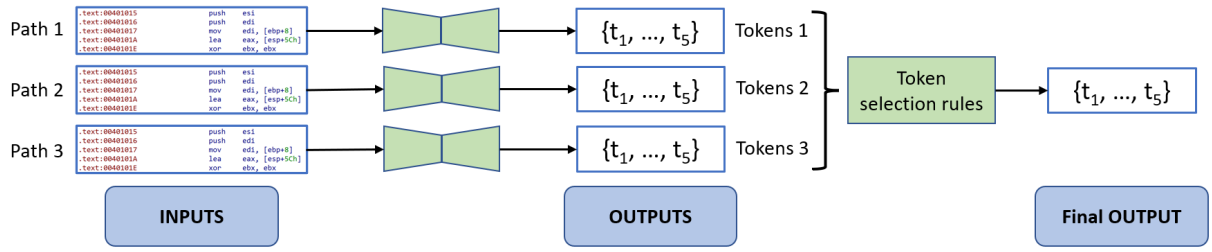


Fig. 4. Overall pipeline of the model. Each function contains multiple sequences. Each sequence produces a list of 5 tokens. We use a set of rules to combine them into a list of 5 tokens.

*Detection* (*RTD*), which both have been proven effective in the literature. We follow the procedures described in the original CodeBERT [26] publication, adapted to assembly code. We apply MLM on bimodal data of *Source Code - Assembly Code* (*SC-AC*) pairs. We mask 15% of the tokens from the input and ask the model to predict the original tokens, which are masked out, using a discriminator that predicts a token from a large vocabulary set, while we train RTD on unimodal data, i.e., assembly without source code. The original objective of RTD is to efficiently learn a pre-trained model for natural language. We adapt it for our scenario. We use two data generators, a source code generator and an assembly code generator, both for generating plausible alternatives for the set of randomly masked positives. The discriminator is trained to determine whether a word is the original one or not, which is a binary classification problem.

### C. Decoding to Natural Language

The decoding process takes the context vector produced by the encoder as its input and outputs natural language tokens taken from the previously built token vocabulary. We first used Seq2Seq (STM/GRU)-based Context Representation. Since the internal states and the context vector depend on the encoder model used, we can only use an LSTM decoder with an LSTM decoder, the same applies for GRU. We then changed to an Attention-based Context Representation, where an attention mechanism is added to the model, with the perspective of improving the results and determining the leading factors behind predicting the output. After generating a list of output tokens for each path in the CFG, we extract the top 5 tokens based on their frequency in the outputs. In cases where two tokens have equal frequency, we choose the one with the highest score in the vocabulary, which corresponds to the token that appears most frequently in the dataset.

### D. Optimization

Our models are trained using gradient descent algorithms. We specifically use the Root Mean Squared Propagation (RMSprop) algorithm, due to its adaptive learning ability [29] and use of mini-batches. The adaptive learning rate means that each parameter is updated at its own rate in order to optimize its individual loss.

*a) Objective function.:* In this study, we adopt cross-entropy as the objective function for our models. Due to the sparse nature of our data, we utilize sparse categorical cross-entropy. Specifically, this function measures the distance between the probabilities obtained from the softmax function and the true values. For instance, assuming that the first output tokens have been produced, if the next target token is ”data” and based on the softmax function results, ”data” has a 80% probability, there would be a 20% loss if the model were to predict the wrong token.

## V. Experiments

In this study, we train and evaluate BETAC on two primary tasks: masked language modeling and replaced token detection. These evaluations are conducted on three distinct datasets, which include two standard datasets for function name recovery and an additional dataset composed of public libraries. To assess the performance of BETAC on uncommon code, we also conduct tests on a publicly available malware dataset.

### A. Evaluation metrics

Evaluation metrics play a crucial role in quantifying the performance of the model’s predictions. While various evaluation metrics exist, it is essential to select those that are most relevant to the specific research conducted. In this study, we focus on four primary evaluation metrics: Bilingual Evaluation Understudy (BLEU), accuracy, recall, and precision.

*a) Bilingual Evaluation Understudy Score (BLEU).:* When evaluating the quality of machine translation on natural language data, it is crucial to assess the readability and fidelity of the translated output in comparison to the original text. One widely adopted metric for this purpose is the BLEU Score. This score serves as a quantitative measure to assess the

TABLE I
SUMMARY OF DATASETS

| Dataset | File Type | Collected Samples | After Preprocessing | Train Size | Test Size |
|---|---|---|---|---|---|
| Juliet [27] | Source & Assembly | 272,431 | 83,326 | 62,494 | 20,832 |
| NDSS18 [28] | PE & ELF | 62,563 | 15,102 | 11,102 | 4,000 |
| Ubuntu | ELF | 50,055 | 25,340 | 20,210 | 5,230 |
| MALWARE | PE | 3,710 | 3,710 | - | 3,710 |

translation performance by comparing the generated sentence with a reference sentence. It is scaled between 0 and 1. A BLEU score of 1 indicates a perfect match, while a score of 0 represents a complete mismatch. Higher BLEU scores signify better alignment with professional human translations.

The BLEU score operates on a per-token basis, disregarding the specific token positions within the sequence. To ensure fairness in evaluating repeating tokens within a reference sequence, the comparison is adjusted. For instance, if a ground-truth sequence contains the word "create" one time, and the predicted sequence contains the word "create" two times, only one instance of "create" can be considered as correctly matching token.

The BLEU score also takes into account the word distribution in the reference sentence. Consequently, the score naturally decreases when the predicted sequence is longer than any of the reference sentences. It should be noted that the BLEU score can be influenced by the number of reference sentences used during the evaluation. Having more reference sequences per translation increases the likelihood of achieving a higher BLEU score, as there are more opportunities for the prediction to align correctly. This aspect makes it difficult to compare the BLEU scores between different research studies, due to variations in the number of reference sentences used for evaluation.

*b) Accuracy.:* In evaluating the model's performance, the accuracy metric is used. An output token is classified as a true positive if it is present in the ground truth, while it is classified as a true negative if it is absent in the ground truth. On the contrary, false positives and false negatives represent tokens that are incorrectly identified as present or absent, respectively.

To measure accuracy, the true natural language description extracted from the source code is compared with the corresponding predicted outputs of the model. Accuracy increases as the number of pairs of matching truth predictions increases. It is worth noting that accurate predictions do not need to exactly replicate the true description. Various phrasings and expressions can convey the same message. For instance, "a number greater than zero" is synonymous with "a non-negative number" and "a positive number". A prediction that preserves the underlying meaning of the original description is considered equally accurate, if not more so, as it demonstrates the model's learning capability rather than a mere replication of the training data.

The accuracy of the model is computed using two sets of results. Adjusted accuracy assesses the model's ability to predict the correct output when provided with the correct input. In this case, the decoder inputs are adjusted to consistently supply the expected input for the target sequence. Decoder outputs that do not match the expected result are excluded from subsequent word predictions. Overall accuracy gauges the independent performance of the model, where the output of the decoder is the input for the subsequent decoding step.

*c) Recall:* Recall, defined as the proportion of token inputs that yield accurate outputs, serves as a valuable measure for evaluating the coherence of predicted tokens. This involves the assessment of the grammatical correctness of the predicted outputs. The methodology used deems an output token as correct if it is proximate to the ground truth, even when it is not an exact match. Consequently, a high recall is indicative of a model that outputs a logically consistent sequence of tokens.

*d) Precision:* In contrast, precision represents the ratio of correctly identified outputs to total outputs, evaluated against the true output. This metric appraises the degree of match between the current token output and the corresponding ground truth target. Given the diverse functions' behaviors and roles incorporated into the dataset, function names predominantly exhibit uniqueness. Therefore, a high precision rate for initial output tokens in a sequence tends to suggest an overall high precision for the complete output.

### *B. Datasets and pre-processing*

We train our data on three datasets and test it on four datasets (Table I). The datasets utilized encompass a range of software types, from benign to malicious, offering a broad spectrum for training and testing the feature encoder. The Juliet [27] and NDSS18 datasets [28], comprising source and assembly code pairs, are pivotal for training on diverse code samples and architectures. The Ubuntu dataset[3], with its extensive collection of ELF files from various releases, and the MALWARE dataset, specifically for testing against malicious code samples, underscore the comprehensive approach to enhancing model robustness and applicability in real-world scenarios.

We use the following set of hyperparameters: batch size of 1,024 and learning rate of 5e-4. We use Adam to update the parameters and set the number of warm-up steps as 10K. In the fine-tuning step, we set the learning rate as 1e-5, the batch size as 64, and the fine-tuning epoch as 8. As the same for pre-training, we use Adam to update the parameters.

[3]https://old-releases.ubuntu.com/

TABLE II
BLEU SCORES OF BETAC AND OTHER METHODS

| Model | Juliet | NDSS18 | Ubuntu | Malware |
|---|---|---|---|---|
| LSTM | 0.523 | 0.512 | 0.129 | 0.221 |
| Bi-LSTM | 0.506 | 0.492 | 0.210 | 0.231 |
| GRU | 0.513 | 0.481 | 0.189 | 0.229 |
| Bi-GRU | 0.534 | 0.516 | 0.401 | 0.301 |
| Bi-GRU + Att | 0.537 | 0.880 | 0.612 | 0.321 |
| NERO-GNN | 0.278 | 0.331 | 0.650 | 0.321 |
| DIRE | 0.228 | 0.387 | 0.310 | 0.249 |
| **BETAC** | **0.538** | **0.882** | **0.651** | **0.351** |

TABLE III
ACCURACY OF BETAC AND OTHER METHODS

| Model | Juliet | NDSS18 | Ubuntu | Malware |
|---|---|---|---|---|
| LSTM | 77.7 | 65.9 | 38.7 | 41.6 |
| Bi-LSTM | 74.9 | 59.6 | 42.2 | 49.3 |
| GRU | 76.3 | 57.7 | 45.1 | 52.7 |
| Bi-GRU | 79.2 | 69.2 | 49.9 | 55.4 |
| Bi-GRU + Att | **79.6** | **84.1** | **64.5** | 55.3 |
| NERO-GNN | 66.7 | 65.3 | 51.6 | 39.2 |
| DIRE | 59.1 | 59.2 | 53.6 | 41.4 |
| **BETAC** | 78.0 | 79.0 | 63.4 | **60.0** |

TABLE IV
PRECISION OF BETAC AND OTHER METHODS

| Model | Juliet | NDSS18 | Ubuntu | Malware |
|---|---|---|---|---|
| LSTM | 76.3 | 37.0 | 21.0 | 39.0 |
| Bi-LSTM | 73.8 | 40.0 | 22.1 | 41.4 |
| GRU | 76.9 | 31.0 | 23.8 | 38.1 |
| Bi-GRU | 79.2 | 38.6 | 26.3 | 40.2 |
| Bi-GRU + Att | 77.9 | 68.6 | 39.4 | 40.9 |
| NERO-GNN | 79.6 | 39.0 | 23.14 | 35.6 |
| DIRE | **80.1** | 32.3 | **40.53** | **42.1** |
| **BETAC** | 79.0 | **68.3** | 25.10 | 40.9 |

TABLE V
RECALL OF BETAC AND OTHER METHODS

| Model | Juliet | NDSS18 | Ubuntu | Malware |
|---|---|---|---|---|
| LSTM | 66.9 | 32.9 | 21.2 | 29.2 |
| Bi-LSTM | 64.2 | 31.8 | 23.9 | 23.1 |
| GRU | 65.8 | 26.8 | 23.0 | 21.0 |
| Bi-GRU | 69.0 | 32.7 | 26.9 | 24.6 |
| Bi-GRU + Att | 68.6 | 60.0 | **39.9** | 32.5 |
| NERO-GNN | 58.5 | 62.3 | 37.26 | 25.2 |
| DIRE | 60.1 | **67.2** | 25.88 | 28.7 |
| **BETAC** | **68.7** | 29.23 | 29.23 | **29.5** |

### C. Results

The objective of this experiment is to evaluate the performance of our proposed model, BETAC, with other potential solutions on benignware datasets, as well as on some uncommon datasets, such as obfuscated malware files.

*a) BLEU score of the model.:* Table II shows the results we obtained on the three main datasets: the Juliet Test Suite, NDSS18, and Ubuntu. We compare the results of our model, BETAC, with the results obtained by previous research and Seq2Seq models on our testing datasets. We obtain better BLEU scores than the other methods, which we interpret as having trained our model well and giving really good natural language names to our functions.

*b) Accuracy of the model.:* Although our model yields accuracy marginally lower than the top-performing Bi-GRU model supplemented with an attention mechanism, it demonstrates the second-highest performance among all evaluated models. The consistent superior performance of the models incorporating an attention mechanism suggests that this component contributes significantly to enhancing the model efficiency. Results are shown in Table III.

Our model, with a depth four times greater than the Bi-GRU model, may not have reached its full potential due to limitations related to the dataset size and the extent of fine-tuning applied. For comparison, models similar in scale to ours, such as CodeBERT, are typically trained on a substantially larger number of samples, ranging from 100,000 to several million. In contrast, our training dataset was limited from 10,000 to 100,000 samples. This discrepancy suggests that the comparatively lower accuracy of our model could be attributed to dataset size and the model fine-tuning constraints.

*c) Precision and recall of the model.:* In terms of precision and recall, our model excels in one of three datasets and closely approaches top performance in the remaining datasets. Results are shown in Table IV for precision and Table V respectively. Occasional losses in precision and recall across these datasets can be attributed to the existence of samples featuring translations that, while not entirely precise, are not incorrect. For instance, a predicted output of "non-negative" versus a ground truth of "greater than zero" might decrease precision and recall scores, although the semantic interpretation of these outputs is equivalent. This scenario underscores the limitation of these metrics in accurately capturing semantic correctness.

Our model consistently delivers the most stable performance across all datasets, establishing it as an optimal choice for real-world applications. Notably, reverse engineers seldom need to handle well-known open-source files, as these are usually identified by a clone search engine prior to any manual work. Instead, reverse engineers are more likely to encounter malicious files. Accordingly, the results for our malware dataset demonstrate that our model provides superior results on diverse and uncommon data. We assume that the application of a large, deep neural network is the most effective approach for creating a robust model capable of consistently delivering high-quality results in real-world scenarios.

*d) Qualitative analysis.:* The findings of our metrics analysis were supplemented with a manual review, comparing the predicted outputs with their respective true descriptions. We exhibit select intriguing examples in Table VI for elementary function names and Table VII for their more advanced counterparts. Certain output instances closely align with, or are identical to, the ground truth. The model generation process prioritizes the output of the most pertinent words first, while subsequent words depend not only on the input, but also on the preceding output words. This implies that the initial words in each list are more relevant than those appearing later. Consequently, the sequence in which words are output provides an indirect measure of their relative significance in the context of the model's predictions.

## VI. CONCLUSION

In this paper, we have presented a large bimodal pre-trained neural network model called BETAC to tackle th

TABLE VI
SAMPLES OF PREDICTED SHORT FUNCTION NAMES

| Prediction | Ground Truth |
|---|---|
| allocate array sizeof | allocate memory sizeof |
| copy array | copy array |
| add value | increment index |
| array sort | check index validity |
| create connect | create socket |

TABLE VII
SAMPLES OF PREDICTED LONG FUNCTION NAMES

| Prediction | Ground Truth |
|---|---|
| get elements by tag name | read input tag value start loop |
| create new document | create new document file input |
| delete last entry from table | delete element table input previous |
| try connection with socket | connect array by file |

problem of assembly function name recovery. BETAC encodes the assembly language and is the first model to use transformers in function name recovery. We demonstrate that fine-tuning BETAC achieves state-of-the-art performance for function name recovery on typical benignware datasets and outperforms state-of-the-art models on atypical data such as malware. BETAC is a promising model for real-life use cases, as reverse engineering often focuses on legacy software written in different styles and on obfuscated malware.

## REFERENCES


[1] O. Katz, N. Rinetzky, and E. Yahav, "Statistical reconstruction of class hierarchies in binaries," *SIGPLAN Not.*, vol. 53, no. 2, p. 363–376, mar 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173202

[2] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs." 01 2011.

[3] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries," *CoRR*, vol. abs/1902.09122, 2019. [Online]. Available: http://arxiv.org/abs/1902.09122

[4] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015, pp. 709–724.

[5] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *CoRR*, vol. abs/1808.01400, 2018. [Online]. Available: http://arxiv.org/abs/1808.01400

[6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *CoRR*, vol. abs/1803.09473, 2018. [Online]. Available: http://arxiv.org/abs/1803.09473

[7] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: A neural code search," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, p. 31–41.

[8] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei, "Bimodal modelling of source code and natural language," in *ICML*, 2015.

[9] Y. Lu, S. Chaudhuri, C. Jermaine, and D. Melski, "Data-driven program completion," *CoRR*, vol. abs/1705.09042, 2017. [Online]. Available: http://arxiv.org/abs/1705.09042

[10] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," *CoRR*, vol. abs/1805.08490, 2018. [Online]. Available: http://arxiv.org/abs/1805.08490

[11] Y. Li, S. Wang, and T. N. Nguyen, "A context-based automated approach for method name consistency checking and suggestion," *CoRR*, vol. abs/2103.00269, 2021. [Online]. Available: https://arxiv.org/abs/2103.00269

[12] R. Bavishi, M. Pradel, and K. Sen, "Context2name: A deep learning-based approach to infer natural variable names from usage contexts," *CoRR*, vol. abs/1809.05193, 2018. [Online]. Available: http://arxiv.org/abs/1809.05193

[13] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. L. Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," 2019.

[14] P. Bielik, V. Raychev, and M. T. Vechev, "Phog: Probabilistic model for code," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, ser. JMLR Workshop and Conference Proceedings, M.-F. Balcan and K. Q. Weinberger, Eds., vol. 48. JMLR.org, 2016, pp. 2933–2942. [Online]. Available: http://jmlr.org/proceedings/papers/v48/bielik16.html

[15] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017. [Online]. Available: http://arxiv.org/abs/1711.00740

[16] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," *SIGPLAN Not.*, vol. 50, no. 1, p. 111–124, jan 2015. [Online]. Available: https://doi.org/10.1145/2775051.2677009

[17] D. DeFreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to specification mining," *CoRR*, vol. abs/1802.07779, 2018. [Online]. Available: http://arxiv.org/abs/1802.07779

[18] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum, "A next-generation platform for analyzing executables," in *Programming Languages and Systems*, K. Yi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 212–229.

[19] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 480–491. [Online]. Available: https://doi.org/10.1145/2976749.2978370

[20] J. Qiu, X. Su, and P. Ma, "Library functions identification in binary code by using graph isomorphism testings," in *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 04 2015, pp. 261–270.

[21] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 406–415. [Online]. Available: https://doi.org/10.1145/2664243.2664269

[22] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm – software protection for the masses," in *2015 IEEE/ACM 1st International Workshop on Software Protection*, 2015, pp. 3–9.

[23] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013. [Online]. Available: http://arxiv.org/abs/1310.4546

[24] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1667–1680. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243866

[25] F. Artuso, G. A. D. Luna, L. Massarelli, and L. Querzoni, "Function naming in stripped binaries using neural networks," *CoRR*, vol. abs/1912.07946, 2019. [Online]. Available: http://arxiv.org/abs/1912.07946

[26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," *CoRR*, vol. abs/2002.08155, 2020. [Online]. Available: https://arxiv.org/abs/2002.08155

[27] P. Black, "Juliet 1.3 test suite: Changes from 1.2," 2018-06-14 2018.

[28] T. Le, T. Nguyen, T. Le, D. Phung, P. Montague, O. De Vel, and L. Qu, "Maximal divergence sequential auto-encoder for binary software vulnerability detection*," in *International Conference on Learning Representations 2019*, A. Rush, Ed., 2019.

[29] Y. N. Dauphin, H. de Vries, J. Chung, and Y. Bengio, "Rmsprop and equilibrated adaptive learning rates for non-convex optimization," *CoRR*, vol. abs/1502.04390, 2015. [Online]. Available: http://arxiv.org/abs/1502.04390