2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-5278-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/SANER56733.2023.00033

Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binaries

Ali Al-Kaswan
Delft University of Technology
Delft, The Netherlands
a.al-kaswan@tudelft.nl

Toufique Ahmed
University of California, Davis
Davis, California, USA
tfahmed@ucdavis.edu

Maliheh Izadi
Delft University of Technology
Delft, The Netherlands
m.izadi@tudelft.nl

Anand Ashok Sawant
University of California, Davis
Davis, California, USA
asawant@ucdavis.edu

Premkumar Devanbu
University of California, Davis
Davis, California, USA
ptdevanbu@ucdavis.edu

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
arie.vandeursen@tudelft.nl

***Abstract*—*Binary reverse engineering* is used to understand and analyse programs for which the source code is unavailable. Decompilers can help, transforming opaque binaries into a more readable source code-like representation. Still, reverse engineering is difficult and costly, involving considering effort in labelling code with helpful summaries. While the automated summarisation of decompiled code can help reverse engineers understand and analyse binaries, current work mainly focuses on summarising source code, and no suitable dataset exists for this task. In this work, we extend large pre-trained language models of source code to summarise de-compiled binary functions. Furthermore, we investigate the impact of *input* and *data properties* on the performance of such models. Our approach consists of two main components; the data and the model. We first build CAPYBARA, a dataset of 214K decompiled function-documentation pairs across various compiler optimisations. We extend CAPYBARA further by removing identifiers, and deduplicating the data. Next, we fine-tune the CodeT5 base model with CAPYBARA to create BinT5. BinT5 achieves the state-of-the-art BLEU-4 score of 60.83, 58.82 and, 44.21 for summarising source, decompiled, and obfuscated decompiled code, respectively. This indicates that these models can be extended to decompiled binaries successfully. Finally, we found that the performance of BinT5 is not heavily dependent on the dataset size and compiler optimisation level. We recommend future research to further investigate transferring knowledge when working with less expressive input formats such as stripped binaries.**

***Index Terms*—Decompilation, Binary, Reverse Engineering, Summarization, Deep Learning, Pre-trained Language Models, CodeT5, Transformers**

I. Introduction

Reverse engineering binary programs has many applications, in particular, software security [1]. Binary reverse engineering is a hard task, requiring highly skilled reverse engineers [1, 2]. Disassemblers and decompilers can help in this process. Disassemblers transform the binary into a low-level intermediate representation, and decompilers lift the representation to a high-level programming language-like representation. But the output of decompilers is still difficult to read and understand [1, 3]. Much of the work that goes into reverse engineering a binary is spent labelling functions with semantic descriptions [1]. Current approaches [4–10] mainly focus on recovering aspects lost in the compilation and decompilation process, such as names and types. Existing works fail to address the inherent difficulties in binary code comprehensibility, namely, the need for a high-level overview of the code.

For source code, methods exist to automatically generate summaries from code [11, 12]. Source code summarisation is used to automatically generate short natural language descriptions of code, which support program comprehension and aid maintenance [12, 13]. While these methods have been successfully applied to programming languages such as Python, Java and PHP [14–16], using pre-trained language models [14–16], none of these methods has been applied to the relatively syntactically-poor output of decompilers (see Figures 1a and 1b). Being able to quickly determine the context and application of a function, can save valuable analysis time, and greatly benefit reverse engineers. Function and variable names alone, are inadequate representations of the source code [12], which is why having descriptive summaries of binaries is desirable.

Following [17], source code can be described as having two information channels: the algorithmic channel and the natural language channel. The algorithmic channel specifies the execution of a program (semantics), while the natural language channel explains the purpose and context of the program to humans [17]. The natural channel includes function and variable names, code comments and the specific human-readable structure of programs. Processors only consider the algorithmic channel to execute a program, while humans use both the algorithmic channel and the natural channel to understand a piece of code [17]. Furthermore, code is very regular and predictable, even more so than natural languages [18]. The compilation process, which transforms readable code into executable binaries, removes much of the information contained in the natural channel. Especially stripped binaries — binaries of which the symbol table is removed — are challenging, since they have almost no identifiers at all as

2640-7574/23/$31.00 ©2023 IEEE
DOI 10.1109/SANER56733.2023.00033

can be observed in Figure 1c.

The goal of this paper is to advance the field of binary reverse engineering by exploring the application of code summarisation to decompiled binaries by taking advantage of source code pre-trained language models.

However, there exists no dataset of aligned binaries and source code summaries since this is a new and unexplored task. As pointed out by LeClair and McMillan, the lack of standardised datasets is a major barrier to ongoing research, which we will address for this task [19]. In this paper, we create a dataset containing pairs of decompiled and stripped-decompiled functions and summaries of these functions. During the creation of this dataset, we conform to the current best practices for dataset construction [19, 20].

We apply this dataset to an existing pre-trained language model using transfer learning, by fine-tuning this pre-trained model on our dataset. For this task, we selected a pre-trained CodeT5 model, which was only trained on source code [14].

We perform experiments on this model to explore the impact of decompilation, and the importance of identifiers. Furthermore, we explore the impact of compiler optimisation levels, the dataset size and the level of duplication.

Our findings are that the decompilation and alignment of stripped functions has a very high failure rate; and the resulting stripped model has low performance. But, we found that the model shows state-of-the-art performance with both decompiled code as well as demi-stripped stripped code, code of which the identifiers were removed after decompilation. Our experiments on data duplication and dataset size further show that these models can be trained with few data, and that while duplicates have a high impact on performance, their presence is not paramount to model performance.

Our key result: *language models pre-trained on source code can be fine-tuned on binaries, opening up a range of new possibilities for the automated analysis of binaries.*

To summarise, the main contributions of this paper are:

- CAPYBARA[1], a dataset of ***C**ombined **A**ligned de-com**P**iled **B**inar**Y** code **A**nd **R**elated **A**nnotations*. A novel dataset of aligned, C, decompiled, stripped-decompiled and demi-stripped summary pairs[2] (Section III);
- BinT5[3], a **Bin**ary summarisation Code**T5** model, a simple and straightforward adaptation of a source code trained code summarisation model to decompiled code using CAPYBARA (Section IV);
- An empirical investigation on the impact of the properties of decompiled code and the properties of CAPYBARA (Sections V and VI);

The materials, including the processed and raw data[1], the trained model checkpoints and steps to replicate our experiments[3], are openly available in our replication package[4].

[1]CAPYBARA: https://doi.org/10.5281/zenodo.7229809
[2]Decompiled code with strip-like obfuscation applied
[3]BinT5: https://doi.org/10.5281/zenodo.7229913
[4]Replication package: https://github.com/AISE-TUDelft/Capybara-BinT5

II. Background

In this section, we introduce the background of compilers, binary reverse engineering, transfer learning and the code summarisation task.

A. Compilers and Optimisation Levels

Compilers are programs that convert source code from one programming language to another, but generally, and in the context of this work, the term is used to refer to programs that translate high-level code, like C, to a lower-level language such as machine code or bytecode. For our work, we focus on the GNU Compiler Collection (GCC)[5] and Clang/LLVM (Clang)[6].

Compilers feature optimisation levels. Generally, the goal of optimisations is the improvement of runtime performance or program size at the expense of compilation time and the ability to debug [21].

By default, if GCC is invoked without any optimisation options, the program will be compiled with -O0. -O1, -O2 and -O3 incrementally apply more optimisation to the binary at the expense of a higher compilation time [22]. Optimisations can restructure and transform the program in relation to the source code, by changing the control flow or the data of the program [23]. This obfuscation can complicate the reverse engineering process by reducing the accuracy of tools [23].

B. Ghidra

Ghidra[7] is a free and open-source reverse engineering toolkit developed by the US National Security Agency. Ghidra contains many separate analysis modules that allow a reverse engineer to analyse binaries. Ghidra features a disassembler, which assembles binaries back into an intermediate representation. In the case of x86-x64 binaries like the binaries this work focuses on, the intermediate representation will be the Assembly language. The decompiler, on the other hand, is a processor language-agnostic transformation engine that takes the disassembled code and creates a source code representation, namely pseudo-C. Pseudo-C follows the general language conventions of C, but it cannot be compiled.

Observe the relatively simple `rtp_sess_ssrc` function from `creytiv/re`[8] shown in Figure 1a. We compile the project using the -O3 compiler level as defined in the project. We decompile the binaries using Ghidra's decompiler using the standard configuration, the resulting pseudo-code is shown in Figure 1b. We observe that aside from the function name, almost the entire natural channel has been destroyed by the compilation and decompilation process. The parameter and variable names are gone, any documentation is removed and the relatively simple logic has been unrolled to a much more difficult-to-understand representation. Ghidra also incorrectly labelled many of the variable types and failed to identify the `struct` datatype.

[5]GCC: https://gcc.gnu.org/
[6]Clang: https://clang.llvm.org/
[7]Ghidra: https://ghidra-sre.org/
[8]re: https://github.com/creytiv/re

```c
/**
 * Get the Synchronizing source for an RTP/RTCP
↪  Socket
 * @param rs RTP Socket
 * @return Synchronizing source
 */
uint32_t rtp_sess_ssrc(const struct rtp_sock *rs){
        return rs ? rs -> enc.ssrc : 0;}
```

(a) Source rtp_sess_ssrc function

```c
ulong rtp_sess_ssrc(long param_1){
    uint local_14 ;
    if (param_1 == 0){
        local_14 = 0;
    } else {
        local_14 = * (uint *) (param_1 + 4);}
    return (ulong) local_14;
}
```

(b) Decompiled rtp_sess_ssrc function

```c
ulong FUN_00100d30 ( long param_1 ){
    uint local_14 ;
    if (param_1 == 0) {
        local_14 = 0 ;
    } else {
        local_14 = * (uint *) (param_1 + 4);}
    return ( ulong ) local_14 ;}
```

(c) Stripped decompiled rtp_sess_ssrc function

Fig. 1: Example source, decompiled and stripped code snippet

Using our trained BinT5 model we can summarise the decompiled code and generate the following summary: Get the source for an RTP/RTCP Socket. This summary gives us an indication of the purpose of the function. Integrating this generated summary into Ghidra increases the readability of the entire binary. Keep in mind that a reverse engineer has to understand not just this function, but hundreds of different functions in a single binary.

C. Stripping

Aside from compiling with higher optimisation levels, binaries can also be stripped to obfuscate the underlying code and to resist analysis [24]. Commercial off-the-shelf software is often stripped to reduce the memory and storage footprint of the binaries, and to resist analysis to protect the intellectual property of the creator. Many vulnerable and malicious binaries are, unfortunately, also stripped to resist security analysis and hide their faults [5].

Unix and Unix-like operating systems include a strip utility. The strip utility removes any operands that are not necessary for the execution of the binary while ensuring that the execution of the binary remains unchanged. The exact implementation and what constitutes unnecessary operands are left to the implementor.[9] The strip utility as implemented in GNU/Linux removes the symbol table from the binary. The symbol table contains each symbol's location, type and name.

Like higher optimisation levels, the use of stripping can greatly complicate the efforts to reverse engineer a binary, as well as reduce the accuracy and effectiveness of reverse engineering tools [24].

For example, we compile, strip and decompile the function in Figure 1a, and the resulting stripped decompiled function is shown in Figure 1c. In addition to the details lost by the decompilation process, the stripper removed all symbols, like the function names.

D. Code Summarisation Task:

Code summarisation (also referred to as source code summarisation) is the task of writing short descriptions from source code, usually a single-sentence summary of the source code. The main use is for software documentation, like the one-sentence JavaDoc description used in Java [19]. This documentation is important for program comprehension and maintenance. But the process of writing and maintaining these descriptions is a labour-intensive and time-consuming task, which is where the benefits of automating that process arise. Automatic code summarisation is an active and popular research problem in the field of software engineering [19].

E. Transformer-based Models

Transformers were originally proposed by Vaswani et al. as a sequence-to-sequence architecture [25]. Unlike the Recurrent Neural Networks [26] (RNN), the Long Short-Term Memory [27] (LSTM) variant of RNNs [26] and Convolutional Neural Networks [28] (CNN), Transformers only use a mechanism called self-attention to capture dependencies between the input and output. The current state-of-the-art NLP models for programming languages such as CodeT5 [14], CodeBERT [15] and PolyGlotCodeBERT [16] are all based on the Transformer architecture [25].

F. Transfer Learning

Pre-trained Transformers-based language models, such as RoBERTa [29], CodeBERT [15] and CodeT5 [14] utilise a pre-train then fine-tune paradigm. The bespoke paradigm was initially introduced by Kenton and Toutanova. In this paradigm, the models are first trained in an unsupervised manner on a large unlabelled dataset. These pre-trained models can then be fine-tuned to perform a more specialised task, such as summarisation. Transfer learning uses the knowledge that is obtained in one task to solve a different task. It allows the creation of general models that are trained once on massive datasets. These general models, which contain general domain knowledge can then be fine-tuned for a specific downstream task. This approach is quicker and requires less training data than training a model on the downstream task from scratch [30].

[9]strip: https://pubs.opengroup.org/onlinepubs/7908799/xcu/strip.html

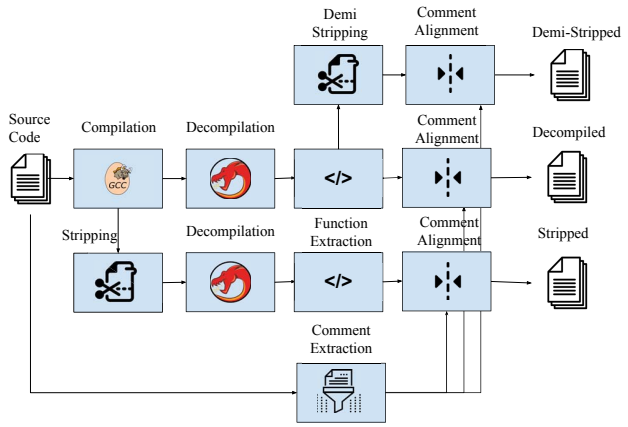

Fig. 2: Data Collection Pipeline

III. CAPYBARA DATASET

We require a dataset of decompiled functions labelled with a descriptive summary to create and assess our solution. This dataset should be relatively large to suit the 'data-hungry' nature of deep-learning models. Furthermore, the dataset needs to feature a diverse set of data representative of our solution's actual real-life use case.

A. Data Collection

To create such a large and diverse dataset we made use of BinSwarm [7], an existing dataset of aligned decompiled and stripped decompiled functions[10]. BinSwarm collects C-based projects from Github. The projects are filtered to only include those that are actively being developed, using Travis CI and built for Ubuntu Linux. The projects are built using Docker. The resulting binaries are then copied and stripped, and both the stripped and unstripped binaries are decompiled using Ghidra. The functions are extracted from the stripped and unstripped decompiled code and aligned with the source code. The BinSwarm dataset only contains aligned tuples of source code and (stripped-) decompiled functions. We extract documentation from the original source code files to add descriptive comments to this dataset. To that end, we depend on the documentation included in the source code by the original authors in the form of single and multiline comments. We locate the functions in the unbuilt project files and align the decompiled functions with the comments in the source code using srcML[11] to extract any documentation located directly before a function signature. A high-level overview of the entire process is shown in Figure 2.

A function's documentation often also contains other details besides the descriptive summary. We found that C projects do not follow a single documentation standard. For example, Javadoc for Java has a short one-line description or summary for each method at the beginning of the multiline comment

[10]BinSwarm: https://hub.docker.com/r/binswarm/cbuilds
[11]srcML: https://www.srcml.org/

```
/** @brief Select the source of Microcontroller Clock Output
 * Exact sources available depend on your target.
 * On devices with multiple MCO pins, this function controls MCO1
 * @param[in] mcosrc the unshifted source bits
 */
```

Fig. 3: Example of documentation from jeanthom/ DirtyJTAG: rcc_set_mco

block. In C, there is no singular documentation standard, so there might not be a single-line summary, and we will need to locate it in the comment block automatically.

a) Summary Extraction Rules: We observe that the majority of single-line data are descriptive summaries, so we extract the first sentence. We identify many documentation styles in our multi-line data, we define some automated rules to extract summaries from the documentation:

- **@brief or @purpose:** If the documentation contains a '@brief' or '@purpose' tag, we extract the first sentence after the tag. The 'brief' tag is part of the Doxygen documentation standard[12], an example is shown in Figure 3[13].
- **Description:** If the documentation contains a line with 'Description:', we extract the following sentence.
- **@param or @v:** Documentation that contains an '@v' or '@param' tag, usually has a summary in the sentence before the tag. We extract that sentence.

b) Filtering Rules: To improve the quality of the dataset we filter out samples based on the rules used by the CodeSearchNet dataset [20] included in the CodeXGlue benchmark for the summarisation task [31]:

- **Documentation length:** We remove any summaries that are too long or too short and remove anything shorter than 3 or longer than 256 tokens.
- **Special tokens:** We follow the example of the CodeSearchNet [20] and remove all documentation that contains special tokens. We scan for web tokens (like 'http://'), HTML tokens (like '<head>'), paths (like 'C://Users/..'), since this documentation usually refers to external resources. We additionally filter any developer tokens (like 'FIXME:'), as these documents do not provide meaningful information about the function itself, but contain comments about the development process.
- **Language:** We filter out any documentation that was not written in English using the FastText language identification algorithm [32]. Around 92.19% of the documentation is in English.
- **Empty documentation:** We find that a large number of functions did not have any documentation associated with them at all. We simply remove these samples from the dataset.

[12]Doxygen:https://doxygen.nl/manual/docblocks.html
[13]jeanthom/DirtyJTAG:rcc_set_mco:https://gitlab.com/insane-adding-machines/unicore-mx/-/blob/master/lib/stm32/common/rcc_common_all.c#L192

- **Abstract Syntax Tree:** The authors of the CodeSearchNet dataset [20] additionally, remove any samples that do not parse into an AST. We choose to omit this step since all of our samples have been successfully compiled and have thus at one point been parsed into an AST by the compiler.

B. Dataset Preparation

a) Synthesis of Demi-stripped Code: From the dataset of decompiled functions, we also create another dataset. We emulate the process of stripping by removing all the identifiers from the decompiled code and replacing them with placeholders. For clarity, we call this demi-stripped data. Like the stripped dataset, the identifiers are all removed, but this is only done after the decompilation process. The decompiler still had access to the identifiers and could use the symbol table during decompilation. Most importantly, this demi-stripped dataset still has the same structure and control flow as the unstripped decompiled dataset and avoids any decompilation issues arising from stripping.

b) Data Split: The dataset is split into a train, test and validation set. These sets constitute approximately, 80%, 10% and 10% [19] of the complete dataset. As recommended by Shi et al. and LeClair and McMillan, we prevent leakage of vocabulary and code patterns between the sets, by sampling the sets in a cross-project manner [13, 19]. This means that an entire project gets assigned to one of the sets, and functions from the same project cannot be assigned to different sets. The projects in the test and validation set are the same across all datasets.

c) Duplication: Large corpora of code, like the corpus gathered by BinSwarm, tend to have a high degree of duplication [19]. As a result, snippets of code that are relatively unchanged appear in multiple parts of the corpus. This can be in the form of copied, generic or auto-generated functions. These functions will appear in multiple repositories and might be duplicated across the training and testing data. Besides exact duplicates, near-duplicates can also occur. Near-duplicates differ in a few minor aspects like additional code comments or different function names. While removing exact duplicates is relatively fast and straightforward, removing near-duplicates is much more challenging and computationally intensive [33]. The issue with code duplication in classical code summarisation is that the models and tools are supposed to be used to generate summaries for new and unseen code. The evaluation metrics should therefore measure the generalisation of the tool on new samples [33]. Duplicates and near-duplicates are not defined as new samples. A user of such a tool could simply look these samples up. Furthermore, large, high-capacity models like CodeT5 with 220M [14] or CodeBERT with 128M [15] parameters, have a large capacity to memorise duplicated code [33].

However, the use case outlined in this work is more akin to deobfuscation. As explained by Allamanis, deobfuscation could be a use case where duplicates are valid and part of the true distribution of the problem [33]. Compiled code contains a lot of duplicate code, and understanding this code is still difficult and essential for understanding the binary. While regular source code allows the reader to look up code snippets, decompiled binaries have an additional obfuscation applied. We, therefore, focus on the model's performance on code with duplicates as we believe duplicates to be part of the true distribution of the data, but we also report the deduplicated results.

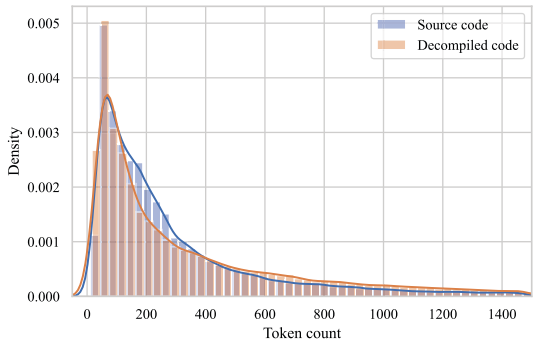

Fig. 4: Tokens in source C and decompiled code

C. Dataset Properties

Table I shows the size of the processed dataset. Of the 2.1M aligned decompiled functions, we extract documentation for 215k of them, and we found that the majority of samples, 1.5M did not have any documentation at all. Furthermore, BinSwarm only provided us with 415k aligned stripped samples, and we can extract documentation for only 14k of these samples.

Dataset	Including duplicates	Deduplicated
C/Demi/Decom	214,587	79,673
Stripped	14,245	7,826

TABLE I: Number of functions in dataset

The vast majority of documentation is in the form of multi-line comments as opposed to single-line or double-slash comments. We found that the documentation and comments had a mean length of 42.60 and 8.14 tokens, respectively.

Figure 4 shows the distribution of the number of tokens in source code and decompiled code. The source and decompiled code have a mean length of 399 and 779 tokens, respectively. Decompiled code also has close to double the LOC of source code, with means of 30.77 and 53.42 lines for source and decompiled, respectively.

The majority of decompiled functions are compiled with optimisation level -O2, with a similar number of -O1 and -O3 samples and relatively few -O0 samples. Stripped data has a very even distribution of optimisation levels, with only -O0 having significantly fewer samples. Note that there are more optimisation levels than shown in Figure 5, for brevity the different levels are grouped into their base optimisation

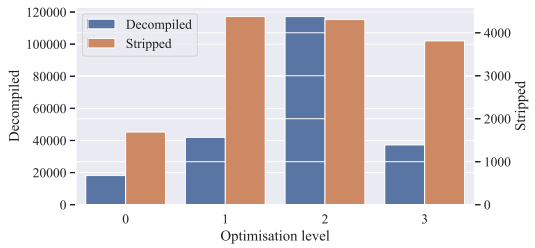

Fig. 5: Distribution of optimisation levels in decompiled (left) and stripped (right)

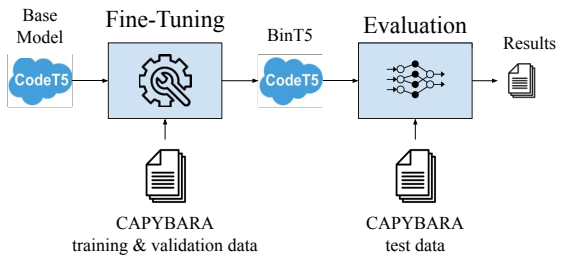

Fig. 6: BinT5 fine-tuning pipeline

level. -Oa is grouped with -O0, -Of and -Og are grouped with -O1, -Os is grouped with -O2. We also observe some samples with an optimisation level higher than -O3 (-O8 and -O7), as specified by the GCC documentation, these levels are equivalent to -O3[14].

IV. BinT5

We select CodeT5 [14] as the base-model for our experiments since it is the highest-scoring publicly-available model on the CodeXGLUE [31] Code Summarisation benchmark[15]. CodeT5 is a programming language model built on the T5 (Text-to-text Transfer Transformer) architecture [34] and pre-trained on a mix of supervised and unsupervised tasks. CodeT5 employs an encoder-decoder architecture. In contrast to other models, CodeT5 is trained using both unimodal (PL only) and bimodal (NL-to-PL) tasks in eight programming languages. This bimodal training allows CodeT5 to perform strong cross-modal tasks such as code summarisation and code generation (PL-to-NL). Many other models only use the data and languages included in the CodeXGlue dataset [15, 16, 31], while CodeT5 also uses a mined dataset of C and C++ code for its pre-training objectives [14]. The inclusion of C training data should help the model with the CAPYBARA dataset. There could be some overlap in the training data between CAPYBARA and the dataset used by Wang et al. which would cause leakage, we address these concerns in Section VII.

CodeT5 also utilises the transfer learning paradigm, which allows us to train the model with relatively little data. In this case, we make use of the CodeT5-base model, which was trained on mixed upstream tasks by the authors [14]. We fine-tune this model on the code summarization task on CAPYBARA. An overview of how we applied the model to create BinT5 is provided in Figure 6.

V. Experimental Setup

To assess the effectiveness of our approach, we first evaluate the performance of the model, we then identify the aspects of the data that make this task inherently difficult, and we finally investigate aspects of the datasets and their influence on the complexity of the task.

[14]GCC optimisation levels: https://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Optimize-Options.html#Optimize-Options

[15]CodeXGLUE benchmark: https://microsoft.github.io/CodeXGLUE/

A. Research Questions

In the context of the study, we thereby formulate the Research Questions (RQ) as follows.

RQ1: *How effective are fine-tuned Transformer-based models at decompiled code summarisation?* To investigate the application of existing models to binaries using CAPYBARA, we set a baseline by training a model on the code summarisation task on the source C-code dataset. We then train a summarisation model on both the decompiled and the stripped dataset. We use the evaluation metrics to compare the performance of the different models.

RQ2: *Which aspects of the input contribute most to model performance?* We investigate which aspects of decompiled code increase the difficulty of the task. We, therefore, look at the impact of the symbol table on decompilation, for this, we fine-tune a model on the demi-stripped dataset and compare it to the other models. We also investigate the importance of the function name by removing *just* the function name from the decompiled code. Furthermore, we investigate the impact of the optimisation level by exploring the performance per optimisation level.

RQ3: *What is the impact of dataset properties on model performance?* We finally investigate how the construction of CAPYBARA influences the models. To answer the final research question we remove the duplicates from the datasets and retrain the models, after which we compare the performance to the baselines. Furthermore, we investigate the impact of dataset size, by incrementally reducing the size of the training sets.

B. Baselines

To first establish a performance baseline, we train a CodeT5-base model on the summarisation task on source C. Note that only samples which are aligned with decompiled code are included in the source C dataset. The baseline is used to compare the decompiled C, stripped decompiled C and the demi-stripped datasets to the source code.

C. Evaluation Metrics

We evaluate the performance between the reference summary from CAPYBARA and the candidate summary produced by BinT5 using the EM, BLEU-4 [35], ROUGE-L [36] and, METEOR [37] metrics.

a) Exact Match (EM): The simplest metric is the EM which scores a prediction one if it matches its reference exactly and zero otherwise.

b) BLEU-4: The most widely used metric in the code summarisation task is the Bilingual Evaluation Understudy Score (BLEU) [13]. BLEU-4 produces a percentage number between 0 and 100, which defines the similarity between a candidate and a set of reference sentences. BLEU-4 calculates the cumulative 4-gram precision scores, the number of matching 4-grams divided by the total number of 4-grams in the candidate sentence [35]. The unigrams and bigrams account for the adequacy of the candidate while the longer three and 4-grams account for fluency. To prevent short sentences the result is multiplied by a brevity penalty as well. A smoothing function is applied to prevent sequences with no matching 4-grams to score zero [38]. While Shi et al. recommend BLEU-4 with smoothing method 4 [13], we opted to use the Moses [39] implementation of BLEU-4 which uses smoothing method 2 since this is also utilised by CodeSearchNet, CodeXGlue and CodeT5 [14, 20, 31].

c) ROUGE-L: ROUGE or Recall-Oriented Understudy for Gisting Evaluation, is a package which includes several metrics, the most popular among them is ROUGE-L [36]. ROUGE-L is more recall oriented than BLEU-4. ROUGE-L simply finds the longest common subsequence (LCS) between the reference and the candidate. Note that the words do not need to be consecutive but they have to be in order.

d) METEOR: METEOR or Metric for Evaluation for Translation with Explicit Ordering [37] uses word lists and stemming to also take synonyms into account and calculates the harmonic mean of the unigram precision and recall. Similar to ROUGE-L, METEOR is more recall-focused. METEOR has a higher correlation with human judgement than BLEU-4 [19] at the sentence level.

D. Data deduplication

To create a deduplicated version of the CAPYBARA dataset we make use of a fork[16] of the near-duplicate-code-detector [33]. We use this tool to compare all the datasets' functions and find clusters of near-duplicate functions. We randomly select one function per cluster and discard the rest from the dataset. We use the standard tool configuration as recommended by Allamanis. Of the removed duplicates, we observe that a relatively large number originates from common libraries, such as SQLite[17], that are packaged with binary programs. Thus a certain amount of duplication is also likely to occur "in the wild".

E. Configuration

We process and visualise the data with Pandas 1.4.3 and Ghidra 10.0.4[18]. FastText 1.0.3 with the largest lid.176.bin model is used to detect languages. We train the model using Transformers version 4.16.2 running on Torch 1.9.0+cu111 in the nvidia/cuda:11.4.0-base docker container image. We share a Docker image with all the libraries required to run BinT5 pre-installed on DockerHub[19].

A grid search of the optimal settings was infeasible from a time perspective, so we performed training mainly using the recommended settings from the CodeT5-base model [14]. We double the source length for the decompiled, stripped, and demi-stripped code to 512 tokens instead of the standard 256 tokens used for the source code to compensate for the fact that the average length of decompiled code is almost twice as long as the source code. We trained the model on a machine with an NVIDIA GeForce RTX3080 with 10GB of VRAM and an AMD Ryzen Threadripper 3990X 64-Core Processor with 192GB of RAM running Ubuntu 20.04.4 LTS. The GPU is running Nvidia driver version 510.60.02 with Cuda 11.6. The authors of CodeT5 used an NVIDIA A100 GPU with 40GB of VRAM for fine-tuning [14]. To compensate for the lack of memory, we reduced the batch size to 2, which was the maximum length that could still fit in the VRAM, we increase the 'gradient_accumulation_steps' to 24 to still achieve the effective standard batch size of 48.

	BLEU-4	EM	METEOR	ROUGE-L
C	60.83	52.19	65.33	66.51
DecomC	58.82	48.92	63.14	64.51
Stripped	11.26	1.85	14.50	17.25

TABLE II: Result of fine-tuning CodeT5-base on mined datasets

VI. Results

We present the results of our experiments to answer the research questions, results are grouped per research question. The metrics are calculated for each sample from the test set, and the average scores are presented.

A. RQ1: Model Effectiveness

The performance of the CodeT5-base model on each of the datasets is presented in table II.

We found that the decompiled code model generally produced good summaries, evidenced by the BLEU-4 score of 58.82, which is slightly lower than the baseline set by the source code. The stripped model mainly produced unusable summaries, as evidenced by the BLEU-4 score of 11. The high EM score could be an indication of a high duplication factor.

Initial experiments with GraphCodeBERT [40] and PolyglotGraphCodeBERT [16] base models fine-tuned on CAPYBARA show performance around 5 and 3 BLEU-4 lower, respectively. This is a relatively small difference, especially considering the model size. This shows that the performance of BinT5 does not heavily depend on the additional pre-training

[16]Near Duplicate Code Detector: https://github.com/SERG-Delft/near-duplicate-code-remover

[17]SQLite: https://www.sqlite.org/index.html

[18]It is not recommended to use Ghidra versions before 10.1 since these versions have not been patched against a Log4J RCE

[19]BinT5 Docker Image: https://hub.docker.com/r/aalkaswan/bint5/tags

	BLEU-4	EM	METEOR	ROUGE-L
DecomC	58.82	48.92	58.4	60.32
Demi	44.21	35.10	47.89	49.59
NoFunName	46.99	37.12	45.92	48.07

TABLE III: Result of fine-tuning CodeT5-base on synthetic data

Opt level	BLEU-4	EM	METEOR	ROUGE-L
-O0	72.88	34.18	73.19	74.84
-O1	50.30	59.84	55.36	54.84
-O2	62.31	46.23	64.50	66.05
-O3	54.68	54.99	58.25	59.28

TABLE IV: Average BLEU-4 score of decompiled code per optimisation level

on C and C# performed by Wang et al.. Furthermore, this result shows that it is improbable that significant dataset leakage has taken place.

We found a relatively large difference between the number of recovered decompiled and stripped decompiled functions. This can likely be attributed to the fact that Ghidra struggles a lot more with recovering stripped functions. Recall that the symbol table commonly contains information regarding the location and name of functions. When this table is dropped, the start- and endpoints of functions are hard to infer by automatic tools, especially since many functions get inlined, and JUMP instructions replace CALL instructions. Aside from difficulties in demarcating functions, it is also difficult to align the associated source code function with the decompiled function. With unstripped code, the function name remains, meaning the functions can be aligned using the name. We attempted to utilise an existing solution by Alves-Foss and Song called Jima [41] to find function boundaries. Jima is the current state-of-the-art tool for function boundary detection in stripped binaries. The tool is implemented as a plugin for Ghidra, but in our experiments, we find no statistical difference between the base performance of Ghidra and Jima on our dataset. The difficulties in extracting stripped functions, make training and applying a model to stripped binaries challenging.

B. RQ2: Input Properties

As can be observed in Table III, the summaries produced by the demi-stripped model were substantially worse than the decompiled model, but most were still very usable, evident by the BLEU-4 score above 44. Just removing the function name gave quite similar results to demi-stripping. We find that the loss of identifiers significantly lowers the performance of the model, but stripped code also suffers from decompilation faults, which seem to have a much larger impact on the model performance. Hence, the performance of BinT5 on demi-stripped code can be viewed as more representative of the actual model and not impacted by faults introduced by Ghidra.

Table IV shows the average score per optimisation level. We can observe that -O0 and -O2 perform better than -O1 and -O3. Recall that -O0 is completely unoptimised, and that the vast majority of our decompiled dataset is compiled with -O2, which would explain why those optimisation levels perform better.

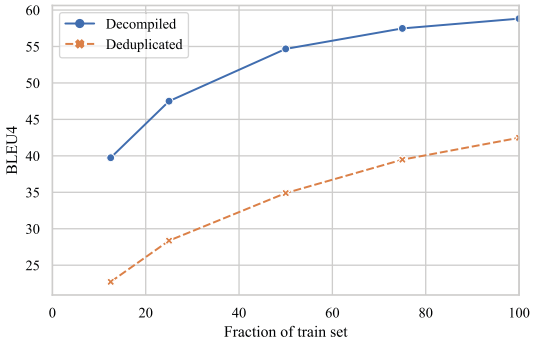

Fig. 7: BLEU-4 per trainset size for decompiled code and deduplicated decompiled code

C. RQ3: Dataset Properties

The performance of the base model on each of the deduplicated datasets is presented in table V:

	BLEU-4	EM	METEOR	ROUGE-L	ΔBLEU-4
C	45.86	32.87	46.06	47.53	14.97
DecomC	42.48	28.08	25.23	27.66	16.34
Demi	25.38	14.51	42.47	44.47	18.83
Stripped	7.19	0.00	4.75	5.50	4.07

TABLE V: Result of fine-tuning CodeT5-base on the deduplicated datasets and the difference with the baseline

We find that the influence of deduplication on our model's performance is relatively small on source code, at only 24%. Duplicates have a relatively large impact on the decompiled (28%) and demi-stripped (43%) code. Deduplication also greatly decreases the EM rate across the board. Duplicates have a relatively large impact on performance, but even with the duplicates removed the model still produces many high-quality summaries. The experiments on deduplication show that the model seems to have a deeper understanding of the data and is not simply reproducing previously seen samples.

As can be seen in Figure 7, the dataset size does not have much of an impact, the model can be trained with half or a quarter of the training samples without suffering a considerable hit to performance. This could be attributed to the high duplication factor of our dataset. It could also be because the model was already pre-trained well by Wang et al. and requires very little data for fine-tuning. This is a testament to the relative ease with which these models could be extended to decompiled code.

We also performed experiments where we did not apply the filtering rules provided by CodeXGlue and where we always mined the first sentence of any type of documentation. While

we were able to collect around 480K decompiled samples, the model performed substantially worse, only scoring 36.97 and 33.26 BLEU-4 on C and decompiled code, respectively. These results show that the dataset quality also heavily impacts the model performance.

VII. Discussion

In the previous section, we found that BinT5 shows considerable performance for decompiled code and demi-stripped code on both regular as well as deduplicated data. While this is a promising result, we conduct a small investigation of the decompiled samples. We will put our observations on identifiers into the context of the extreme summarisation task. Based on this we discuss the implications of our work. Finally, we will close this section by discussing the threats to validity.

A. Exploration of Results

To explore the results of BinT5 we pick 25 high and 25 low-scoring samples from the test set of the deduplicated decompiled dataset. High samples have a BLEU-4 score higher than 75 while low-scoring samples have a score lower than 25.

a) High Samples: With the high-performing samples BinT5 tends to produce summaries which are very close to the references. For instance, BinT5 produced Print description of a datatype in XML against the baseline Dump description of a datatype in XML. Of the 25 high-scoring samples we found that all have counterparts with a similar function summary in the training set. These functions also tend to have similar names, but their decompiled function body was significantly different, which is likely why deduplication didn't remove these functions.

b) Low Samples: From the low-performing samples we observe that many summaries produced by BinT5 are semantically very similar to the reference. For instance, the function vl_set_simd_enabled[20], has the reference Toggle usage of SIMD instructions while BinT5 produced Enable or Disable the Simd Channel. This sample scores a BLEU-4 score of 0.0, because of the limitations around the BLEU-4 metric, while for a human evaluator the output is still very usable. Similarly, for some samples, BinT5 produces shorter summaries containing shorthands. The reference Check if the given nickname is blocked for "normal client" use against Check whether nick is blocked, also scores poorly. Of the 25 low-scoring samples we observe that around 11 are semantically similar to the reference and likely very useful for understanding the function.

B. Identifiers and Extreme Summarisation

We find a relatively small difference in performance between source code and decompiled code. This indicates that in-function comments and variable names are relatively unimportant for the model performance. Although Ahmed and Devanbu observed that identifiers might be more important than syntax in the code-summarisation task [16], we can further conclude that the function name is explicitly essential for model performance. Removing just the function name from the decompiled samples, as opposed to removing all identifiers in demi-stripping, results in slightly higher performance than demi-stripped code, which indicates a very high dependence on the name of the function in the code summarisation task, which is a logical finding in the context of the extreme code summarisation task.

The extreme code summarisation task, as proposed by Allamanis et al. aims to reproduce the function name given a function body [16, 42]. It is framed as a summarisation problem where the output is around 3 tokens in length, instead of the 10+ tokens that regular code summarisation targets. We found similar results when performing this task with our dataset, namely, high performance on regular decompiled code (with function names removed) and low performance on stripped code.

A manual assessment of the stripped data shows that many of the aligned functions were not decompiled properly. We find that many functions are cut-off after a few instructions because the decompiler did not recover the full control flow. Other functions are missing side effects, like changes to global variables.

C. Implications

We propose a novel solution to aid reverse engineers in their work. If the application of NLP to binaries gets significantly better, and the limitations around stripping and other obfuscation techniques get resolved, it would have severe implications for the cybersecurity domain. On one hand, it could help malware analysts understand novel malware and its weaknesses quickly. Software can be analysed to find possible vulnerabilities and malicious payloads. Source code can be reconstructed for old binaries for which the source code is lost. But on the other hand, attackers can leverage these same methods to find and exploit vulnerabilities and lift intellectual property from binaries.

CAPYBARA itself could be used to create and assess neural decompilation, to perform a deeper investigation into the extreme summarisation task, or to simply train a code summarisation model on C code. CAPYBARA consists of a large corpus of C and decompiled C code, which could be used to pre-train language models, such that these models could support decompiled code out-of-the-box.

While our work focused on decompiled code, our observations show some limits of transformer-based models and their applicability to different data. Our dataset can help and inspire other researchers to improve upon our work. We hope other researchers use this dataset to train and evaluate their own models. Furthermore, the process outlined in Chapter III could help others construct standardised datasets for other tasks and languages.

D. Threats to Validity

Internal Validity questions if other factors could have affected the outcome. The training and evaluation data contains

[20]Colmap/Colmap:vl_set_simd_enabled: https://github.com/colmap/colmap/blob/87b3aa325bd8e5fb913788e29e9ac1e085e28b67/lib/VLFeat/generic.c#L1070

a significant amount of noise, either in the form of badly decompiled functions or incorrect documentation. We carefully collect and process the data, but we are unable to know to which extent the documentation matches the original code. While machine learning models (and specifically NLP models) should be able to handle noisy data, this might introduce some bias into the models. CodeT5 was also pre-trained on a C and C# dataset, this dataset is unpublished and we were unable to reach the authors. Some data leakage might have taken place, but as explained in Section VI it is unlikely that it had much of an impact. To prevent this threat from arising in any future studies, we make CAPYBARA publicly available.

External Validity refers to the generalisability of our results. This work only focuses on stripping and compiler optimisations as a means of resisting binary analysis, other techniques like control flow obfuscation and packing are also used to prevent reverse engineering. Other works focus on unpacking and deobfuscation, so we consider our work orthogonal to theirs. The data gathered for CAPYBARA were exclusively from open-source projects. Decompiling closed-source projects is explicitly forbidden by some EULAs and the lack of source code documentation makes it difficult to evaluate using reference summaries. However, reverse engineering open-source software is not very useful in practice, since the source code is readily available. Closed-source software might have different data distribution and will present other challenges like obfuscation. Finally, only functions that decompile (Ghidra produces any output) and that are documented, are represented in CAPYBARA. This is most apparent in the stripped dataset, where we can only recover a small fraction of the total number of functions. A deeper investigation into new decompilation techniques for stripped code, specifically into the aspect of function boundary detection is left as future work.

Construct Validity relates to the adequacy of the theoretical constructs and the use of appropriate evaluation metrics. The leading metric in our evaluations does not capture semantic meaning. While BLEU-4 is the most popular metric for this task, its reliability has been called into question [43, 44]. We, therefore, included other metrics, which do take semantics into account, in our evaluation. Finally, our entire approach hinges on the assumption that function summaries, as they are used for source code, are useful for binary analysis. Whether or not this is actually the case, should be further investigated with a qualitative user study, this is left as future work.

VIII. Related Work

Binary reverse engineering and the use of NLP for software engineering are vast and active fields, so we select and discuss the closest state-of-the-art works in the field. We categorise the studies into identifier recovery and binary translation. Finally, we will discuss the open challenges and the relation of our own work to these challenges.

a) Recovering Identifiers from Stripped Binaries: **Debin** [5] aims to recover debug information from stripped binaries. The authors use a tree-based classification and a probabilistic graph-based model. All the variable names and types are jointly recovered using a maximum a posteriori probability inference. **VarBERT** [45] uses a Transformer-based NLP model for the task of variable name recovery. The authors pre-trained a BERT model which is then fine-tuned to predict the names and types from *unstripped* binaries.

FUNCRE [7] uses a pre-trained and fine-tuned ROBERTA [29] model to predict usages of inlined library functions. Recall that compilers with optimisations enabled can inline functions in the binary (Chapter II). The authors use indelible markers, which do not get destroyed by the compiler, to mark usages of library functions and to construct a dataset and train a model.

b) Binary Translation: **Neutron** [10] frames decompilation as a neural machine translation problem and utilises an Attention-LSTM-based neural translation network to translate disassembled binaries back to C source code. The binaries are not stripped and do not have any optimisations enabled. The translations created by Neutron can contain syntax errors, so the authors apply regular expressions to create a tailor-made syntax checker. Neutron achieves high accuracy on the translation task, but only on unstripped and non-optimised code.

c) Our Novelty: Several aspects have not been properly addressed and investigated. The application of code summarisation methods to decompiled code has not been addressed by any work at all. Furthermore, some works on binary code fail to take compiler optimisations into account [10]. We, therefore, investigate the application of code summarisation methods to decompiled code and we enable compiler optimisations.

IX. Conclusion

In this paper, we proposed a new automatic binary code summarisation task. With this new task, we also introduce CAPYBARA, a novel dataset to train and evaluate models on this task, with both mined as well as synthetic data. Paired with this dataset, we train BinT5, a Transformer-based code summarisation model to show the effectiveness of CAPYBARA. We used BinT5 to further explore the datasets, outlining the inherent difficulties in the data.

We found that while BinT5 shows considerable performance on regular decompiled code, but its performance is being hampered by the decompiler on stripped code, evidenced by BinT5s strong performance on demi-stripped code. Furthermore, we found that while duplicates have a large impact on the model, their presence is not paramount to the model's performance. Finally, we observe that BinT5 could be trained with just a fraction of the samples in CAPYBARA.

Our work has shown that a well-known and well-studied task from the source code domain [13], namely source code summarisation, can be applied to binary code. This is only one of the many different applications of NLP for code. Our paper constitutes the first step in the application of source code NLP methods to such tasks on binary code.

REFERENCES

[1] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' process and mental models," in *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI EA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–6. [Online]. Available: https://doi.org/10.1145/3290607.3313040

[2] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428293

[3] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, May 2016. [Online]. Available: https://doi.org/10.1145/2896499

[4] L. Chen, Z. He, and B. Mao, "Cati: Context-assisted type inference from stripped binaries," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 88–98.

[5] 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*, 2018, pp. 1667–1680.

[6] 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)*. IEEE, 2019, pp. 628–639.

[7] T. Ahmed, P. Devanbu, and A. A. Sawant, "Learning to find usage of library functions in optimized binaries," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[8] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," 2022.

[9] D. Lehmann and M. Pradel, "Finding the dwarf: Recovering precise types from webassembly binaries," 2022.

[10] R. Liang, Y. Cao, P. Hu, and K. Chen, "Neutron: an attention-based neural decompiler," *Cybersecurity*, vol. 4, p. 5, 03 2021.

[11] C. Zhang, J. Wang, Q. Zhou, T. Xu, K. Tang, H. Gui, and F. Liu, "A survey of automatic source code summarization," *Symmetry*, vol. 14, no. 3, p. 471, 2022.

[12] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," ser. ASE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 43–52. [Online]. Available: https://doi.org/10.1145/1858996.1859006

[13] Shi, E. Wang, Y. Du, L. Chen, J. Han, S. Zhang, H. Zhang, D. Sun, and H. Sun, "On the evaluation of neural code summarization." ICSE, 2022.

[14] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[15] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[16] T. Ahmed and P. Devanbu, "Multilingual training for software engineering," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 1443–1455.

[17] C. Casalnuovo, E. T. Barr, S. K. Dash, P. Devanbu, and E. Morgan, "A theory of dual channel constraints," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, pp. 25–28.

[18] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, p. 122–131, apr 2016. [Online]. Available: https://doi.org/10.1145/2902362

[19] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 3931–3937. [Online]. Available: https://aclanthology.org/N19-1394

[20] 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.

[21] K. Hoste and L. Eeckhout, "Cole: compiler optimization level exploration," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, 2008, pp. 165–174.

[22] M. T. Jones, "Optimization in gcc," *Linux journal*, vol. 2005, no. 131, p. 11, 2005.

[23] S. Blazy and S. Riaud, "Measuring the robustness of source program obfuscation: Studying the impact of compiler optimizations on the obfuscation of c programs," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 123–126. [Online]. Available: https://doi.org/10.1145/2557547.2557577

[24] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, "Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 659–676.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit,

L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 6000–6010.

[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Representations by Back-Propagating Errors*. Cambridge, MA, USA: MIT Press, 1988, p. 696–699.

[27] J. Schmidhuber, S. Hochreiter *et al.*, "Long short-term memory," *Neural Comput*, vol. 9, no. 8, pp. 1735–1780, 1997.

[28] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[29] L. Zhuang, L. Wayne, S. Ya, and Z. Jun, "A robustly optimized BERT pre-training approach with post-training," in *Proceedings of the 20th Chinese National Conference on Computational Linguistics*. Huhhot, China: Chinese Information Processing Society of China, Aug. 2021, pp. 1218–1227. [Online]. Available: https://aclanthology.org/2021.ccl-1.108

[30] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of NAACL-HLT*, 2019, pp. 4171–4186.

[31] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[32] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016.

[33] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Athens Greece: ACM, Oct. 2019, pp. 143–153. [Online]. Available: https://dl.acm.org/doi/10.1145/3359591.3359735

[34] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[35] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[36] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013

[37] A. Lavie and M. J. Denkowski, "The meteor metric for automatic evaluation of machine translation," *Machine translation*, vol. 23, no. 2, pp. 105–115, 2009.

[38] B. Chen and C. Cherry, "A systematic comparison of smoothing techniques for sentence-level BLEU," in *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Baltimore, Maryland, USA: Association for Computational Linguistics, Jun. 2014, pp. 362–367. [Online]. Available: https://aclanthology.org/W14-3346

[39] H. Hoang and P. Koehn, "Design of the moses decoder for statistical machine translation," in *Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, 2008, pp. 58–65.

[40] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *ICLR*, 2021.

[41] J. Alves-Foss and J. Song, "Function boundary detection in stripped binaries," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 84–96. [Online]. Available: https://doi.org/10.1145/3359789.3359825

[42] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2091–2100. [Online]. Available: https://proceedings.mlr.press/v48/allamanis16.html

[43] D. Roy, S. Fakhoury, and V. Arnaoudova, *Reassessing Automatic Evaluation Metrics for Code Summarization Tasks*. New York, NY, USA: Association for Computing Machinery, 2021, p. 1105–1116. [Online]. Available: https://doi.org/10.1145/3468264.3468588

[44] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, "Semantic similarity metrics for evaluating source code summarization," *arXiv e-prints*, pp. arXiv–2204, 2022.

[45] P. Banerjee, K. K. Pal, F. Wang, and C. Baral, "Variable name recovery in decompiled binary code using constrained masked language modeling," *arXiv preprint arXiv:2103.12801*, 2021.