Project Report
LSP-375

On the Feasibility of Training an AI to Understand Programs: FY23 Cyber Security Line-Supported Program

A.T. Davis
A.M. Interrante-Grant
H.N. Preslier
T.R. Leek

26 April 2023

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.

This report is the result of studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Massachusetts Institute of Technology
Lincoln Laboratory

On the Feasibility of Training an AI to Understand Programs: FY23 Cyber Security Line-Supported Program

A.T. Davis
A.M. Interrante-Grant
H.N. Preslier
T.R. Leek

Group 59

Project Report LSP-375

26 April 2023

This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001.

Lexington Massachusetts

This page intentionally left blank.

ABSTRACT

The United States is suffering a critical shortage of cyber security personnel [1]. While several efforts exist to attempt to train and recruit a larger workforce, an important parallel effort involves building tools to amplify the effectiveness of current practitioners. Artificial intelligence (AI) has been used to build many tools to augment workforces in other areas [2] [3] [4] [5] [6] [7] [8]. Additionally, there have been multiple academic publications on using AI to aid in reverse engineering (a critical cyber security task) [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21]. In this study, the possibility of training an AI on the task of program understanding was investigated. Specifically, the AI would take, as input, mechanically extracted features of programs and output English word-and-sentence descriptions of functionality. This output would be expected to aid a reverse engineer in investigating the capabilities and vulnerabilities of a piece of software. The input features might be static, meaning they are gleaned only from inspection of the software, or they might be dynamic, meaning they are extracted from program executions. In this seedling study, we investigated a number of recent publications, existing datasets, data sources, and embeddings[1] for binaries and English prose. As part of our study, we constructed a novel dataset, which will be made available to the research community for general use. In brief, the results of this study are twofold. First, the dataset we constructed from over a million stack overflow pages is not of high enough quality to be used in training an AI for program understanding. Further, there is some evidence that the embedding used for English prose is too coarse for our purpose, conflating concepts we would have hoped it to distinguish. This report concludes with some ideas for future investigations including using our dataset quality measures to identify or weight higher quality exemplars, and some ideas involving using prose extracted from source and auto-generated web searches.

[1] An embedding, here, is simply a way of projecting data such as English prose or program binary code onto a vector space for use in machine learning, such as with a neural network.

This page intentionally left blank.

ACKNOWLEDGMENTS

The authors would like to thank the Line committee for giving us seedling funding to pursue this research and the MIT Lincoln Laboratory Supercomputing Center for computing resources and help using them.

This page intentionally left blank.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF FIGURES

This page intentionally left blank.

LIST OF TABLES

This page intentionally left blank.

1. INTRODUCTION

Programs are big and opaque and we typically know little, with certainty, about their function except at the highest levels of abstraction. This fact is at variance with the obvious need to be able to state, with confidence, that some code does this or that and no more. Today, there are two approaches to addressing this issue of *program understanding*. The first is to trust the developer and/or the vendor of the software to properly and completely document its functionality. For small systems, this can work well, assuming the requirement for documentation was in place from the start. However, this approach fails when the developers of the software are unavailable, uninterested in, or even adversarial to documenting their code. Malware is a common case that presents such issues, and an entire industry has been erected around attempting to understand malicious code, in order to detect its presence or mediate its effects. In this case, the second approach is typical, in which a team of specialists is tasked with reverse engineering a system and documenting its functionality. Even for non-malicious software, documentation is costly and time-consuming and rarely kept up to date.

One would expect that program analysis would be of assistance, and it has been applied, historically, to program understanding [22] [23] [24] [25]. More recently, AI has been employed, with surprisingly good effect, to explicate certain aspects of programs: to build plausible decompilers that invert binary code back to source [10] [11] [12] [13] [14] [15], to concoct reasonable variable and function names [16] [17] [18] [19], and to characterize data-flow from dynamically extracted training examples [20].

In this study, we aimed to determine if recent advances in AI might be applied more directly to the goal of understanding program function by constructing prose explanations of what they do. We imagine a tool that reverse engineers could use to generate English descriptions of the functionality they are currently attempting to reverse engineer. The user would provide, as input, some small piece of machine code, say a function and the tool would return an English description such as *This function takes an array of 32 bit integers and returns the average of the positive elements* as output.

Building such a tool is a large task and we identified two requirements

1. A dataset is required containing exemplars that are paired examples of binary code and word-and-sentence descriptions for training an AI model on the task of program understanding
2. Embeddings for binary code and for the prose used to describe code must exist and operate at the correct semantic level for learning the program understanding concept

It was the goal of this study to assess how much of a hurdle these requirements represent before deciding if a larger investment in this endeavor was warranted. Without an appropriate dataset, training an AI would obviously be impossible. The authors initially believed that the quality of available datasets might be a significant obstacle in building the tool. Therefore, we deemed early identification and evaluation of possible datasets a strong indicator for our future likelihood of success. The second hurdle identified is the availability of embeddings suitable for the task. AI

models (more specifically, neural networks) do not operate directly upon inputs such as machine code nor do they generate output such as English prose natively. Instead, embedding functions are used to map inputs (in this case, features of binaries) to a vector space useable by the network and, then, finally, to map output English prose embeddings to actual English prose output. These are two well studied domains; embeddings for program code and embeddings for english exist and are believed to be of high quality, preserving semantic similarity in the embedding space. We would hope to use such embeddings "off the shelf" rather than train them from scratch, but if the embeddings for English and machine code operate at the wrong level of semantic abstraction, then it will not be possible to learn the concept, i.e., to build an AI to understand programs.

2. BACKGROUND & RELATED WORK

2.1 REVERSE ENGINEERING

Reverse engineering is the process of taking something apart to learn what it does and how it does it. It is commonly done in the cybersecurity domain for malware analysis and vulnerability research. It is a tedious, time consuming, and largely human-driven effort, however there are many tools on the market that attempt to aid in the reverse engineering process [26] [27] [28]. Today, those tools are powered by standard program analysis techniques such as disassembly, decompilation, and debugging.

Program analysis is the field of developing algorithms and tools that take a program as input and provide information about the program in question. These analyses are limited in their capability by Rice's Theorem [29], which states that computing any non-trivial statement about a program is undecidable. Despite this famous result, these tools do greatly increase productivity by presenting data to the user in a much more human friendly format.

2.2 ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Natural language processing (NLP) is the field of machine learning concerned with the ability to understand natural languages and neural machine translation (NMT) is a subfield focused on the translations between them. The problem of generating natural-language prose explanations from the languages of programs—the focus of this report—can potentially be mapped to a NMT problem. A common approach for natural language tasks, where sequences are highly important, is to use recurrent neural networks (RNNs).

2.2.1 RNNs and LSTMs

RNNs are neural network architectures that contain feedback loops in the network connections, allowing information to propagate from one step to another as well as back to itself. Due to their internal memory, RNNs are well suited for sequential data and ultimately allow us to operate on sequences of vectors, rather than fixed-sized vectors, in the input, the output, or both. On long sequences, however, RNNs tend to suffer from unstable gradients and gradually forgetting the first few inputs. LSTMs offer a solution to this in the form of more complex memory cells that can learn to recognize an important input and store it for as long as needed.

2.2.2 Encoder-Decoder Frameworks

When the objective is to map one sequence of arbitrary length to another, encoder-decoder frameworks can be useful. Encoders encode the information from the input sequence into a single vector representation (effectively the encoders last hidden state) and decoders decode this vector into an output sequence. This framework tends to work better than trying to translate one sequence to another on the fly with a single sequence to sequence RNN, specifically in the case where later values in the input affect early values in the output. A major weakness present in this architecture occurs at the final state of the encoder: the encoder has to represent the entire meaning of the input

sequence since that is the only piece available to the decoder when it generates the output. This step can be fairly time consuming and can create an information bottleneck for long sequences.

2.2.3 Attention

We could instead let the decoder have access to all the encoders hidden states and assign a different amount of weight or attention to each of those states at every decoding timestep. Although the addition of this attention allows for much better translations, the use of RNNs still render the computations non-parallelizable.

2.2.4 Transformers

In 2017, Google revolutionized the field of NLP with its paper "Attention is All You Need" [30]. This paper introduces transformers: a new architecture that does away with recurrence entirely and instead relies only on attention to draw global dependencies between input and output, ultimately allowing for significantly more parallelization. This paper led to the development of many influential models such as Googles Bidirectional Encoder Representations from Transformers (BERT) [31] which is used in Googles search engine and OpenAIs Generative Pre-trained Transformer 3 (GPT-3) [32], which is used the OpenAIs ChatGPT [33]. It is this recent advance in NLP and the many projects that have built upon it (described below) that lead us to believe that the application of these technologies might be viable to the problem of generating prose explanations from programs.

2.3 APPLICATIONS OF AI/ML TO REVERSE ENGINEERING

Recently, there have been several advances using new approaches in NLP to augment the reverse engineer process, specifically in the recovery of source code and the recovery of names and types of variables and functions in decompilation.

2.3.1 Recovering Function Information

In Automating Reverse Engineering with Machine Learning Techniques [9], the authors used support vector machines and the Gaussian process to classify subroutines as one of six specified types by viewing some subset of the subroutines instructions, API calls, and neighbor information. Artuso et al. [18] attempts to assign names to assembly functions that would resemble what a human reverse engineer would name it with a simple evaluation metric based on whether tokens of the prediction are present in the actual name.

2.3.2 Recovering Variable Information

In DIRE [17], the authors implement encoders (trained with open source software) for lexical (code tokens) and structural (AST) information with an LSTM model and graph neural network respectively; these encoders, together with attention and an LSTM decoder, are used to predict variable names from decompilation. They can recover around 74% of the original variable names in decompiled code. Direct [19] uses the same tokenizer as DIRE [17] but uses only transformers for its encoder-decoder framework, yielding an increase in accuracy of around 7 percentage points.

In the paper "Augmenting Decompiler Output with Learned Variable Names and Types" [16], the authors improved the output of decompilers with their DIRTY (DecompIled variable ReTYper) tool. They trained a transformer model on open source software to augment the output of the popular reverse engineering tool Hex-Rays [26] by generating names and types for variables that are closer to those a human would choose. Overall, they were able to recover types 75.8% of the time, compared to the baseline of 37.9%, and recover names more accurately than DIRE [17] (81% compared to 74%) and similarly to Direct [19].

2.3.3 Recovering Source Code

In "Towards Neural Decompilation" [12], the authors apply an encoder-decoder NMT model based on LSTMs together with an attention mechanism to the problem of binary decompilation (i.e., the translation of assembly code to source code) with limited code manipulations to their original samples of low and high-level languages. Using multiple iterations of the decompiler, they achieve about 95% success from LLVM IR to C and around 88% on x86 to C.

Coda [13] uses the same LSTM-based encoder-decoder model with attention for decompilation but in a more complex two-phase end-to-end framework. In the first phase, it uses control (i.e., AST) as well as data dependency information for prediction and then, in the second phase, an RNN-based error predictor to identify potential mistakes in the first phase. Coda achieves about 82% accuracy on relatively short and non-complex ISAs and programs (i.e., its operations and structures). The same authors later developed N-Bref [14] using transformers which outperforms their previous work (an accuracy now of 90%) and can better handle longer programs and more complex control flows.

SEAM [11] is a transformer-based model used to recover semantics as well as overall program functionality during decompilation. The authors create and use a new intermediate language to translate high-level languages into a more similar form to low-level ones while still keeping type information present. In parallel, they also use an LSTM model for semantic-based identifier recovery. They achieve an average of 94% program accuracy and 92.64% semantic accuracy on x86_64.

2.4 NEURAL NETWORK EMBEDDINGS

The values input and output by an AI trained to understand programs have to be encoded in some way. If the AI is a neural network of some kind, the most natural inputs and outputs are vectors of real values. An embedding for a neural network corresponds to a mapping between a sparse high-dimensional feature vector space and a lower-dimensional vector space. As a concrete example, consider English words, which are readily encoded "one-hot" by having a neural network layer in which each unit represents a unique word. However, this means a layer with many units, as vocabulary size can be tens of thousands of words. For most tasks, a better encoding would not only be lower-dimensional, but also would represent the words "trout" and "fish" with similar vectors, i.e., that are near one another in the vector space (see Figure 1). Note that, if two words are semantically similar, the fact that they are near one another in the embedding vector space means that they will elicit a similar input to the network, which is desirable.

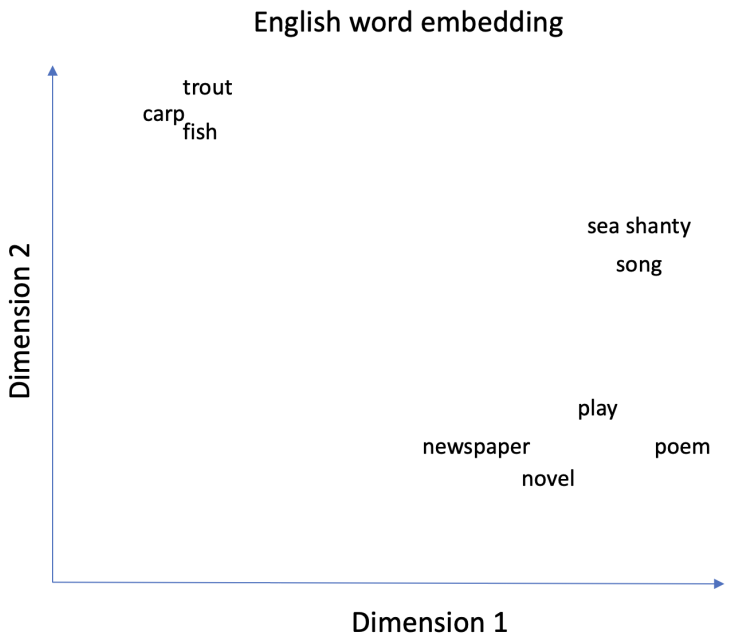

Figure 1. An illustration of a semantics-capturing English word embedding in two dimensions. Words with similar meanings are close to one another. Note that a real embedding would be in hundreds of dimensions, typically.

Embeddings like this can be arbitrary or hand-tuned functions, but they are often, themselves, learned from data. This can mean they are learned as a module in a bigger task—for example English to French translation. In this case, the embedding corresponds to a hidden layer in a larger neural network, and that portion of the network going from inputs, encoded simplistically (for words, perhaps in a one-hot encoding), to the hidden layer, is simply extracted and evaluated for general use. If an embedding for English words works well enough for translations, it is likely to work well for other natural language tasks, or be a good start that can be fine-tuned.

Embeddings can be used to compose higher-level embeddings. For instance, a summarization network would likely employ a trained word-embedding on its input. And, while the network output would be a summary in words, of variable length, the full network would contain a hidden layer of fixed length that, itself, could function as an embedding vector for the summary. Similarly, a network trained to classify or learn useful concepts involving binary code of variable length can have a hidden layer that can suitably be used as an embedding to represent that code's meaning.

3. DATASETS

One of the requirements for training a neural network is a large, high quality dataset. Since we believe the availability of such a dataset would be a major challenge in building our desired system, we decide to investigate datasets for our seedling effort.

3.1 DATASET REQUIREMENTS

We decided upon the following desiderata for an acceptable dataset.

1. **Many exemplars**: In the paper *Scaling Laws for Autoregressive Generative Modeling* [34] the authors state that the effectiveness of autoregressive generative modeling does increase as the size of the training dataset increases, so the larger the dataset, the better. For example, the much discussed GPT-3 [32] large language model was trained on approximately 499 billion tokens.

2. **Source code**: The dataset must contain either assembly code or source code that we can easily compile to machine code. The goal is to evaluate the feasibility of training a neural network to aid in reverse engineering and therefore will be working with machine code. In addition, if the dataset includes source code, it would ideally be C or C++ source since those are the languages most often used to produce the binaries our reverse engineers are interested in.

3. **Prose explanations**: Code must be paired with English language descriptions of the code's functionality at a useful semantic level. This turns out to be a tricky criteria to meet. Source code often contain comments (although not as often as it should) which at first glance appear to provide descriptions. However, based on the previous experience of one of the authors and multiple searches during the our effort, source comments are often at the wrong semantic level. They often describe the functionality of the code within the context of the whole program rather than the section of code on its own. For example, a comment in a web server may say "Serves web page" rather the "Listens on port 80".

3.2 EVALUATED DATASETS

We evaluated the following datasets as part of this seedling effort. These datasets were found as part of a literature review done by the members of the team. Ultimately, we decided to construct our own dataset, as none of these met our our criteria.

3.2.1 CodeXGLUE/Code-Text

This dataset was produced as part of the CodeSearchNet [35] project. The goal of that project was to release a dataset and challenge with hopes to improve semantic code search. The dataset contains functions in six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby) and automatically generated natural language queries. Given that only one of the languages, Go,

can be compiled to binary code and that Go binaries are rarely encountered by reverse engineers, we decided this dataset was insufficient.

3.2.2 Deepcom-Java (BASTS)

This project's goal was to improve code summarization [36]. It involved two separate datasets. One of those datasets was the Python portion of the above CodeXGLUE dataset. The other was a dataset adopted from another project [37] that consisted of Java source code. Since neither Python or Java can be compiled, these datasets were insufficient.

3.2.3 SPoC

Search-based Pseudocode to Code (SPoC) [38] was a project attempting to generate source code from pseudocode. Their dataset consists of 18,356 C++ programs with pseudocode descriptions. While generating pseudocode from machine code is valuable, that generally falls into the work of a decompiler that is a well studied area of program analysis. Given the lack of English labels and the semantic level of the pseudocode, this dataset was found insufficient.

3.2.4 CONCODE

The CONCODE [39] dataset consists of environmental information, comments, and code from public Java code repositories found on GitHub [40]. Given that Java is not compiled to machine code, the dataset was found insufficient.

3.2.5 GitHub-Code

This dataset [41] is a collection of code in various languages. It contains C and C++ code, but does not contain any associated English text. For this reason, it was found insufficient.

3.2.6 CoNaLa

CoNaLa: The Code/Natural Language Challenge [42] is a dataset of code and natural language descriptions of its functionality. This dataset was generated by mining the website Stack Overflow [43]. This dataset is very close to what we were looking for, but was a mismatch for our purposes since it deals in Python code rather than C/C++.

3.2.7 XLCoST

XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence [44] is a dataset consisting of an English problem description and implementations of solutions in up to seven programming languages. The dataset contained both C and C++. We used our framework to evaluate this dataset and it preformed very poorly on our metrics.

3.2.8 HumanEval-X

The HumanEval-X [45] dataset consists of handwritten English statements and implementation in five programming languages: Python, C++, Java, JavaScript, and Go. Although the data

appears to be of very high quality, it only consists of 820 total samples. Given its small size, we determined it would be insufficient for training, but maybe useful in evaluation of a trained model.

3.2.9 Google Code Jam

The Google Code Jam dataset [46] is a collection of problem statements and submitted solutions to the Google Code Jam [47] programming competition. This dataset does include C++ code, however the problem descriptions are very high-level and avoid describing the low-level functionality of the expected solution. For this reason, we found the dataset insufficient for our purposes.

3.3 CONSTRUCTING A DATASET

After our review of available datasets, we decided to construct our own dataset that we hoped would better align with our goals. Taking inspiration from CoNaLa, we used the popular programming question/answer website Stack Overflow [43] as the source of our data. We began with an offline snapshot of the website provided by the Kiwix [48] project. Below we discuss the process we followed to create our dataset.

It is our intention that the dataset constructed for this seedling effort as well as the scripts used to compose it will all be made publicly available to researchers who might find them of use.

3.3.1 Parsing Stack Overflow

The Kiwix [48] project provides snapshots of popular websites in the zim file format [49]. Our first step was to parse all of the pages that were tagged with either C or C++ and convert them to a format more conducive to automated processing. We decided on the jsonl [50] format since it works well with the hugging face [51] datasets library which provides good streaming access to avoid having the whole dataset in memory at once. This process resulted in extracting data from 1107347 questions from the snapshot. Our initial pass over the Stack Overflow (SO) pages extracted data into the schema in Table 1. As each page has one question and one ore more answers, the *answers* field is described separately in Table 2.

TABLE 1

SO Page Extraction Schema

Field Name	Description
`path`	the path in the zim file (used for debugging)
`title`	the title of the question page
`score`	the question's score on Stack Overflow at the time the snapshot was taken
`question_id`	an internal identifier for the question (used for debugging)
	Continued on Next Page

TABLE 1 SO Page Extraction Schema (continued)

Field Name	Description
owner_id	an internal identifier for the user who posted the question
raw	the raw html of the question
raw_hash	a hash of the raw question
tags	a list of user provided tags for the question
snippets	the question processed into snippets (see Snippet Process)
answers	a list of structures representing the answers described below

TABLE 2

SO Page Answer Extraction Schema

Field Name	Description
score	the answer's score on Stack Overflow at the time the snapshot was taken
answer_id	an internal identifier for the answer (used for debugging)
accepted	whether or not the answer had been accepted by the asker of the question
raw	the raw html of the answer
raw_hash	a hash of the raw answer
snippets	the answer processed into snippets (see Snippet Process)

3.3.2 Snippet Processing

Users provide questions and answers to Stack Overflow [43] as free-form HTML without any semantic structure. Code is typically intermixed with data and explanations. To make it easier for later stages of our processing to handle this unstructured data, we process the raw posts into *snippets* that contain the text contained within a set of HTML tags. Thus, a question or an answer to a question consists of one or more snippets, the schema for which appears in Table 3.

TABLE 3

SO Snippet Schema

Field Name	Description
code_tag	whether or not the snippet was formatted as code
	Continued on Next Page

TABLE 3 SO Snippet Schema (continued)

Field Name	Description
`text`	the raw text with all of the tags stripped away

3.3.3 Generating Candidate Programs

After we've parsed the questions and their answers, we endeavor to compile binaries from the source code collected. This is tricky since users provide no semantic information to the site only formatting and often post incomplete or ill-formed source code. We used an algorithm inspired by CoNaLa, but made modifications for compilation to generate our candidate source files.

A single Stack Overflow page contains one question and one or more answers. For the question and each answer, separately, we took all of the snippets that had been formatted as code and compute every permutation of contiguous concatenation. As an example, given the code snippets for a question or answer

a, b, c

we would consider each of the following sequences of snippets as possibly compilable programs, or *candidates*:

a, ab, abc, b, bc, c

We run each of these candidates through a preprocessor that attempts to fix or detect common errors introduced when users are writing code with no requirement that it compile. We then inject each of the candidates into a few C and C++ templates that provide boilerplate code that is often left out when discussing code on the site. Next, we run each candidate through a post processor that attempts to fix any errors introduced in the previous processes, and drop those that, based upon a handful of heuristics, seem unlikely to compile, as compilation is a bottleneck in our process. This gives us a set of candidate source files to validate.

To validate the candidates, we run each through a series of tests. We compile each sample with the appropriate compiler with optimizations enabled and compare the resulting assembly with that of each template with statements that do not result in code injected into them. The dead code elimination optimizations ensure that candidates that contain syntactically valid statements that are not used in computation, such as variables that are declared and never used, are discarded. If the source compiles but generates a binary of the same size as that generated via the C or C++ template with no real code, then the candidate is discarded. In other words, if the candidate compiles but contains nothing but boilerplate machine code, we discard it. Finally, we assemble the candidate assembly and ensure that it is valid. This check eliminates the candidates that contain invalid inline assembly. If a single question or answer results in multiple valid candidates that compile, we retain the longest one. We generate one entry in our dataset per question that generates a valid binary and one per answer that generates a valid binary. This means that, for a given SO page, if it consists of a question and N answers, all of which have at least one valid source candidate, then $N + 1$ entries will be added to our dataset for that single page.

3.3.4 Final Dataset Format

After the above processing, we end up with the follow data structure per *exemplar* or validated source candidate plus associated SO page information, including text but also scores, tags, and other features. Table 4 provides a schema for each exemplar, which can have one or more answers, the schema for which is detailed in Table 5. Our final dataset contains 73209 exemplars.

TABLE 4

Dataset Exemplar Schema

Field Name	Description
`id`	an identifier for the question used a way to reference a particular question
`title`	the title of the question page
`score`	the question's score on Stack Overflow at the time the snapshot was taken
`text`	the concatenation of all of the non-code snippets from the question
`tags`	a list of user provided tags for the question
`source`	the selected candidate source code
`binary`	the BASE64 encoded binary from the compiled and assembled source
`compiled`	true if the source and binary fields were extracted from the question
`answers`	a list of structures described below

TABLE 5

Dataset Exemplar Answer Format

Field Name	Description
`score`	the answer's score on Stack Overflow at the time the snapshot was taken
`accepted`	whether or not the answer had been accepted by the asker of the question
`text`	the concatenation of all of the non-code snippets from this answer
`tags`	a list of user provided tags for the question
`compiled`	true if the source and binary fields were extracted from this answer

4. EMBEDDINGS

In order to evaluate the suitability of datasets for training a large model, we experimented with several domain-specific embedding models to assess correlations between various features in the datasets. This section describes all of the embedding models that we used or considered for evaluating the datasets. Where possible, we selected the current state-of-the art embedding models measured by their performance on benchmark embedding tasks.

4.1 TEXT EMBEDDINGS

The following text embedding models convert sentences of words into semantics-preserving embeddings that can be compared with simple distance metrics. These models can be used to compute the similarity of the ground truth labels/annotations in our datasets.

4.1.1 BERT Hidden State

The simplest method for computing an embedding for text that takes advantage of the latest technology in large-language modeling is to extract the hidden state vector of a pre-trained BERT model. This requires no additional training and similarities between embeddings can be computed simply as the cosine distance of hidden state vectors. For the base model, we used the Detoxify—a BERT-based model fine-tuned to identify toxicity in internet comments [52].

While this approach is relatively easy to implement, it makes a naive assumption that the hidden state vector space learned by the model will contain some amount locality.

4.1.2 Sentence Transformers

Sentence Transformers offers an improvement on the above naive approach by introducing a training objective that optimizes the hidden state vector space for locality—effectively making vector similarity more semantically meaningful [53]. Sentence Transformers adds an additional training step where the model is given pairs of samples with a known level of similarity. Sample pairs are passed forward through the network and their difference computed. The resulting difference value is compared to the ground truth similarity value in the training data and then backpropagated through the entire network to further optimize for similarity.

This model shows the best performance on a range of embedding datasets and so this is the model that we selected for text embedding in evaluating our datasets.

4.1.3 OpenAI Ada

OpenAI provides text embedding models for search, similarity, and classification [54]. We experimented with these models but found Sentence Transformers' open-source model to be both faster and easier to use. Other researchers have also found that OpenAI's models perform worse than the open-source Sentence Transformers library in a text embedding task [55].

4.2 BINARY CODE EMBEDDINGS

The following code embedding models convert sequences of executable instructions into semantics preserving embeddings which can be compared with simple distance metrics. These models can be used to compute the similarity of the binary code samples in our datasets.

4.2.1 BinShot

Ahn et al used a contrastive (Siamese) fine tuning task to build a BERT-based model for binary instruction embedding called BinShot [56]. BinShot provides cross-architecture, program semantics-preserving binary code embeddings and we were able to use the pre-trained models that they published to generate embeddings for the code in our datasets.

4.2.2 Instruction Trace Embedding Model

At the same time as the authors of BinShot were developing their approach, another Lincoln Laboratory program independently designed and trained a very similar model which uses the same contrastive learning approach to fine tune a BERT-based model for binary instruction sequence embedding. The approaches differ slightly in their instruction encoding strategy, but in our evaluation perform very similarly.

This is the primary model we selected for code embedding in evaluating our datasets. For thorough description of the design and architecture of this model, see BinShot [56].

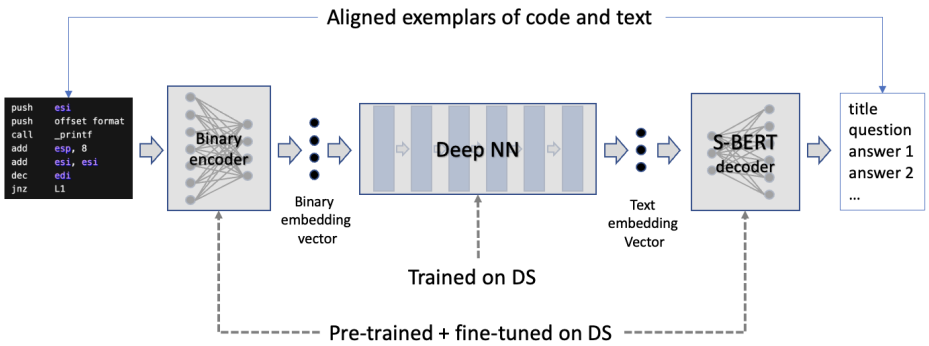

Figure 2. Binary to prose description model. Pre-trained embeddings for binary code and for English prose would be used to encode input and decode output embeddings, but would be fine-tuned on our dataset.

5. EVALUATION

5.1 METHODOLOGY

The dataset described in Section 3.3 consists of N exemplars each of which has a variety of features (see Tables 4 and 5). For the purposes of this evaluation, only the text and binary code features were considered. At a high level, our goal was to test the viability of training a model such as is depicted in Figure 2. In that model, transformer NN models would be employed to map binary code to embedding vectors and also to map text embedding vectors to prose output. Note that a transformer contains both an encoder and a decoder module, and the binary-to-words model proposed, here, employs only the encoder from the binary code model and the decoder from the text model. While high quality models for both kinds of embedding exist and we would be using them "off-the-shelf", their weights would be fine-tuned on our dataset. By contrast, the deep NN in the middle of this model architecture would be trained entirely on our data; its sole job would be to learn the mapping from one embedding space to the other.

Given this dataset and the proposed model architecture, a necessary precondition for success is that the concept represented by the dataset is actually learnable by the deep neural network—i.e., the mapping from binary code embedding space to prose embedding space. Concretely, we assessed this by the following algorithm and analysis, depicted in Figure 3.

1. Choose two exemplars from the dataset, e_1 and e_2
2. Generate the binary embedding vectors for both exemplars: $b(e_1)$ and $b(e_2)$
3. Generate the prose embedding vectors for both exemplars: $p(e_1)$ and $p(e_2)$
4. Compute the distance between the two binary embedding vectors $D(b(e_1), b(e_2))$
5. Compute the distance between the two prose embedding vectors $D(p(e_1), p(e_2))$
6. Repeat many times until you have a large number of pairs $D(b(e_1), b(e_2)), D(p(e_1), p(e_2))$

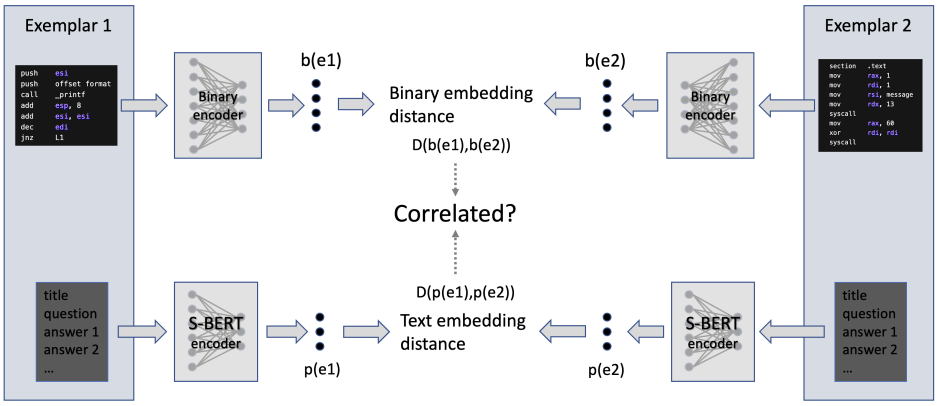

Figure 3. For a particular pair of exemplars, the distance in the binary code embedding space and in the prose embedding space should be correlated. If not, the concept is not learnable.

If we make a scatter plot of $D(b(e_i), b(e_j)), D(p(e_i), p(e_j))$ we would hope the two distances would be correlated, which would manifest as a line-like cloud of points. If that were true, then, certainly, the concept is learnable. If not, it is hard to see how the concept can be learnable. We can also compute the correlation coefficient between the distances in the two different embedding spaces and use standard methods to determine how many pairs of exemplars to use to get statistically significant results. Note that there are no guarantees, here. The embeddings might operate at the wrong level of semantic abstraction for this application. Or, the training data may be of low quality. In either case, this should manifest as low correlation between the distances in the two embedding spaces.

5.2 RESULTS

5.2.1 Methodology Validation

Evaluating the quality of a dataset for training a machine learning model is a complex task - one that is not well-defined and for which there is no standard methodology. To our knowledge, no previous work has used a methodology similar to ours to evaluate sequence-to-sequence, summarization-style datasets. In order to validate our methodology and to understand what "good" performance looks like on a high-quality dataset, we first performed an analysis of the BillSum dataset [57]. BillSum is a dataset of over 20,000 U.S. Congressional bills and reference summaries that is often used as a text summarization benchmark and is widely cited by text summarization papers.

Using the Sentence Transformers text embedding model, we generated and compared embeddings for the full text and the summary of a given bill in the dataset; the results are depicted in Figure 4.

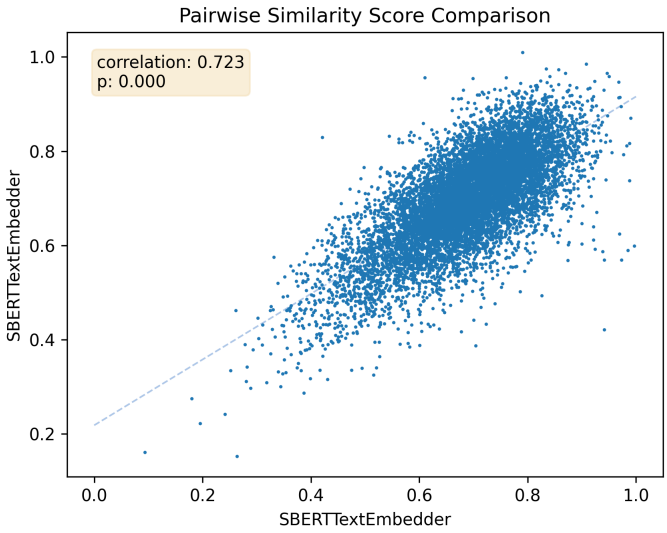

Figure 4. BillSum dataset results. There is a strong, positive correlation (0.723) between the similarity of full bill texts and their summaries, with a p-value less than 0.05. This is a high-quality dataset and this summarization problem should be learnable.

5.2.2 Embedding Distance Correlation Evaluation

We evaluated three different datasets: HumanEval-X [45], XLCost [44], and our own Stack Overflow dataset. We used the Sentence Transformers text embedding model to embed the code summaries, and the instruction trace embedding model to embed the disassembled `.text` section of the associated binary compiled from each sample's source code. Only the `.text` section of the binary is considered so that binary headers, data, and metadata do not affect the embedding.

HumanEval-X Dataset Performance on the HumanEval-X dataset is depicted in Figure 5. Although the HumanEval-X dataset is the best-performing dataset, the correlation here is quite weak compared to our baseline BillSum dataset. A summarization model trained on this data would likely perform poorly.

XLCost Performance on the XLCost dataset is depicted in Figure 6. There is practically no correlation between the code similarity and the text summary similarity in this dataset. A summarization model trained on this dataset would perform extremely poorly. Additionally, the code similarity binning visible in Figure 6 suggests issues with the diversity of the dataset that should be investigated further.

Stack Overflow Dataset Performance on the Stack Overflow dataset, i.e. the one created by the authors and described in Section 3.3 is depicted in Figure 7. Similar to the XLCost dataset

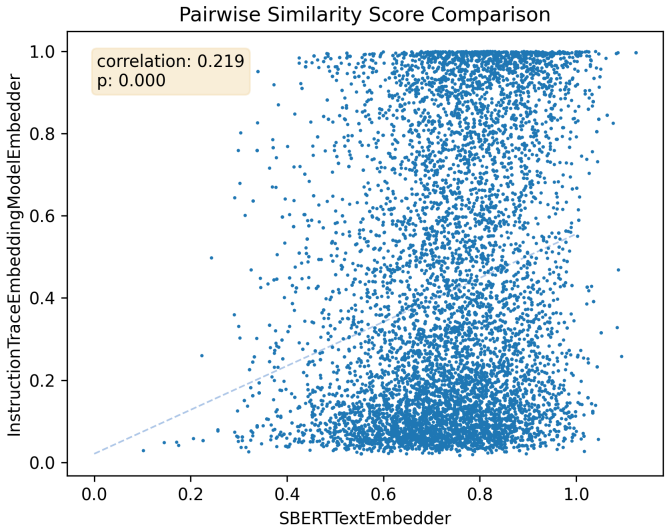

Figure 5. HumanEval-X dataset results. There is a weak, positive correlation (0.219) between the binary code similarity and the text summary similarity, with a p-value less than 0.05. This is not a high-quality dataset and this summarization problem would be difficult for a model to learn.

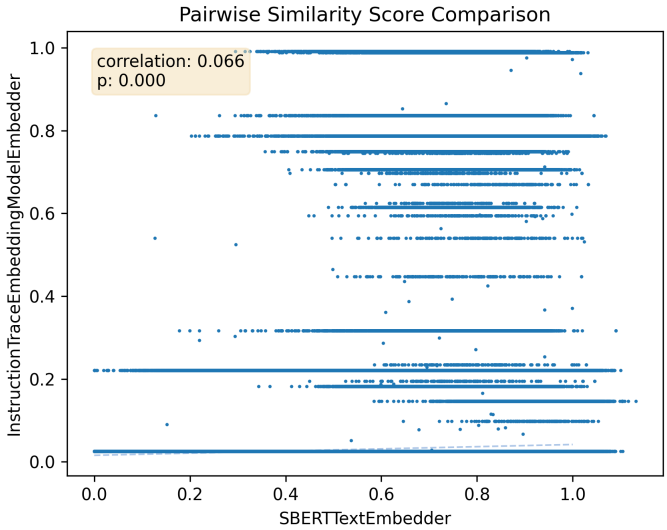

Figure 6. XLCost dataset results. There is nearly no correlation (0.066) between the binary code similarity and the text summary similarity, with a p-value less than 0.05. This is a very low-quality dataset and this summarization problem would be very difficult for a model to learn.

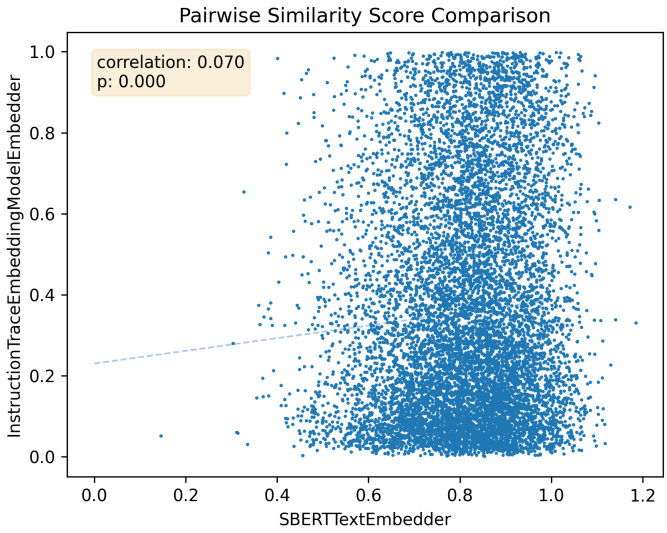

Figure 7. Stack Overflow dataset results. There is a very weak, positive correlation (0.070) between the binary code similarity and the text summary similarity, with a p-value less than 0.05. This is a very low-quality dataset and this summarization problem would be very difficult for a model to learn.

there is unfortunately little to no correlation between the code similarity and the text summary similarity in this dataset. While this dataset does not suffer from the same diversity issues as XLCost, a summarization model trained on it would still likely perform very poorly.

5.2.3 A Manual Survey of the Stack Overflow Dataset

The results in Section 5.2.2 are discouraging. One explanation for them would be that the data is not very useful for learning the concept we wish to learn. You can imagine that almost identical source code could be present in two different SO pages and very different discussions might be going on in the two pages. For example, the source code could be a bubble sort of numbers in an array. In one page, a discussion about the ideas behind and merits of this kind of sort could be playing itself out. In another page, a trivially different implementation could have a buffer overflow that was the topic of the discussion. Another possible explanation for the poor results in terms of embedding distance correlation could be that the embeddings themselves are poorly suited to the task at hand. The SBert embedding might be trained on a lot of data but it might also tend to believe all discussions of program internals are rather close to one another.

As a check on the blind use of data, we decided to perform a manual survey of a subset of the dataset. To that end, a random sample of 100 pairs of binary embeddings for which the distance in the embedding space was less than 0.2 were selected. Similarly, 100 pairs of text embeddings for which the distance was less than 0.5 were selected. These cutoffs were chosen based on Figure 7 to select pairs that seem to be judged similar based on the embedding and distance measure. Note the limited dynamic range of distance values for SBert evident in that figure, for which a distance of 0.5, for the prose in our dataset, is considered quite close. These 100 pairs were divided up into

four sets and presented to the four authors of this report for adjudication. For binaries, the source code used to create the binaries was presented for consideration, along with the distance in the embedding space. For prose, the title and text used were presented for each of the exemplars in the pair, as well as the distance. Adjudicators were asked to grade each pair in one of three ways:

- **Agree**: Agree with distance computed; this pair is similar
- **Unsure**: Unsure if this distance make sense
- **Disgree**: Disagree with distance; pair seems unrelated

TABLE 6

Survey Results

Embedding	c(Agree)	c(Unsure)	c(Disagree)
binary	42	12	46
text	20	10	70

The results of the survey appear in Table 6. Adjudicators agree with the binary embedding distances only 42% of the time, and with the text embedding distances only 20% of the time. This seems strong evidence for something being wrong either with the embedding or with the dataset. Anecdotally, adjudicators noted that they could sometimes see why the text embedding might be judged similar, such as when both contained prefatory remarks such as "This is for an algorithms class and I can't get my code to compile" or "This program compiles fine but segfaults on any input." These aren't really statements about the probable business of the code (the segfault is assumed to be unintentional) yet they might make up a large fraction of the words used.

5.2.4 Existing "Solutions"

Recent advances in the domain of natural language processing have shown that large language models are surprisingly adept at answering complex, nuanced questions in very niche or complicated domains. The release of ChatGPT, a language model designed to interpret and answer questions in an conversational fashion [33], has led researchers in the field of program analysis to wonder if it can be used to summarize code as an aid to reverse engineering. Several plugins for popular disassemblers and reverse engineering tools have been released that submit portions of the code under analysis to large language models and ask for a concise summary of the runtime behavior of the code [58–60]. These GPT-based solutions, if sufficiently capable of summarizing code without any additional training, would seem to obviate the need for a dedicated binary code summarization model (and consequently obviate the need for a high-quality code summarization dataset).

In order to evaluate GPT-based binary code summarization, we evaluated it against the best-performing dataset in our dataset analysis: HumanEval-X [45]. To measure the performance of

these approaches, we queried GPT-3 with the disassembly of the compiled code samples along with a prompt asking for a detailed summary of the code's functionality and informing GPT-3 that the provided assembly was for the x86 architecture and in Intel syntax. Next, similar to our dataset evaluation methodology, we computed pairwise similarity scores between the GPT-3 code summaries and between the ground truth, human generated code behavior annotations in the HumanEval-X dataset using the Sentence Transformers text embedding model again. We then computed the correlation between these two sets of similarity scores to measure the overall performance of the GPT-3 code summaries. The results of this analysis are depicted in Figure 8.

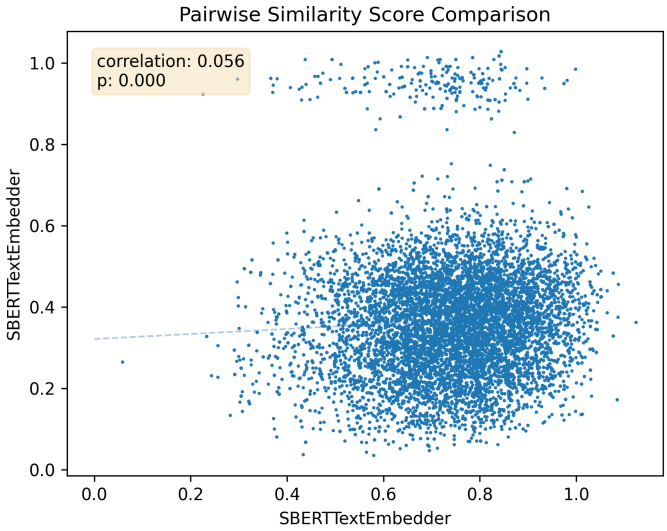

Figure 8. GPT-3 binary code summarization performance. There is a very weak correlation (0.056) between the similarity of the GPT-3 summaries and ground truth data, with a p-value less than 0.05. GPT-3 does a very poor job of summarizing the code in this dataset.

Upon closer inspection, we identified a couple of failure classes exhibited by the GPT-3 model for code summarization.

Convincingly Incorrect The most common type of failure that we saw consisted of a coherent and somewhat convincing description of a program that had nothing to do with the sample provided. This is a common problem when using large language models to answer concrete questions and is often referred to as model "hallucination". For example, GPT-3 gave the following summary for a sample in the dataset:

> *This is a function that calculates the n-th Fibonacci number. It begins by saving the base pointer registry (RBP) onto the stack using the PUSH instruction and assigns the stack pointer value to the RBP registry with MOV...*

The ground truth description for this program has nothing to do with Fibonacci numbers:

> *Given a positive integer n, return the count of the numbers of n-digit positive integers that start or end with 1...*

This error case would be particularly bad if presented to a user that is unfamiliar with reverse engineering who is unlikely to be able to verify the code summary and might be easily convinced by a such a confident-sounding summary.

Generic and Unhelpful The next most common type of failure that we saw consisted of an overly generic and largely unhelpful description of the sample code. For example, one sample was summarized by GPT-3 as:

> *This assembly code snippet is performing a calculation. The calculation breaks down into 5 steps: 1. The code begins with the instruction endbr64, which allows for 64 bit indirect branch enforcement when executed in hardware. This instruction has been added in the x86 instruction set to Offer Enhanced Security...*

This description is largely unhelpful and even appears to contain some text that was copied verbatim from the x86 instruction manual. The ground truth description for this program is:

> *Return median of elements in the vector l.*

Assembly Regurgitation In a small set of cases, the model did not return readable text at all. Instead, it just returned a subset of the provided assembly with no additional context or summary.

6. FUTURE WORK

6.1 DATASETS

We have thoroughly reviewed the datasets, embeddings, and models of today, but machine learning is a rapidly evolving field. According Stanford's Artificial Intelligence Index Report in 2021, private investment in AI was about $93.5 billion and there were 334,497 AI publication [61]. Given the speed at which the field is evolving, what was impossible one day may become trivial the next.

Our work has shown that there is a clear need for additional datasets to support the development of improved binary code summarization models; therefore there are many directions for future work.

6.1.1 Evaluation of New Datasets

New datasets are published frequently in this domain—often, unfortunately, to the detriment of overall research quality. It is nearly impossible to compare the performance of new and competing models unless the community has some set of standard, high-quality datasets. New datasets are released every day and websites such as Hugging Face [51] are collecting those datasets in central repositories. Additionally, private companies are generating internal datasets for their own use. Mandiant, an information security company recently bought by Google, published a blog post [62] that details an internal dataset that appears to match our requirements. In the future, such a dataset may be made public or MIT LL might gain access via government or industry partnerships and the initial work in this project toward a methodology for assessing the quality of these datasets will be instrumental in evaluating new datasets.

6.1.2 Improve Existing Datasets

Another approach would be to improve an existing dataset. Data augmentation is a common technique across various data science and machine learning domains whereby some set of semantics-preserving transformations is used to synthetically expand the size of an otherwise small dataset [63]. Data augmentation strategies could be pursued to multiply the value of higher-quality, hand-annotated datasets. One approach could be to run our English prose through a summarization model with the hopes of generating more potent text. Alternatively, we could process existing, large, low-quality datasets by filtering or weighting samples based on their quality as determined by the methodology outlined in this report.

6.1.3 Create A Dataset

Creating a dataset by manually annotating or writing samples would likely produce a very high-quality dataset. However, creating a sufficiently large dataset for training large models in this way is a daunting task for a small team of engineers. But there are several potential options that could maximize or multiply the value of individual samples created by hand that we have already discussed. A small, handwritten dataset could be augmented using data augmentation strategies, or provide a basis for filtering or weighting samples from a larger dataset.

Conversely, we believe it could be possible to use our novel dataset evaluation methodology (see Section 5.1) to prune or weight exemplars for training, that is, to use it to grade exemplars by quality. Intuitively, we can weight a particular exemplar based upon the correlation between the distances between its embedding vectors and those of a number of other exemplars. An exemplar that is close in both spaces to a number of other exemplars is more likely to be of high quality. And an exemplar that is often close in one space but not in the other is likely of low quality. Note that exemplars that are always far from others in both spaces are harder to judge, as they could merely be distinctive. This seems a fruitful avenue of exploration. If we believe our methodology for judging the overall quality of a dataset is sound, it might also be employed to judge the quality of individual exemplars.

Another option we would likely pursue would be auto-generation of prose "descriptions" from source code and automated Google searching. Many of the exemplars in our SO dataset are trivial one-line programs, which is a problem. As a remedy, we would instead construct our dataset starting with real github repositories of source code for substantial programs. Short sections of instruction traces (perhaps a few hundred instructions) could be extracted from compiled binaries via micro-execution [64]. And descriptions, of a sort, could be assembled by mining the corresponding source code for comments, variable and type names, called and calling function names, and strings. These might be augmented by Google searches and examination of output to identify likely explanatory prose. Finally, this jumble of words could be passed through a well-performing summarization NN which might generate cogent descriptions to be used either additionally or instead as the target concept to be learned.

Note that these last two techniques can be used in tandem to better effect than either alone. On the one hand, the proposed auto-generation of prose descriptions can generate a potentially very large number of exemplars, without supervision. On the other hand, our embedding distance correlation method, repurposed for evaluating individual exemplars can prune away (or at least de-weight) worse examples, leaving a higher quality dataset for training.

6.2 MODEL ADVANCES

Current approaches to binary code summarization look very similar to naive, baseline summarization models developed by the natural language processing community for text summaries. As we have shown, using generative models for summarization (like GPT-3) does not work particularly well for binary code—this is not a solved problem. In the NLP domain, models fine-tuned specifically for translation tasks offer state-of-the-art performance for summarization tasks. Assuming a reasonably high-quality dataset can be built, we have presented the basis of a translation model that could be fine tuned for binary code summarization. Using pre-trained models for binary code encoding and text generation, a fine tuning task to link the two in a full translation pipeline should require significantly less data than attempting to train a full summarization pipeline from scratch. More work could be done to fully design this model and assess exactly how much data would be required for training. This work could inform dataset requirements and give a clear direction on the suitability of the dataset generation strategies we have already discussed.

7. CONCLUSIONS

In this report, we detailed the findings of the Undertale seedling study to investigate the feasibility of training an AI to produce word-and-sentence descriptions of the purpose of binary code. We began by defining some characteristics of a dataset that would make it desirable for this work, including dataset size, existence, and characteristics of source code, and semantic traits of the corresponding prose descriptions. Using these characteristics, we identified and obtained several existing datasets in order to assess their suitability. Two were judged interesting and were used in our evaluation in some form. However, none of the datasets were exactly to purpose, and, therefore, we constructed a new dataset using code from posts on the popular programming question and answer platform Stack Overflow along with labels dervied from the corresponding text from each post. In the course of this study, we developed a novel method for evaluating dataset quality by considering pairs of dataset exemplars, projecting the code and prose onto their respective embedding vector spaces, and then computing the correlation between distances between the vectors in the two spaces. It was expected that a high-quality dataset would manifest a correlation between these distances, and that a low-quality dataset would not. We validated this methodology on a known high-quality dataset, which, indeed, displays a strong correlation between the distances in its two embedding spaces. Conversely, we found that the new dataset we had constructed from Stack Overflow was likely of low quality as the distances in the two embedding spaces of binaries and prose were uncorrelated.

The major contributions of this project are as follows:

- A characterization of the features of an "ideal" dataset for binary code summarization.
- A collection of the existing datasets that might have been used for binary code summarization.
- A new dataset generated from Stack Overflow posts.
- A novel method for evaluating data set quality for the purpose of training a neural network.
- An evaluation of several datasets as well as a manual survey of dataset quality.
- An assessment of existing solutions for binary code summarization on one of the datasets we evaluated, along with characterization of the limitations and error cases observed.

Our dataset evaluation results were conclusive but hardly encouraging—none of the existing datasets nor our new dataset were able to match the quality of a similar, high-quality summarization datasets from a different domain. Additionally, our manual survey of the dataset we generated from Stack Overflow indciate that simple, automated dataset harvesting and generation strategies are not adequate to generate binary code summary datasets that encapsulate a meaningful and useful notion of summarization to an analyst. Furthermore, we evaluated existing GPT-based binary code summarization tools and found their output to be worse than our dataset whilst being, at the same time, precise and incorrect. In this domain, large language models appear to either hallucinate very specific wrong answers or output their input without commentary; neither is useful for our purpose.

Useful binary code summarization may be possible, given a model more specifically designed for it and a higher quality dataset. Several directions for future work that could make these a reality are outlined, including a dataset pruning or weighting strategy based upon our embedding distance dataset quality measure, and a dataset construction approach involving gleaning summary-like information from source code from large open source software repositories.

GLOSSARY

AI	Artificial Intelligence
ML	Machine Learning
GCC	GNU Compiler Collection
GPT-3	Generative Pre-trained Transformer 3
JSON	JavaScript Object Notation
jsonl	JSON Lines
HTML	Hyper Text Markup Language
BASE64	an encoding where 24 bits are represented as four 6-bit printable characters
BERT	Bidirectional Encoder Representations from Transformers

This page intentionally left blank.

NOTATION

e_i	an exemplar from a dataset
$b(e_i)$	the binary embedding vector of an exemplar
$p(e_i)$	the prose embedding vectors of an exemplar
$D(v_1, v_2)$	the distance between to vectors

This page intentionally left blank.

REFERENCES

[1] "Announcement of white house national cyber workforce and education summit," URL https://www.whitehouse.gov/briefing-room/statements-releases/2022/07/18/announcement-of-white-house-national-cyber-workforce-and-education-summit/, UNCLASSIFIED.

[2] "The benefits of ai-driven scheduling," URL https://zira.ai/articles/benefits-of-ai-created-schedules/, UNCLASSIFIED.

[3] "Using artificial intelligence as a scheduling tool," URL https://www.celayix.com/blog/using-artificial-intelligence-as-a-scheduling-tool/, UNCLASSIFIED.

[4] "Ai powered auto scheduling." URL https://www.rotageek.com/rota-auto-scheduling, UNCLASSIFIED.

[5] "Quantiful," URL https://quantiful.ai, UNCLASSIFIED.

[6] "remi," URL https://www.remi.ai, UNCLASSIFIED.

[7] "Hire for potential with deep learning ai," URL https://eightfold.ai/products/talent-acquisition/, UNCLASSIFIED.

[8] "findem," URL https://www.findem.ai/, UNCLASSIFIED.

[9] B. Anderson, C. Storlie, M. Yates, and A. McPhall, "Automating reverse engineering with machine learning techniques," in *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, New York, NY, USA: Association for Computing Machinery (2014), AISec '14, p. 103112, URL https://doi.org/10.1145/2666652.2666665, UNCLASSIFIED.

[10] D.S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), pp. 346–356, UNCLASSIFIED.

[11] R. Liang, Y. Cao, P. Hu, J. He, and K. Chen, "Semantics-recovering decompilation through neural machine translation," (2021), URL https://arxiv.org/abs/2112.15491, UNCLASSIFIED.

[12] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," (2019), URL https://arxiv.org/abs/1905.08325, UNCLASSIFIED.

[13] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao, "Coda: An end-to-end neural program decompiler," in H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, Curran Associates, Inc. (2019), vol. 32, URL https://proceedings.neurips.cc/paper/2019/file/093b60fd0557804c8ba0cbf1453da22f-Paper.pdf, UNCLASSIFIED.

[14] "Introducing n-bref: a neural-based decompiler framework," URL https://ai.facebook.com/blog/introducing-n-bref-a-neural-based-decompiler-framework/, UNCLASSIFIED.

[15] R. Liang, Y. Cao, P. Hu, and K. Chen, "Neutron: an attention-based neural decompiler," *Cybersecurity* 4, 5 (2021), UNCLASSIFIED.

[16] Q. Chen, J. Lacomis, E.J. Schwartz, C.L. Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association (2022), pp. 4327–4343, URL https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin, UNCLASSIFIED.

[17] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2019), pp. 628–639, UNCLASSIFIED.

[18] F. Artuso, G.A. Di Luna, L. Massarelli, and L. Querzoni, "In nomine function: Naming functions in stripped binaries with neural networks," (2019), URL https://arxiv.org/abs/1912.07946, UNCLASSIFIED.

[19] V. Nitin, A. Saieva, B. Ray, and G.E. Kaiser, "Direct : A transformer-based model for decompiled identifier renaming," in *NLP4PROG* (2021), UNCLASSIFIED.

[20] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," (2018), URL https://arxiv.org/abs/1807.05620, UNCLASSIFIED.

[21] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "Stateformer: Fine-grained type recovery from binaries using generative state modeling," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA: Association for Computing Machinery (2021), ESEC/FSE 2021, p. 690702, URL https://doi.org/10.1145/3468264.3468607, UNCLASSIFIED.

[22] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Compiler Construction: 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 13*, Springer (2004), pp. 5–23, UNCLASSIFIED.

[23] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *NDSS* (2011), UNCLASSIFIED.

[24] R. Clayton, S. Rugaber, and L. Wills, "On the knowledge required to understand a program," in *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No. 98TB100261)*, IEEE (1998), pp. 69–78, UNCLASSIFIED.

[25] A. Cozzie, F. Stratton, H. Xue, and S.T. King, "Digging for data structures." in *OSDI* (2008), vol. 8, pp. 255–266, UNCLASSIFIED.

[26] "hex-rays," URL https://hex-rays.com, UNCLASSIFIED.

[27] "ghidra," URL https://ghidra-sre.org, UNCLASSIFIED.

[28] "Binary ninja," URL https://binary.ninja, UNCLASSIFIED.

[29] "Rice's theorem," URL https://mathworld.wolfram.com/RicesTheorem.html, UNCLASSIFIED.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," (2017), URL https://arxiv.org/abs/1706.03762, UNCLASSIFIED.

[31] J. Devlin, M.W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," (2018), URL https://arxiv.org/abs/1810.04805, UNCLASSIFIED.

[32] T.B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D.M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," (2020), UNCLASSIFIED.

[33] "Chatgpt: Optimizing language models for dialogue," URL https://openai.com/blog/chatgpt/, UNCLASSIFIED.

[34] T. Henighan, J. Kaplan, M. Katz, M. Chen, C. Hesse, J. Jackson, H. Jun, T.B. Brown, P. Dhariwal, S. Gray, C. Hallacy, B. Mann, A. Radford, A. Ramesh, N. Ryder, D.M. Ziegler, J. Schulman, D. Amodei, and S. McCandlish, "Scaling laws for autoregressive generative modeling," (2020), URL https://arxiv.org/abs/2010.14701, UNCLASSIFIED.

[35] H. Husain, H.H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436* (2019), UNCLASSIFIED.

[36] C. Lin, Z. Ouyang, J. Zhuang, J. Chen, H. Li, and R. Wu, "Improving code summarization with block-wise abstract syntax tree splitting," (2021), URL https://arxiv.org/abs/2103.07845, UNCLASSIFIED.

[37] G. Yang, X. Chen, J. Cao, S. Xu, Z. Cui, C. Yu, and K. Liu, "Comformer: Code comment generation via transformer and fusion method-based hybrid code representation," (2021), URL https://arxiv.org/abs/2107.03644, UNCLASSIFIED.

[38] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P. Liang, "Spoc: Search-based pseudocode to code," (2019), UNCLASSIFIED.

[39] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," (2018), URL https://arxiv.org/abs/1808.09588, UNCLASSIFIED.

[40] "Github," URL https://github.com, UNCLASSIFIED.

[41] "Github code dataset," URL https://huggingface.co/datasets/codeparrot/github-code, UNCLASSIFIED.

[42] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *International Conference on Mining Software Repositories*, ACM (2018), MSR, pp. 476–486, UNCLASSIFIED.

[43] "stack overflow," URL https://stackoverflow.com, UNCLASSIFIED.

[44] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C.K. Reddy, "Xlcost: A benchmark dataset for cross-lingual code intelligence," (2022), URL https://arxiv.org/abs/2206.08474, UNCLASSIFIED.

[45] "Humaneval-x," URL https://huggingface.co/datasets/THUDM/humaneval-x, UNCLASSIFIED.

[46] "Google code jam dataset," URL https://github.com/Jur1cek/gcj-dataset, UNCLASSIFIED.

[47] "Google code jam," URL https://codingcompetitions.withgoogle.com/codejam/archive, UNCLASSIFIED.

[48] "Kiwix project," URL https://www.kiwix.org, UNCLASSIFIED.

[49] "openzim," URL https://openzim.org, UNCLASSIFIED.

[50] "Json lines," URL https://jsonlines.org, UNCLASSIFIED.

[51] "Hugging face," URL https://huggingface.co, UNCLASSIFIED.

[52] L. Hanu and Unitary team, "Detoxify," (2020), URL https://github.com/unitaryai/detoxify, UNCLASSIFIED.

[53] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084* (2019), UNCLASSIFIED.

[54] OpenAI, "New and improved embedding model," (2022), URL https://openai.com/blog/new-and-improved-embedding-model/, UNCLASSIFIED.

[55] N. Reimers, "Openai gpt-3 text embeddings - really a new state-of-the-art in dense text embeddings?" (2022), URL https://medium.com/@nils_reimers/openai-gpt-3-text-embeddings-really-a-new-state-of-the-art-in-dense-text-embeddings-6571f UNCLASSIFIED.

[56] S. Ahn, S. Ahn, H. Koo, and Y. Paek, "Practical binary code similarity detection with bert-based transferable similarity learning," in *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)* (2022), UNCLASSIFIED.

[57] A. Kornilova and V. Eidelman, "Billsum: A corpus for automatic summarization of us legislation," *arXiv preprint arXiv:1910.00523* (2019), UNCLASSIFIED.

[58] "Polar: A lldb plugin which queries openai's davinci-003 language model to explain the disassembly," URL https://github.com/ant4g0nist/polar, UNCLASSIFIED.

[59] "Ghidra-chatgpt," URL https://github.com/SourceDiver42/Ghidra-ChatGPT, UNCLASSIFIED.

[60] "Ghidrachatgpt," URL https://github.com/likvidera/GhidraChatGPT, UNCLASSIFIED.

[61] D. Zhang, N. Maslej, E. Brynjolfsson, J. Etchemendy, T. Lyons, J. Manyika, H. Ngo, J.C. Niebles, M. Sellitto, E. Sakhaee, Y. Shoham, J. Clark, and R. Perrault, "The ai index 2022 annual report," (2022), URL https://arxiv.org/abs/2205.03468, UNCLASSIFIED.

[62] "Annotating malware disassembly functions using neural machine translation," URL https://www.mandiant.com/resources/blog/annotating-malware-disassembly-functions, UNCLASSIFIED.

[63] C. Shorten and T.M. Khoshgoftaar, "A survey on image data augmentation for deep learning," *Journal of Big Data* 6(1), 60 (2019), URL https://doi.org/10.1186/s40537-019-0197-0, UNCLASSIFIED.

[64] P. Godefroid, "Micro execution," in *Proceedings of the 36th International Conference on Software Engineering* (2014), pp. 539–549, UNCLASSIFIED.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
26-04-2023	Project Report	

4. TITLE AND SUBTITLE

On the Feasibility of Training an AI to Understand Programs: FY23 Cyber Security Line-Supported Program

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

A.T. Davis, A.M. Interrante-Grant, H.N. Preslier, T.R. Leek

5d. PROJECT NUMBER
2231-4901

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02421-6426

8. PERFORMING ORGANIZATION REPORT NUMBER

LSP-375

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02421-6426

10. SPONSOR/MONITOR'S ACRONYM(S)
MIT LL

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

13. SUPPLEMENTARY NOTES

13. ABSTRACT
In this study, the possibility of training an AI on the task of program understanding was investigated. Specifically, the AI would take, as input, mechanically extracted features of programs and output English word-and-sentence descriptions of functionality. This output would be expected to aid a reverse engineer in investigating the capabilities and vulnerabilities of a piece of software. The input features might be static, meaning they are gleaned only from inspection of the software, or they might be dynamic, meaning they are extracted from program executions. In this seedling study, we investigated a number of recent publications, existing datasets, data sources, and embeddings1 for binaries and English prose. As part of our study, we constructed a novel dataset, which will be made available to the research community for general use. In brief, the results of this study are twofold. First, the dataset we constructed from over a million stack overflow pages is not of high enough quality to be used in training an AI for program understanding. Further, there is some evidence that the embedding used for English prose is too coarse for our purpose, conflating concepts we would have hoped it to distinguish. This report concludes with some ideas for future investigations including using our dataset quality measures to identify or weight higher quality exemplars, and some ideas involving using prose extracted from source and auto-generated web searches.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	None	52	**19b. TELEPHONE NUMBER** *(include area code)*
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED			