STraceBert: Source code retrieval using semantic application traces

By

CLAUDIO SPIESS

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Premkumar T. Devanbu

Vladimir Filkov

Cindy Rubio-González

Committee in Charge

2023

Table of contents

List of Figures

List of Tables

We do these things not

because they are easy,

but because we thought

they were going to be easy

PROGRAMMERS' CREDO

Abstract

Software reverse engineering is an essential task in software engineering and security, but it can be a challenging process, especially when artifacts are engineered (adversarially) to resist reverse engineering.. To address this challenge, we present STraceBERT, a novel approach that utilizes a Java dynamic analysis tool to record calls to core Java libraries, and pretrain a BERT-style model on the recorded application traces for effective method source code retrieval from a candidate set. Our experiments demonstrate the effectiveness of STraceBERT in retrieving the source code compared to existing approaches. Our proposed approach offers a promising solution to the problem of code retrieval in software reverse engineering and opens up new avenues for further research in this area.

Acknowledgments

First and foremost, I would like to acknowledge and thank my advisor, Professor Prem Devanbu, for his mentorship and support throughout this program. His guidance has broadened my research skills significantly.

I would also like to thank Professor Vladimir Filkov and Professor Cindy Rubio-Gonzalez for their insightful and helpful comments. I would also like to thank Professor Vincent Hellendoorn, and Dr. Anand Ashok Sawant for their indispensible help in making this project a reality. I also would like to thank my lab mates, Dr. Kevin Jesse, Dr. Toufique Ahmed, and Dr-to-be David Gros, for their insights, help, and support.

Finally, I would like to thank my parents, partner, and the Davis community for supporting me throughout my studies.

CHAPTER 1

Introduction

Software reverse engineering is a critical task in the field of software engineering and security. A common approach is to re-engineer the source code from the compiled binary code to understand the functionality and vulnerabilities of the software. The task becomes more challenging in an adversarial setting, where the binary may be intentionally constructed to hinder analysis. A common example is ransomware, which can be expected to be obfuscated and invoke antianalysis techniques. A reverse engineer wants to understand how the malware works, so they can discover weaknesses, capabilities, and vulnerabilities used. Yet, understanding behavior from millions of assembler instructions is untenable, and practitioners have developed decompiler tools such as Ghidra[1] for native binaries, or Procyon[2], CFR[3], and Fernflower[4] for Java bytecode. However, these tools are sensitive to obfuscation techniques, and the usefulness of their output is limited in many such cases.

We hypothesize that while any binary can obfuscate their control flow, variable names, etc, they cannot hide their system calls without a rootkit. These system calls can be recorded on Unix systems with the `strace`[5] command. We hypothesize that a complex enough sequence of system calls can be used to retrieve source similar to that of the system which initiated tem, using an embedding model and vector database to retrieve the most similar known sequence of calls with source code. Due to the challenge of aligning method boundaries and source code with system call sequences, we adapt this problem to the Java ecosystem. We utilize a Java dynamic analysis tool Jackal, that instruments bytecode on the fly and records events such as method calls, exits, and field modifications.

[1]Ghidra: `https://github.com/NationalSecurityAgency/ghidra`
[2]Procyon: `https://github.com/ststeiger/procyon`
[3]CFR: `https://www.benf.org/other/cfr/`
[4]Fernflower: `https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine`
[5]strace: `https://strace.io/`

We draw an analogy between system calls and calls into Java core libraries i.e. `java.*`. Using Jackal, we trace the test suites of a set of open source Java projects, record their true source code, train a BERT-style model on the sequences of Java calls, embed all sequences, and then evaluate the similarity of proposed source code candidates for a holdout set. Through our experiments, we demonstrate the effectiveness of our proposed model in retrieving the source code compared to existing baselines.

CHAPTER 2

Background and Motivation

2.0.1. Reverse Engineering & Motivation. The task of reverse engineering is an important part of software engineering and security. A reverse engineer in a software context attempts to analyze and understand a piece of software. Usually, this means the original source code is not available to the reverse engineer. Therefore, the reverse engineer is motivated to produce source code for the given artifact they are analyzing. The process of returning from low level bytecode or assembly to a high level language is called decompilation. This is usually done with an analytical i.e. procedural decompiler that is tailor built to the specific instruction or bytecode set and target output language.

Reverse engineering is important for various reasons, the most critical of which can be considered its application in security. When security specialists encounter a piece of malware, whether through automated submission via antivirus software, or specialist extraction following a high-profile attack such as Stuxnet or the Colonial Pipeline ransom, their goal is to understand how the malicious software (“malware”) accomplished its attack and what its capabilities are. With this knowledge, security companies and operating system developers can fix security vulnerabilities and develop counter measures. With this goal in mind, reverse engineers usually use a set of common tools depending on the type of software artifact used during an attack. It is likely that most malware is written in C/C++ and distributed in binary format, potentially with measures that make reverse engineering more difficult. Such measures often include obfuscation, which is the process of making the binary/byte code instructions, or the original code, more complicated. This is usually achieved by removing or renaming variable names to nonsensical values, adding dummy instructions or code, complicating control flow, and more. Given that most artifacts are binaries, reverse engineers use tools such as HexRays disassembler or the NSA’s Ghidra decompiler. Disassemblers focus on displaying the binary instructions in a more visual and navigable way, while decompilers attempt to produce source code based off the binary instructions.

However, binary instructions are extremely verbose and low level, making it a slow and painstaking process that requires a highly specialized and experienced engineer to understand the workings of a complicated, adversarial (e.g. obfuscated) piece of software. On the other hand, decompilers use heuristics to produce code that was likely to produce the instructions, but without a symbol table (which would likely be removed) cannot produce meaningful variable names. Decompilers also produce ”unnatural” code, i.e. code that programmers would find difficult to follow. Often, the code produced is not actually compilable or usable without extensive changes by an engineer. Therefore, it serves to assist the analysis of the artifact only.

Given these limitations, this work explores a novel approach to reverse engineering. Given that all software, even malware, must make system calls to achieve basic functionality such as connecting to the internet or writing to a file. These calls can be monitored with simple tools such as `strace`. By recording system calls for known pieces of code, we can produce a database of code, and then search for the source code of a set of system calls for which the code is desired. The possible code candidates can then be presented to a reverse engineer as they are working on analyzing the artifact. This thesis investigates the feasability of this approach.

2.0.2. Tracing: Jackal & Bartok. This work uses Jackal, a dynamic analysis tool developed in lab by previous members. The tool consists of a modified OpenJDK JVM and a Java *agent*. The agent calls the added functions in the custom JVM to achieve low level operations such as dumping the recorded data to disk. The agent is a piece of Java software that implements the JavaAgent interface. It is given the *bytecode* of each class encountered during runtime, which is then modified to include instrumentation: code that records the values of fields, method calls, etc. The bytecode is returned to the JVM which then executes it. Once the JVM is closed e.g. when the program finishes, it dumps all the gathered data to disk. These dumps are then processed to extract useful data for our task. This is further described in chapter 4 starting on page 7.

CHAPTER 3

Related Work

We searched for similar works applying dynamic analysis for obtaining or augmenting source code, decompilation with ML augmentation, and clone detection. There is a line of work that uses ML methods to improve decompiled code [21, 25] or improve source code retrieval using static analysis products [1]. Another line of work looks at binary clone detection & binary diffs [10, 12, 18]. Finally, a line of work looks at embedding binary code [8]. We found only some work [30] which uses dynamic analysis techniques in a similar fashion as this work.

3.0.1. Augmenting Decompiled Code & Static Analysis. This line of work uses the output of a decompiler as a starting point for producing helpful information to a reverse engineer. They do not include any dynamic analysis products in their methodologies. Al-Kaswan et al. [21] extended large pre-trained models of source code to summarize de-compiled binary functions in C. The goal of this work was to produce natural language descriptions from decompiled code that closely matched the original documentation of the function in the dataset. They achieved a BLEU-4 score of 58.82 and 44.21 for decompiled and decompiled obfuscated code respectively. Lacomis et al. [25] focused on variable name recovery from decompiled source code in C. This approach predicted identifier names identical to the ground truth 74.3% of the time. Ahmed et al. [1] found an improvement in the CodeSearchNet [19] code summarization task by few-shot prompting large language models with static analysis products such as repository metadata, data flow graphs, and tagged identifiers, suggesting that static analysis products may help guide large models.

3.0.2. Binary Clone Detection & Diffs. The binary clone detection and binary diff tasks aim to quantify the similarity of two given binaries and produce matchings. Such approaches relate to this thesis as we also aim to find similar binaries (with associated source code) for a given sample. Farhadi et al. [12] presented non-probabilistic algorithms for matching binaries *i.e.* detecting binary clones. Hu et al. [18] introduced a semantic approach for binary clone detection that utilizes dynamic analysis products as part of their algorithm. Similar to this thesis, they executed a binary function with its test cases and emulated their execution. Duan et al. [10] introduced an unsupervised code representation learning approach, using code semantics and control flow to generate block level embeddings, and a greedy matching algorithm. Their approach outperformed previous, non-neural methods. Koo et al. [24] improved upon Duan et al. [10], reporting an improvement of 49.84% on average, using a BERT-style [7] model on assembler instructions.

3.0.3. Embedding Binary Code. Ding et al. [8] proposed jointly learning lexical semantic relationships and assembly function vector representations for the clone detection task. They report that their method outperforms prior work and is robust to obfuscation and compiler optimizations.

3.0.4. Dynamic Analysis. Pei et al. [30] proposes a transfer-learning based approach to learning execution semantics from function traces, teaching a model the execution semantics of instruction sequences in a unsupervised pre-training manner. They then fine-tuned the model to match semantically similar functions. Their results are encouraging as they report an up to 14.3% improvement in an obfuscated setting over prior work.

CHAPTER 4

Methodology

4.1. Dataset

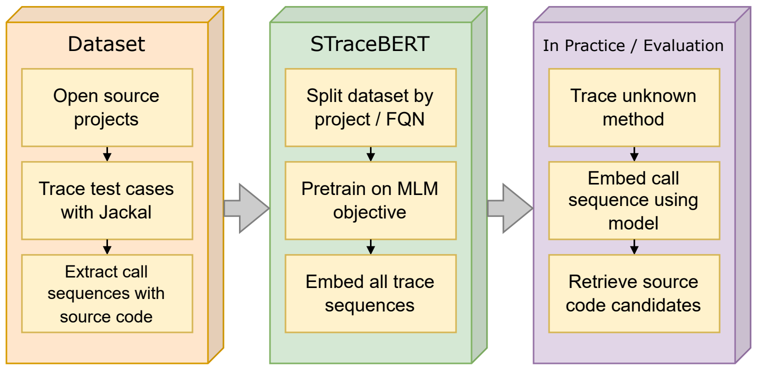

Dataset
Open source projects
Trace test cases with Jackal
Extract call sequences with source code
STraceBERT
Split dataset by project / FQN
Pretrain on MLM objective
Embed all trace sequences
In Practice / Evaluation
Trace unknown method
Embed call sequence using model
Retrieve source code candidates

FIGURE 4.1. Overview of the STraceBERT approach.

As our approach is novel, we needed to construct the Java Trace Dataset (JTD) [6]. The JTD consists of Java application traces obtained from a variety of open-source projects. To produce this dataset, we gathered a list of top Java projects using the Maven build system on GitHub by number of stars. See table A.1 on page 35 in the appendix for a full list. The motivation for Maven-only projects was driven by the existing JACKAL infrastructure for Maven projects, and that we avoid Android specific projects. Additionally, Maven's Project Object Model (POM) schema allows us to programmatically add additional plugins, specifically the Maven ProGuard plugin, and remove plugins that conflict with instrumentation.

To collect the projects, we performed the following steps:

(1) Called the GitHub API to search for repositories that match the following three queries, sorted in descending order by number of stars:
   (a) `org:apache OR org:eclipse path:**/pom.xml`
   (b) `topic:java-tutorials`
   (c) `java language:java`

(2) For the queries (b) and (c), we further processed the first page of results returned by piping the returned repositories into another search query `repo:$REPOSITORY path:**/pom.xml`. Combined with the results of query (a), this produced a list of popular projects that use the Maven build system.

(3) Used `git clone --depth 1` to clone the projects without their entire commit history.

4.1.1. Tracing. We trace the test suites of the individual projects using the Jackal tracing system. Jackal consists of a modified OpenJDK 13 JVM and a Java agent developed by previous members of the DECAL lab[1]. To run many test suites in a parallelized manner, we developed Bartok, a custom Docker orchestrator written in Python that analyzes a project folder for Maven test suites, producing a YAML[2] file of test suites and their build path with respect to the project root. The YAML file acts as a configuration file for the next step, which is executing the test suites in their own respective Docker container instance. Once the YAML file of test suites is finalized, Bartok executes a three step process:

(1) Project root is built using Maven in a `build` container, which builds all submodules. This ensures the project builds and test cases run without instrumentation yet. Critically, all class files *i.e.* Java bytecode is built and stored in the project build directory.

(2) If the build succeeds, each test suite specified in the YAML is executed in its own `test` container running the Jackal JVM and Java agent attached to the Maven test runner. The Java agent instruments *i.e.* injects Java bytecode to record operations into the classes encountered at runtime. Prior to instrumentation, each class is written to the file system in a special temporary directory, following the class' fully qualified name package hierarchy. As the test suite runs, all method calls and exits are recorded with this mechanism. Once the test suite terminates, the custom JVM dumps `dump-*.zip` files to disk, that contain the Jackal Domain Specific Language (DSL) representation of the execution trace, an efficient binary representation of the trace events.

(3) If a dump is produced successfully, another Docker container is started that runs the first post processing script. This script loads the zip file with the DSL and parses it. The goal

[1]The authors have not publicly released this tool as of writing this manuscript
[2]YAML: YAML Ain't Markup Language

of this operation is to reorder the trace into a sequential series of events and output JSON for further processing.

4.1.2. Processing. Once the JSON representation of a test suite's trace is available, we begin by decompiling the pre-instrumentation class files dumped, as described in § 4.1.1. Rather than using the original source code, as the gold standard to be retrieved, we use the decompiled code from the Java byte code, because we initially attempted to use a heuristic search to locate the Java source file containing each instrumented method, but found this challenging due to high diversity in project directory organization, and largely complicated by the tracing of **dependencies** as well. As this primarily serves as a dataset construction task, we deem decompiled code of unobfuscated classes as similar enough to the ground truth source code to be usable for our task. We perform a qualitative analysis of source retrieved via our initial heuristic approach and our decompiled approach, and note that apart from comments and anonymous members, the source code is largely identical as identifiers are preserved.

To collect method source code, we construct tooling using TREE-SITTER[3]. We parse all decompiled source code collected for a project into Abstract Syntax Trees (ASTs), organizing them according to their package hierarchy. When a method entry event is encountered during the trace parsing process (described in further detail below), we are able to use the method's fully qualified name and signature to search the ASTs for a match. While this may seem a simple task at first, it is important to note that Java supports both overloading & polymorphism. This presents a challenge as multiple methods of the same name, in the same class, may have argument and return types specified in their source code, that are *never* encountered verbatim during execution. A concrete example of this issue, is that *any* non-primitive type in Java is of type *Object*. Therefore, if the type encountered in the trace is *Object*, any method with a signature *i.e.* arguments of non-primitive type is a valid candidate.

Subsequently, if a method is matched by name, but a verbatim exact match cannot be found by argument(s), we apply a best-guess heuristic, ensuring that the *number* of arguments matches. Finally, we simply record the source code associated with the function node with the traced method.

[3]Tree-sitter

To process a trace, a Python script reads the JSON and transforms it to a text representation suitable for training a large language model. The JSON trace object contains a series of events, which is either a method entry, method exit, method call, or a reference to another trace in the same JSON file, representing a different traced method. Method calls represent calls to methods that are not traced, primarily methods built in to the JVM or Java core libraries. These calls can be considered "java calls" as they usually call into the `java.*` namespace. The call graph is navigated per trace depending on the maximum depth parameter, writing an event string for each event to the trace's string representation. The maximum depth parameter can be 0, meaning a one-to-one mapping is produced of traced method to event sequences. If 1 is specified, only the method and its child is traversed. If -1 is specified, the call graph is navigated from root to leaf, up until the maximum depth parameter. Another parameter, the maximum event parameter, specifies how many events to process. This is necessary as the context window of the downstream language model is limited, so there is no purpose in printing more events than could realistically fit into a model. In our experiments, we do not limit the maximum depth, and limit the events *viz.,* calls to 512 at each level.

Method calls are recorded only if they are to the `java.*` namespace, to maintain our analogy to system calls *e.g.* a `java.net.Socket` call is analogous to a Unix `socket(2)` system call. If `calls_with_boundaries` is enabled, calls to other methods not in the `java.*` namespace are denoted by a `[CALL]` token. Method exits are denoted by `[EXIT]` token. The recursive depth reached is recorded per trace as the maximum depth feature.

A raw trace fundamentally begins with a test case. As we are not interested in tests themselves, but rather the behavior of methods being used, we define a trace of a *method* as beginning when said method is entered. As we process the raw trace JSON, any method without a caller must be a test case, and are eliminated accordingly. Then, any method called from the test case is considered a *root* method. We hypothesize that root methods do not produce many Java calls *e.g.* a call to `savefig` would call figure rendering methods and ultimately a method or utility to actually perform the IO operations using `java.io.*` classes. Therefore, we implement a *recursive* traversal. In this approach, each trace of a root method's calls *i.e.* children to other (non-`java.*`) methods are traversed depth first to produce the sequence of Java calls. Consider a root method that makes an in-project method call (a), invokes some Java core library (b), and then makes another in-project method call (c): in this case, the Java call sequence produced would be: all Java calls by (a) and its children, (b), and all Java calls by (c) and its children.

However, it is important to note that we consider any method that originated from a test case at some point up the call graph. Consider a test case, that calls one method, that calls one method, and so on until the 10th method in the chain makes a call to `java.*` and returns. In this case, we have ten separate traced methods with unique source code, each associated with the Java calls of its children. In this example, since only one Java call was made at the leaf, all ten traces will have the *same Java calls*. This approach was chosen as it would not be feasible to determine which of the ten methods actually made a system call if one was simply recording raw system calls using `strace`. To understand how call boundaries affect discernibility *i.e.* retrieval, we created two sets of Java call sequences for each method, one with call boundaries (as discussed above), and one without.

We removed all duplicate call sequences to ensure there are no exact duplicate call sequences between traces. We note that for each method, an average of 12 traces were recorded. For such cases, the call sequence and thus call sequence with boundaries are unique, but the source code is the same. We then picked four projects Eugenp Tutorials, Eclipse AspectJ, Eclipse LemMinX, and Apache Struts for which the traces act as the candidates set. We split the data into three sets: candidates, for which traces of a particular method by fully-qualified name occur *at least once* in candidate project traces, and queries, which consists of traces for which the method was encountered *only* in non-candidate projects. We create our With Libraries set by sampling 10,000 random traces from the candidate set without replacement, and create Without Libraries by sampling 10,000 random traces from the queries set. This leaves us with a candidate set and two query sets for evaluation: one where we assume the binary under examination uses common libraries such as Apache Commons, and one that doesn't. This provides a reasonable upper and lower bound on expected performance.

For pre-training the STraceBERT models, we used all traces in the candidate set, with or without associated source code. An example of a call is

`-> java.lang.String.trim(): java.lang.String`. We represent each call signature by one unique token in our vocabulary, to maximize the number of calls we can ingest in our model, and to reduce vocabulary size. For sequences with call boundaries, method invocations and exits outside of the `java.*` namespace are demarcated by the special tokens `[CALL]` and `[EXIT]` respectively. We do not include the name of the called method to avoid data leakage, as the method names would be obfuscated in practice.

```
-> java.lang.String.trim(): java.lang.String -> java.util.Iterator.next():
java.lang.Object -> java.util.Collections.emptySet(): java.util.Set ->
java.util.Map.getOrDefault(java.lang.Object,java.lang.Object): java.lang.Object ->
java.util.Iterator.hasNext(): boolean ...
```

FIGURE 4.2. Tokenization of a sequence

```
-> java.lang.String.trim(): java.lang.String [CALL] [CALL] ->
java.util.Iterator.next(): java.lang.Object [EXIT] [EXIT] ->
java.util.Collections.emptySet(): java.util.Set ->
java.util.Map.getOrDefault(java.lang.Object,java.lang.Object): java.lang.Object ->
java.util.Iterator.hasNext(): boolean ...
```

FIGURE 4.3. Tokenization of a sequence with call boundaries, highlighted in red

4.2. Analysis of Dataset

No good machine learning project starts without a solid understanding of the dataset one is working with. This section, we will gain insight into the Java Trace Dataset.

4.2.1. Dataset Columns.

Java Calls	Java Calls with Boundaries	Source Code	Count
True	False	False	293,213
True	False	True	4,872
True	True	False	26,944
True	True	True	231,652
			556,681

FIGURE 4.4. Analysis of missing data

Column	Description
Project	Project from which trace was recorded
Test Suite	Fully Qualified Name of Test Suite
Class	Identifier of traced class
Method Name	Identifier of traced method
Just Class Name	Identifier of traced class with anonymous classes stripped
Just Method Name	Identifier of traced method with anonymous methods stripped
Anonymous Classes	Anonymous Class Identifiers
Anonymous Methods	Anonymous Method Identifiers
Source Code	Java source code of method
Java Calls	Sequence of Java calls encountered during trace
Calls With Boundaries	Sequence of Java calls encountered during trace with method calls and exits denoted by special tokens
Java Call Count	Number of Java calls encountered
Maximum Depth	Maximum recursive depth of call graph

TABLE 4.1. Reference table for columns in dataset.

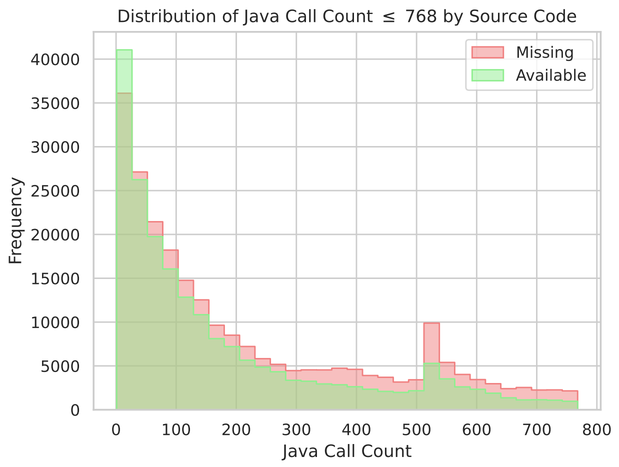


FIGURE 4.5. Distribution of number of Java calls per trace in dataset, split by source code availability.

4.2.2. Missing Data. As with many datasets, there are missing values as a result of errors during data collection or processing. To ensure there is no obvious bias in the data as a result of these missing values, we analyze the number of Java calls and the presence of associated source code. In Figure 4.5, we see that the missing source code does not corelate strongly with the length of the trace *i.e.* number of Java calls. However, shorter traces do tend to have source code associated more frequently than longer traces.

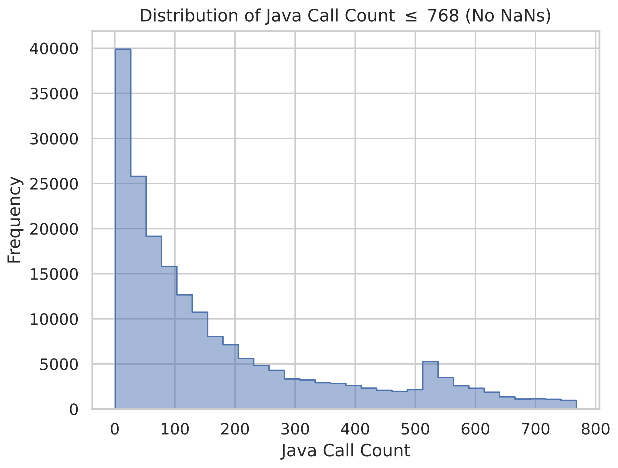


FIGURE 4.6. Distribution of number of Java calls per trace in dataset. Plot covers only traces for which Java Calls, Java Calls with Boundaries, and Source Code are available.

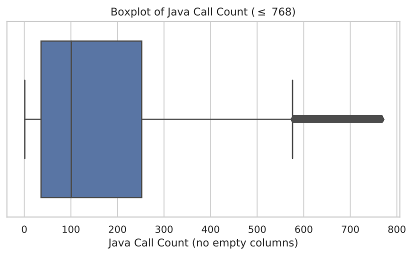
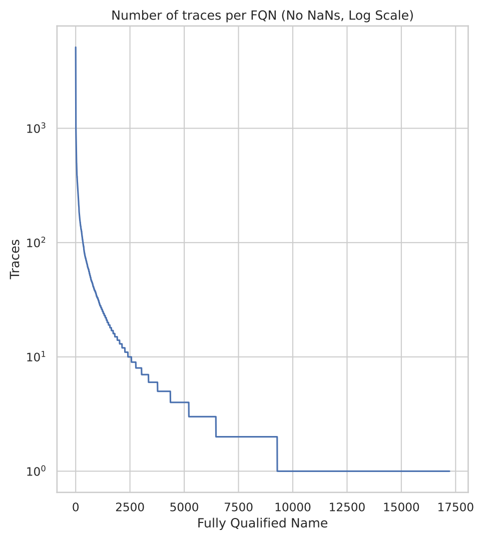


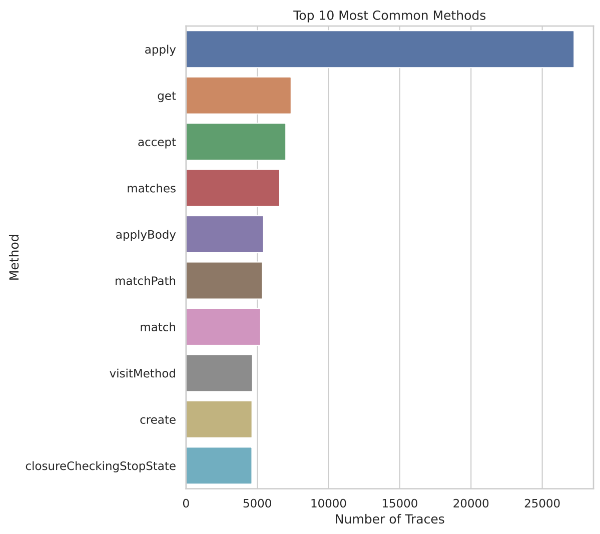
FIGURE 4.7. Boxplot of number of Java calls per trace in dataset. Plot covers only traces for which Java Calls, Java Calls with Boundaries, and Source Code are available.

4.2.3. Semantics of Java Traces. How are many traces as there per called method name? Does every called method name have approximately the same number of traces? In Figure 4.6 and Figure 4.7, we try to answer these questions. We analyze the distribution of the number of Java calls in the dataset. We note that 36,544 traces exceed a length of 768.

We proceed by gaining insight into the *semantics* of Java traces. In Figure 4.8b we see that the method identifier `apply` has the most traces recorded, with over 25,000 unique (by call sequence) traces associated. This is approximately 20,000 more than the next most frequent method `get`. We note that this distribution has an extremely long tail, which is evident in Figure 4.8a (note the log scale). Of the 17,219 unique FQNs, the top 100 FQNs account for 32.51% of all traces, the top 308



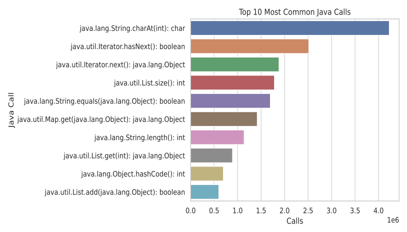
(A) Log scaled plot of number of traces per FQN.



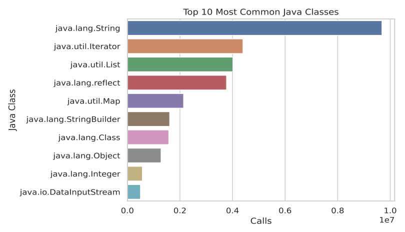
(B) Plot of methods with highest number of traces. Method refers strictly to identifier name, not fully qualified name.

FIGURE 4.8. Fully Qualified Name (FQN) and identifier frequency plots

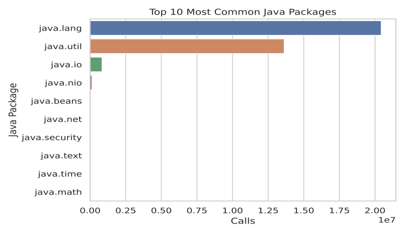
account for 50%, the top 8,286 account for 95%, and the remaining 8,933 account for just 5% of all traces.



(A) Top ten most common Java calls. Colors are for distinguishability and do not carry further information.



(B) Top ten most common classes of encountered Java calls. `java.lang.reflect` is the exception, as it is a package rather than a class. Colors are for distinguishability and do not carry further information.



(C) Top ten most common packages of encountered Java calls. Colors are for distinguishability and do not carry further information.

FIGURE 4.9. Java call plots

How are the Java core libraries used? By looking at the distribution of Java calls with respect to their top level package, down to typed calls, we gain insight into how the Java language is used. This is presented in Figure 4.9. From Figure 4.9a, the most common operation is on strings:

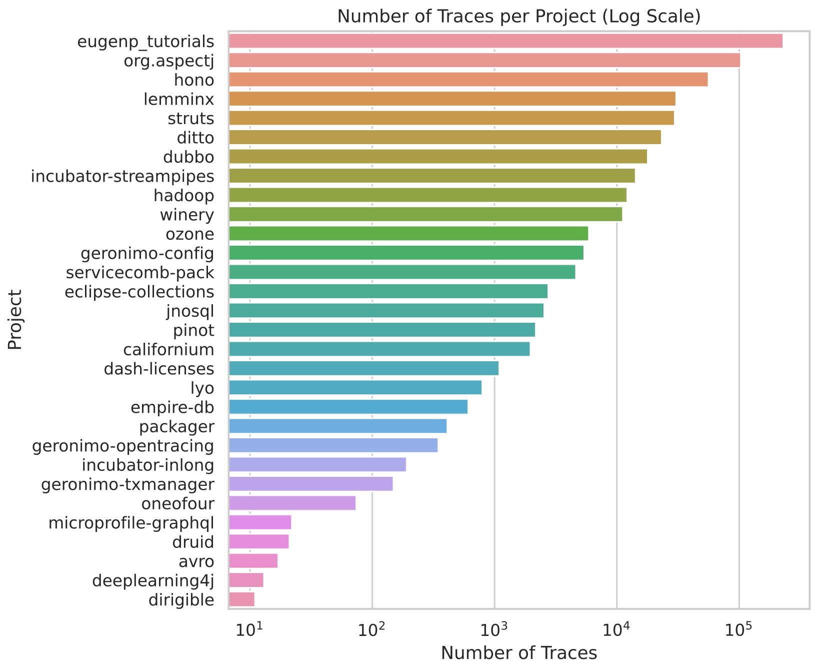
`chatAt`, which retrieves the character at an index in the string. Unsurprisingly, the iterator methods `hasNext` and `next` are frequent. List operations such as addition, retrieval, and length are also frequent. This observation is reflected in Figure 4.9b: operations on strings, iterators, and collections (lists & maps) classes are used very frequently. Interestingly, the boxed form of the primitive integer type is also commonly used. Finally, Input/Output (IO) operations in the form of `DataInputStream` are among the most widely used Java classes.

Finally, we examine how top-level packages are used in Figure 4.9c. `lang` and `util` are by far the most used packages, with other packages such as Java `beans` paling in comparison.

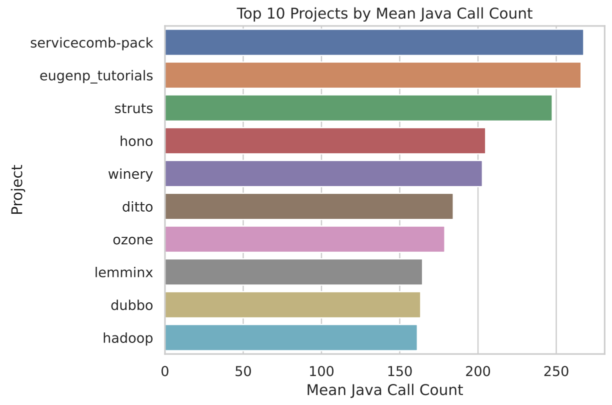
4.2.4. Project Level Analysis. Where do our traces come from? How do different projects differ in terms of their trace lengths? These questions are addressed in Figure 4.10. "eugenp_tutorials" provides the most traces by a significant margin. This is likely due to the structure of this project: many modules consisting of tutorials on how to use a different Java package *e.g.* Spring Boot. As a result, many traces extend into dependencies. From Figure 4.10b and Figure 4.10c we observe that "servicecomb-pack" has the longest traces on average. The top six projects by mean & median Java call count are the same, but below these we observe differences between the measures of central tendency. For example, "ozone" has a high median Java call count, but does not make it into the top ten by median. This suggests various projects have significant differences in how their Java call counts are distributed. A high mean but lower median implies more outliers.

4.2.5. Token Level Analysis. A Type-Token plot (Figure 4.11a) represents the relationship between the number of unique tokens (Java calls) and the total number of tokens as one iterates over all traces, randomly shuffled. In natural text, as more tokens are added *i.e.* as one "reads" further through a text, the number of types (unique words, in our case Java calls) increases, but usually at a diminishing rate. Yet, in the trace context we see that types encountered increases exponentially. This implies that there are a large number of *rare* tokens.

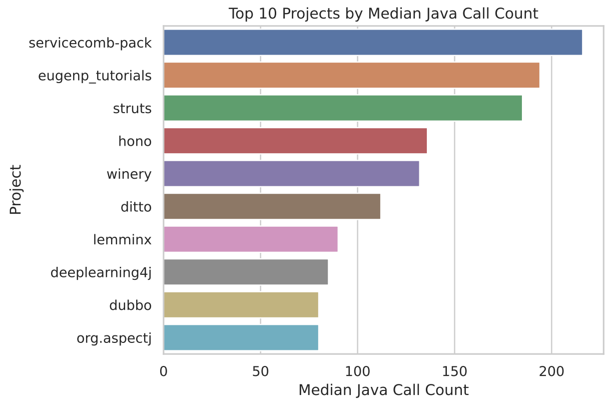
On the other hand, Figure 4.11b represents each unique Java call (token) on the x axis, and the number of times it occurs in the dataset on y. The tokens are sorted in descending order by number of occurences. This shows again that Java core libraries are not used equally: some are far more commonly used than others. The plot also elucidates the rare token dilemma: by the 565th



(A) Log scaled bar chart of number of traces collected by project.



(B) Top ten projects by mean Java call count. Colors are for distinguishability and do not carry further information.



(C) Top ten projects by median Java call count. Colors are for distinguishability and do not carry further information.

FIGURE 4.10. Project level Java call plots

token the counts drop under 10^3 *i.e.* 1000, under 100 by the 1,129th token, and under 10 by the 1,830th token.

The rare word problem is widely studied in the Natural Language Processing community. Tokens that occur too rarely are not useful for training of large language models, as they need a large number of examples to adjust their weights effectively. One can expect *instability* among the representations of rare tokens: outlier values that have the potential to "confuse" a model due to their magnitude.

4.2.6. Combinatorics of Traces. Can traces provide enough signal to be useful for retrieval? We provide some simple analysis using combinatorics. To find the number of possible traces given our constraints of (i) maximum length N of 768 calls (ii) 2,407 unique Java calls k, we begin by finding the number of combinations for a sequence of a given length n:

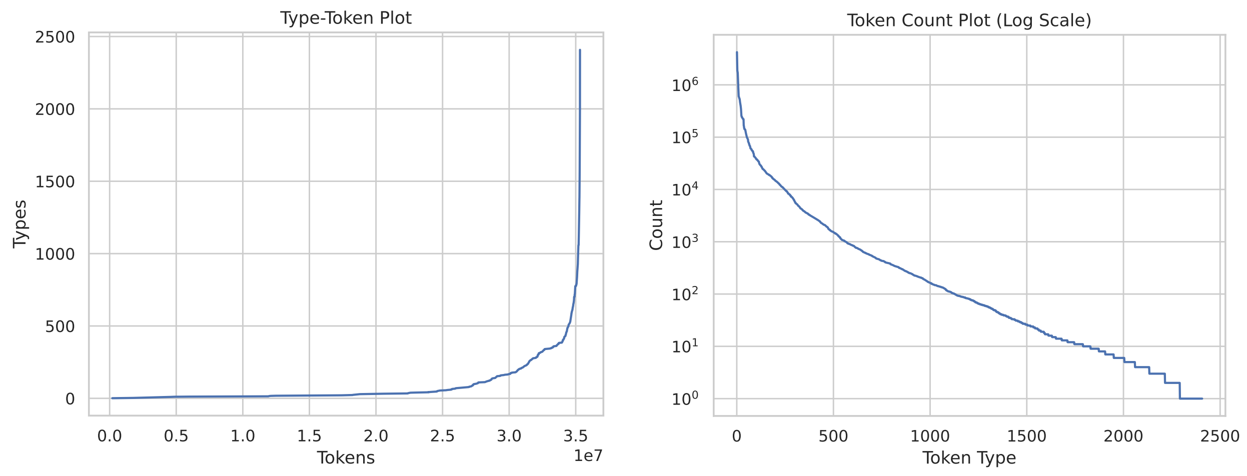
$$\text{Combinations} = k^n$$

Since n may range from 1 to 768, we proceed by finding the sum:

$$\text{Total combinations} = \sum_{n=1}^{N} (k^n)$$

We expand the summation to find:

$$\text{Total combinations} = 2407 + 2407^2 + \cdots + 2407^{768}$$



(A) Type-Token Plot (B) Token count plot, log scaled y axis.

FIGURE 4.11. Token frequency plots

As this is a geometric series with first term $a = 2407$ and a common ratio of $r = 2407$, we can find its sum with the closed form:

$$\text{Sum} = \frac{a(1 - r^n)}{(1 - r)}$$

For our $N = 768$:

$$\text{Total combinations} = \frac{2407(1 - 2407^{768})}{(1 - 2407)}$$

It is evident that the number of possible traces is staggering. We believe the aperture of Java calls and the average & maximum sequence length provides sufficient distinguishability between traces that a retrieval task is feasible.

4.3. Obfuscated Tracing

Once a dataset of traces is constructed, we create a set of test suites that have methods with source code associated. We modify each (sub)module `pom.xml` programatically to add the ProGuard Maven plugin[4] to the build process. We define new class and test class directories and point the ProGuard plugin to them. We specify all JVM JMODs and the submodule's classpath as libraries for ProGuard to reference. JMODs contain core Java library functions. As part of the ProGuard configuration, we do not include dependencies (not inlined into obfuscated class directory), we do not shrink nor optimize, and keep all annotations. We also preserve class names that match the following regular expressions: `**.Test*`, `**.*Test`, `**.*Tests`, or `**.*TestCase`. We then rerun the test suites with the added obfuscation step. This produces traces of the same test suites, but the methods under test are obfuscated.

4.4. Modeling

Our model is an adapted BERT [7] model with a custom tokenizer designed for call sequences, which represents each call signature as one unique token. Our model has an attention window size of 768, an intermediate size of 2048, a hidden size of 512, 8 attention heads, 6 hidden layers, and dropout of 10%. We first pretrain the model on call sequences from the JTD, using the standard Masked Language Modeling (MLM) masking technique, and default mask probability of 15%. We

[4]ProGuard Maven Plugin

train for 400 epochs. Once the model is pretrained, it produces embeddings for each trace in the corpus. These embeddings can then be used for the retrieval of the source code.

4.5. Retrieval

Once a model is trained on trace sequences, we pass all trace sequences through the model and store the embedding of the `[CLS]` token (short for classification). This token is used in BERT-style models as a sentence embedding, *i.e.* a single vector representation of the entire sequence. We store all embeddings of our candidate set in a FAISS [20] vector database. To retrieve the source code of an unknown trace, we simply embed the trace using the same methodology, and then retrieve the k-nearest neighbors of the embedding in the candidate set vector space. As all candidates have associated source code, this allows us to propose likely source snippets for the unknown trace.

4.5.1. Evaluating Performance. To evaluate the performance of our approach, we perform the retrieval procedure for each item (query) in the WITH LIBRARIES and WITHOUT LIBRARIES sets. For each query, we use a code similarity metric to compare the query source code and each of the $k = 10$ candidate source codes retrieved. Once all queries have had their candidates retrieved, we take the average maximum value at k. For example, at $@k = 3$, for each query, the top score of the first three candidates is selected, and finally averaged across all queries. The question on how to best compare code similarity is still an open one in literature. Therefore, we consider three code similarity metrics, and opt for CodeBLEU and BERTScore (F1) in our evaluations[5].

CodeBLEU (Ren et al. [32]): Absorbs the strength of BLEU [29] in the n-gram match, and further injects code syntax via abstract syntax trees (AST) and code semantics via data-flow. Each CodeBLEU score consists of a set weighted components: n-gram match, weighted n-gram match, syntax match, and dataflow match. We also present our own weighting *Syntax/Dataflow match* which gives n-grams no weight, and gives syntax and dataflow 50% respectively. This allows us to compare code similarity in terms of just syntax & dataflow, while ignoring difference in variable names.

CrystalBLEU (Eghbali et al. [11]): BLEU finds common n-grams in unrelated programs, which makes distinguishing similar pairs of programs from dissimilar pairs hard. CrystalBLEU

[5]Figure 5.6b compares CodeBLEU and CrystalBLEU

mitigates the distinguishability problem. The metric maintains the desirable properties of BLEU, such as handling partial code, applicability to all programming languages, high correlation with human judgment, and efficiency, in addition to reducing the effects of the trivially matched n-grams.

BERTScore (Zhang et al. [39]): BERTScore computes a similarity score for each token in the candidate sentence with each token in the reference sentence. However, instead of exact matches, it computes token similarity using contextual embeddings. BERTScore correlates better with human judgments and provides stronger model selection performance than existing metrics. We supply the CodeBertBase [13] model to BERTScore and use the 10th layer for the contextual embeddings. This follows Zhang et al. [39] who determine that the best layer to use for roberta-base is 10. Given that CodeBertBase is a Roberta [26] model of the same size, we chose to use the same layer.

CHAPTER 5

Results

5.1. Analysis of Embedding Space

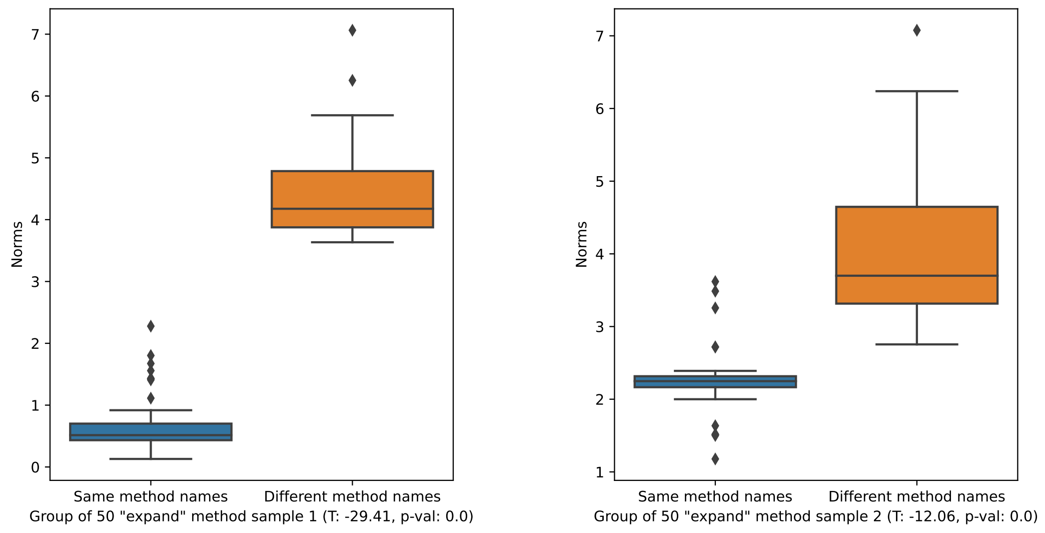


FIGURE 5.1. Paired boxplot for fifty trace samples of method “expand” and fifty random traces of methods of differing names.

5.1.1. Statistical Testing. In Figure 5.1 we present one example from our t-Test procedure. In this procedure, we embedded the traces using STraceBert and sampled one-hundred method names for which there were at least fifty traces. For each method name, we randomly sampled fifty traces and computed the vector norm of their embedding *i.e.* the distance of the vector from the norm (zero) *with respect to the average of the samples.* For each method name, we randomly sampled fifty *other* traces of methods with a different name. We then computed the average and the norm, and performed a paired t-Test on the two sets of norms, and plotted the distributions on paired box plots. This produced one-hundred box plots, that consistently showed that traces of methods of the same name are closer to each other in the high dimensional space than the differing names. This quantitative analysis provides encouraging results that the embedding space captures the semantic meaning of traces.

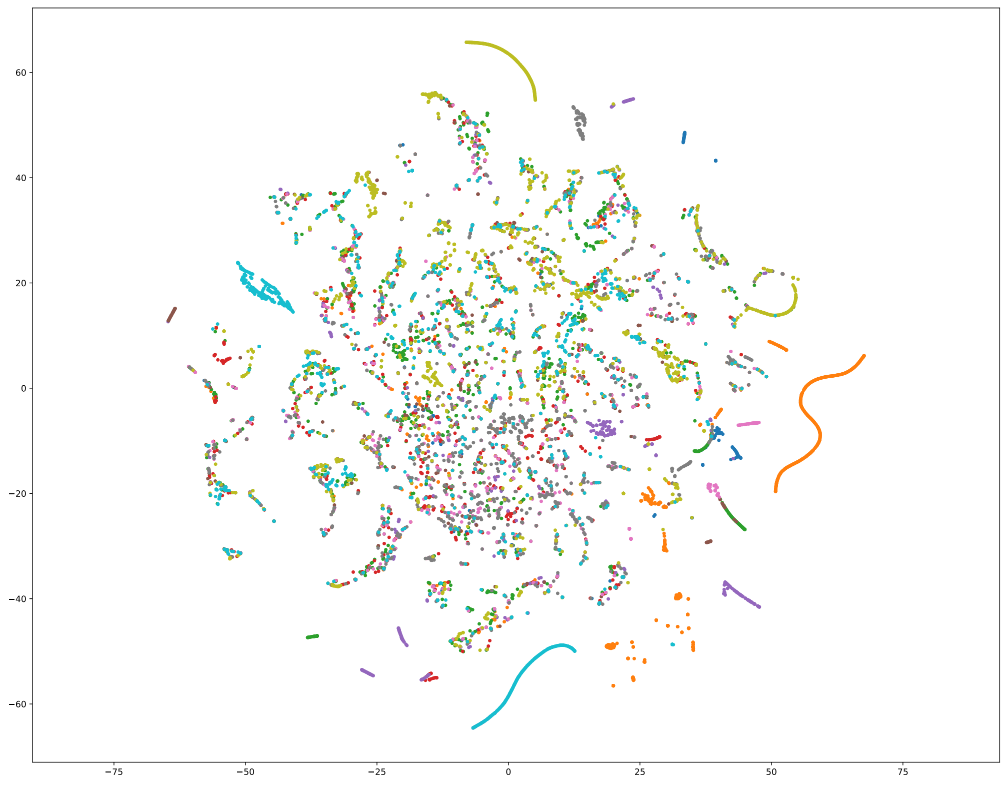


FIGURE 5.2. t-SNE plot of one-hundred sampled method names, fifty traces per name. Color denotes class *i.e.* method name.

5.1.2. Qualitative t-SNE. We subsequently performed a t-distributed stochastic neighbor embedding (t-SNE) [27] dimensionality reduction on the five thousand traces *viz.,* fifty traces per one-hundred sampled method names. The goal of t-SNE is to represent datapoints that are near to each other *i.e.* have a low Euclidean distance in the high dimensional space, as close to each other in a low dimensional space, in our case two dimensions. While the results of a t-SNE reduction depend on hyperparameters such as *perplexity*, we observe that traces of the same class *i.e.* method name (denoted by color) are close to each other, forming clusters. This qualitative analysis further suggests that the embedding space captures the semantic meaning of traces and will be suitable for a nearest-neighbor retrieval approach.

5.2. Retrieval Performance

Our goal is to retrieve candidate snippets of code for a reverse engineer. We presume that presenting five examples to the reverse engineer will allow them to better understand the method they are working with. For this purpose, all metrics are presented in the form of **top@k**. In our evaluation, we utilize the CodeBLEU [32] code similarity measure. The metrics are calculated based on the similarity of the retrieved code snippets against the ground-truth code snippet. When aggregating a metric, we take the average of the max score at *k* across all scores. For example, top@5 would be the average maximum CodeBLEU score in the first 5 candidates retrieved for each query, across all queries.

To evaluate how useful a retrieved snippet is, we measure code similarity between the query source code versus each retrieved candidate snippet. For each trace in our evaluation sets (With Libraries, Without Libraries), we retrieve *k* nearest-neighbors, where *k = 10*, from our candidate set using FAISS [20] and the trace embedding from our model. We then calculate and record the similarity metrics.

We present our results in Table 5.1. As a reference, we provide a random baseline where the candidates are picked randomly, BM25 [33], and Codex. For Codex, we sample 100 queries from each query set, and create an 8-shot prompt following Nashid et al. [28] by finding the nearest neighbors of the trace query in the candidate set, filtered to have less than 100 Java calls and less than 300 characters of source code due to context window limitations. We also compare a straw-man retrieval using the CodeBERT [13] embedding of the source code. Due to the source code overlap but no trace sequence overlap, between the With Libraries set and the candidate set, retrieval via CodeBERT finds exact matches. We then present retrieval performance using trace embeddings, and find an average CodeBLEU of 86.26 @1. Further analysis showed that the CodeBLEU values were not a result of exact matches (see subsection 5.2.2), suggesting similar code

	Candidates	Without Libraries		With Libraries	
	Metric	CodeBLEU	BERTScore F1	CodeBLEU	BERTScore F1
Method	Top@k				
STraceBert	1	22.53	82.07	79.14	95.66
	3	27.80	83.72	89.19	97.81
	5	30.27	84.37	92.68	98.56
	10	32.44	85.09	95.18	99.06
Boundaries	1	23.21	82.28	92.30	98.49
	3	26.75	83.44	94.92	99.00
	5	28.79	83.94	96.11	99.23
	10	30.82	84.58	96.73	99.36
BM25	1	19.73	81.38	43.21	87.95
	3	24.49	83.05	55.10	90.48
	5	26.50	83.63	60.10	91.68
	10	28.98	84.37	66.16	93.02
Codex	1	22.08	-	29.85	-
	5	31.45	-	38.39	-
Random	1	16.74	80.22	17.99	80.56
	3	23.64	82.75	25.75	83.18
	5	26.45	83.63	29.03	84.17
	10	29.71	84.54	33.55	85.42
CodeBERT	1	27.70	84.66	93.37	98.71
	3	30.32	85.58	96.62	99.27
	5	31.26	85.86	97.08	99.36
	10	33.52	86.41	98.03	99.60

Table 5.1. Retrieval performance in terms of average maximum CodeBLEU@k. Codex results for @3 and @10 were not recorded, and BERTScore was not evaluated as part of separate experiment.

produces similar sequences of calls. This presents strong evidence that our trace embeddings can be used to find similar code.

5.2.1. Analysis of Retrieval Performance. We present the results from Table 5.1 as a plot in Figure 5.3. CodeBERT is intentionally omitted as it is retrieving embeddings of source code, rather than traces.

Unsurprisingly, random candidate retrieval performs worst, performing between 33.55 and 17.99. In contrast, retrieval with Codex only performs slightly better than random, outperforming only at the top candidate with a score of 29.85. BM25 performs well considering the simplicity of the approach, but performance quickly drops off as the top candidate pool decreases in size, going from 66.16 to 43.21. Trace embeddings *without* boundaries only perform slightly worse than

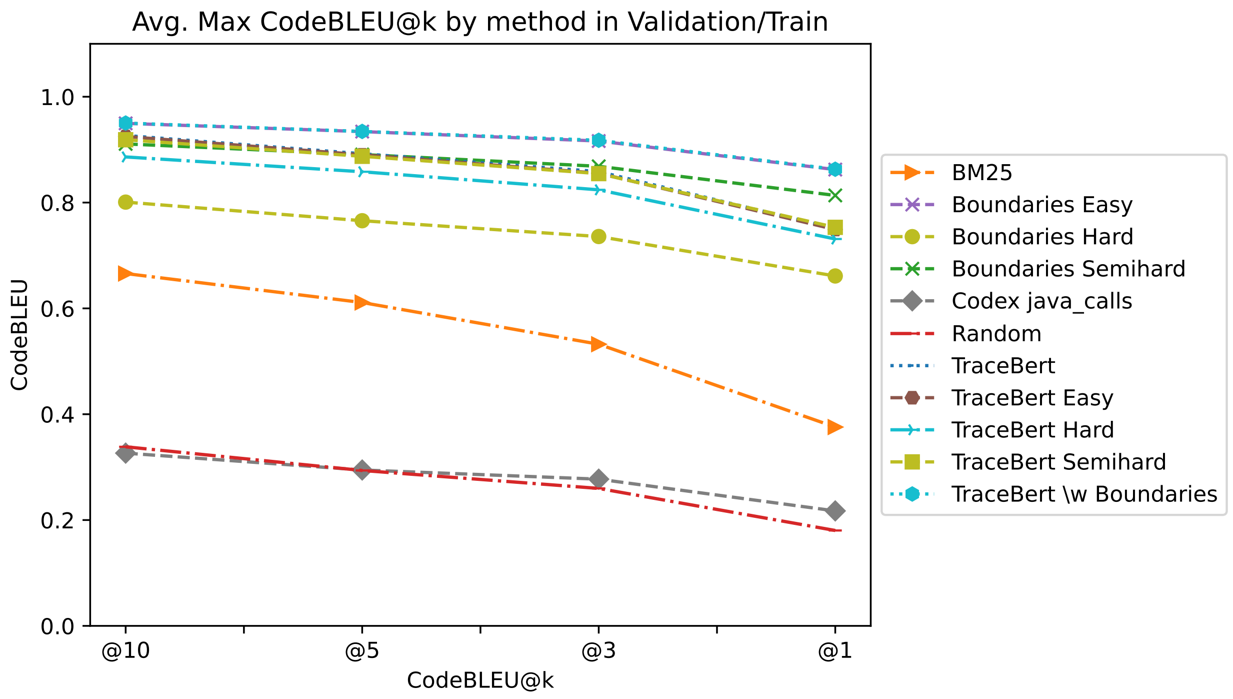


FIGURE 5.3. Performance of all methods at top k@10, @5, @3, and @1.

with boundaries, achieving CodeBLEU of 95.18 at ten, down to 79.14 at the top candidate. It is clear, that embeddings of traces *with* boundaries consistently perform best, achieving an average maximum CodeBLEU of 96.73 in the top ten candidates, 96.11 in top five, 94.92 in top three, and 92.30 for the top candidate. This suggests that boundaries are a useful piece of information for the model. It is also evident, that contrastive training using Easy, Hard, and Semihard triplets did not have a positive impact for either of the trace embedding models. In the case of hard triplets, performance decreased significantly for both models. Easy triplets appear to have no impact on performance.

5.2.2. Retrieval Distribution. We performed our retrieval procedure and analyzed the CodeBLEU distribution of the retrieved candidates. We present the results in Figure 5.4 as a violin plot. This plot shows the distribution of the CodeBLEU score of the ground truth source code of the query trace *versus* the source code of the top k candidates.

5.2.3. Analysis of Codex prompting methods. In this section we analyze the performance of Codex with two different prompting mechanisms. "Codex java_calls" follows a simple eight-shot prompt with *random* traces as examples, whereas “Codex Ret. Prompt” follows Nashid et al. [28], where we first retrieve the nearest neighbors from the candidate set, and use their ground truth source and Java call sequence as examples. Figure 5.5 shows that the approach of Nashid et al. [28] slightly outperforms the naive few-shot prompting approach. In both cases, we restrict the

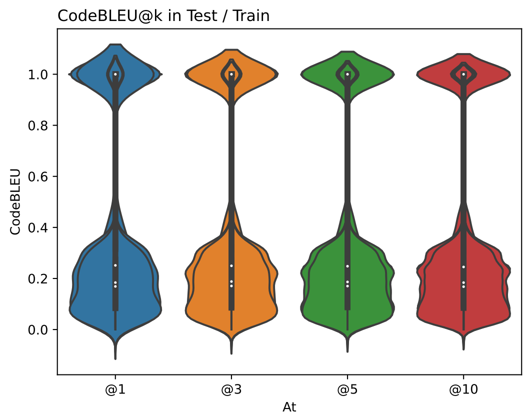


FIGURE 5.4. Violin plot of CodeBLEU distributions at top 1, top 3, top 5, and top 10 when retrieving with STraceBert embedding in WITHOUT LIBRARIES setting.

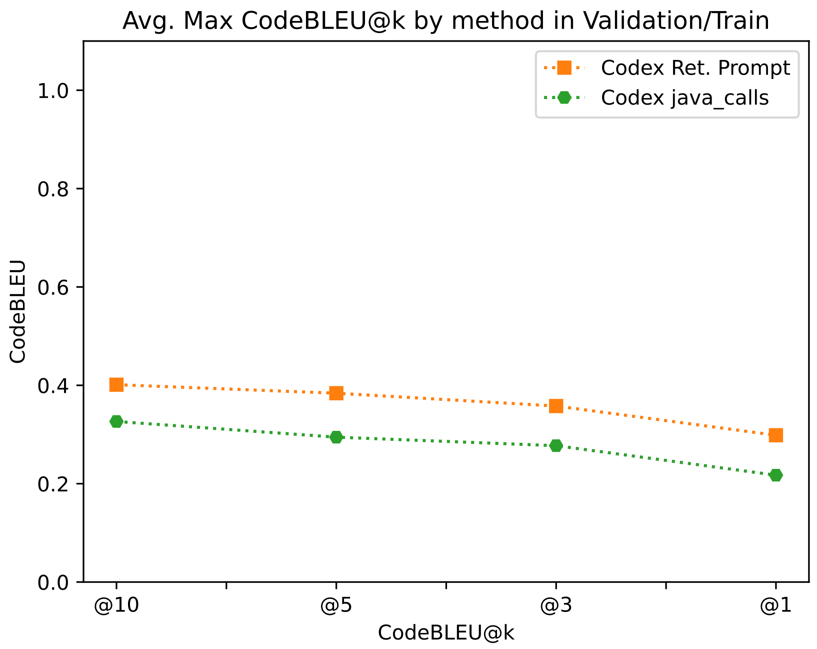


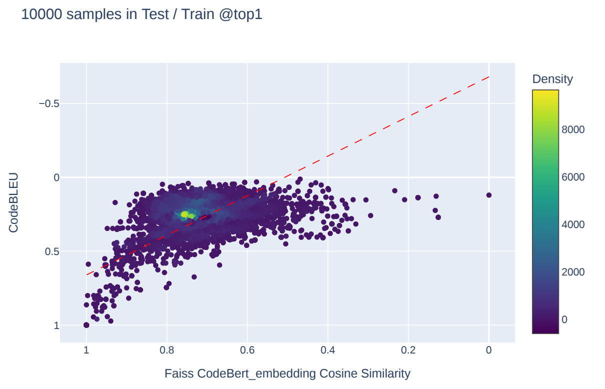
FIGURE 5.5. Codex prompting mechanisms

few-shot examples to have a source code length of less than 300 characters and less than 100 Java calls, as described in chapter 5. CodeBLEU values of “Codex java_calls” are consistently lower than “Codex Ret. Prompt”. This suggests that providing known source code of similar *traces* helps a large language model produce more relevant code. Despite this, it is evident that few-shot prompting is not enough for Codex to actively learn trace semantics and produce relevant code in a known libraries setting.

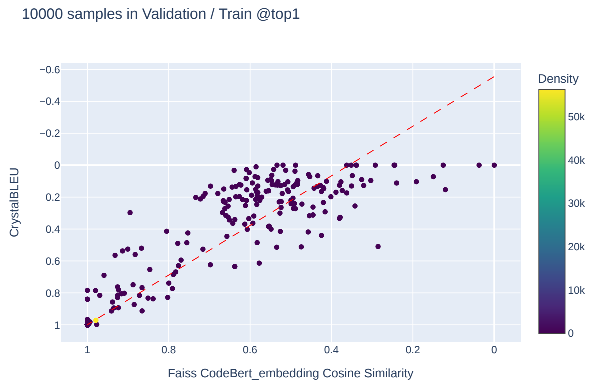
5.2.4. Code Similarity *vs.* Embedding Distance. In an ideal embedding space for traces or source code, the cosine similarity of two given embeddings should correlate highly with their code similarity scores (CodeBLEU, CrystalBLEU, BERTScore).

In Figures 5.6a to 5.6d we explore this relationship. In Figure 5.6a and Figure 5.6b we explore the relationship between CodeBLEU and CrystalBLEU respectively, for our CodeBERT strawman. As CodeBERT is trained directly on *source code* tokens, we expect its embedding space to better capture code similarity than trace embeddings do. In both Figures 5.6a and 5.6b, we observe a decent line of best fit. This suggests two things: (i) CodeBERT embedding *distances* correlate highly with code similarity metrics (ii) CrystalBLEU behaves similarly to CodeBLEU.

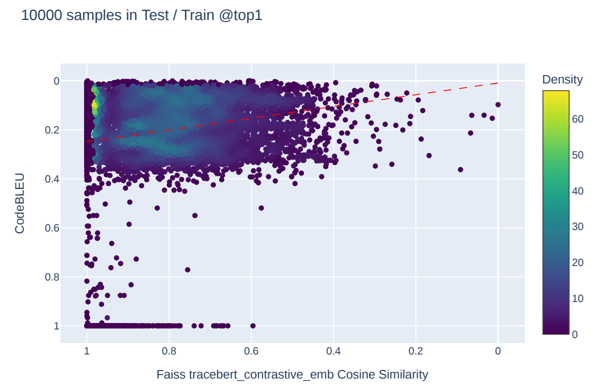
In contrast, in Figure 5.6c and Figure 5.6d we observe a lower correlation between embedding cosine similarity and the code similarity metric. This suggests that the trace embedding space does not capture code similarity as effectively.



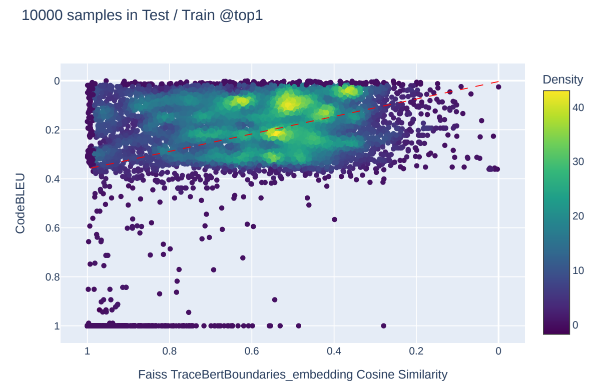
(A) CodeBLEU vs Cosine Similarity, for CodeBERT embeddings



(B) CrystalBLEU vs Cosine Similarity, for CodeBERT embeddings



(C) CodeBLEU vs Cosine Similarity, for STraceBert embeddings



(D) CodeBLEU vs Cosine Similarity, for STraceBert with Boundaries embeddings

FIGURE 5.6. Candidate source code to query ground truth source code CodeBLEU score (y), versus straw-man CodeBERT *source code* embedding cosine similarities (x) in Test / Train pair, top candidate only (*@1*).

CHAPTER 6

Discussion

6.1. Summary

In summary, our results show that trace embeddings are an effective mechanism for retrieving source code in a *known libraries* setting, with the top candidate having an average CodeBLEU ("performance") of 79.14 between it and the ground truth query source code. In this setting, the performance increased over the random baseline of 17.99 CodeBLEU by 339.91%. Including call boundaries, we observed a 16.63% increase over trace embeddings without boundaries under the same conditions, resulting in a 413.06% increase over random. Source code retrieval from a candidate set *without* libraries demonstrated much weaker performance of 22.53 CodeBLEU, and 23.21 CodeBLEU with call boundaries, exhibiting a mere 3% increase in CodeBLEU between the two. Nonetheless, an increase over random of 34.59% and 38.65% was observed.

However, this must be brought into context of the random baseline's average maximum CodeBLEU at rank five in the WITHOUT LIBRARIES setting being 29.03, whereas trace embeddings with and without boundaries resulting in 28.79 and 30.27 CodeBLEU respectively, implying that trace embeddings were barely better than random. Surprisingly, the performance of the traces with boundaries is *lower* than just call sequences at five. As a `[CALL]` special token occupies one position, we hypothesize that the trace embeddings without boundaries capture more valuable information *viz.,* more calls, than the boundaries in the WITHOUT LIBRARIES setting, considering the limited input window.

Finally, we place these results in context with respect to our upper bound performance represented by the *CodeBERT* embedding. In this case, the retrieval is performed on the embedding of the *ground truth source code vs.* the embeddings of the candidate *source codes*. Therefore, in the WITHOUT LIBRARIES setting, an absolute upper bound of 32 CodeBLEU is presumed. Considering this, the trace embeddings without boundaries performed highly at 30.27 CodeBLEU at five.

In contrast, our baselines consistently performed worse than trace embeddings in the With Libraries setting. The best performing baseline was BM25, achieving 42.21 CodeBLEU, followed by Codex with 29.85 CodeBLEU, followed by random with 17.99. The upper bound given by the *code* embeddings is approximately 93 CodeBLEU. Conversely, the Without Libraries setting *at five* yielded more interesting characteristics, where Codex was the best performing model with 31.45 CodeBLEU, followed by trace embeddings with 30.27 CodeBLEU, and then random retrieval with 29.03 CodeBLEU. In both cases, only a slight increase over random was observed, suggesting that call boundaries were not as significant in this setting, and that large language models of code can produce code in a few-shot setting that is more similar to the query call sequence than non-library code in a candidate set. In particular, this final observation is supported by Codex outperforming CodeBert *code embeddings*, which represent the upper bound on retrieval performance from the candidate set.

6.2. Interpretations & Implications

We draw multiple conclusions from these results.

(1) **Call sequences can distinguish source code:** We demonstrate that call sequences recorded during dynamic analysis can be mapped to *their source code.* This implies that the mere occurrences of calls and their ordering is a distinguishing characteristic of source code. This opens the doors for future work in not only reverse engineering, but understanding behavior of source code from runtime observations.

(2) **Bidirectional transformers are effective, generalized feature extractors:** The substantial performance gains of embedding-based retrieval over classic term frequency approaches such as BM25 imply that bidirectional transformers trained using the unsupervised masked language modeling objective are effective at learning and constructing latent features from their training sequences. If they were not, similar traces in terms of semantics would *not* be near each other in the embedding space and thus a low retrieval performance would be observed. Additionally, this demonstrates that bidirectional transformers are by no means limited to natural languages. Such an observation can inform future work and practitioners in domains dealing with non-language data *e.g.* time

series analysis, that maybe be able to apply such models to solve previously challenging problems. Recent work has indicated that transformers excel at vision and audio tasks as well. [22, 9, 4, 5, 14, 15, 16]

(3) **Large language models of code are few-shot learners of execution dynamics:** Despite most likely never having seen a Java call sequence in their training set, the GPT3 based Codex model was able to produce the most relevant source code of all methods in the WITHOUT LIBRARIES setting from just a few examples of call sequence → source code. The implication is that there is perhaps more to the model's internal representation & processing *viz.,* weight matrices, than mere random parroting of code tokens. Perhaps future work can explore how much such models can "understand" of *runtime* behavior of source code. This ties in closely to the symbolic execution task, where a model is provided source code and tasked with "pretending" to be a virtual machine and produce the *output* of the source code. Given that models can perform this task [35], it is plausible that they develop *meaning* so complicated that they can "reason" about such internal representations in complex ways *e.g.* converting source code to system call sequences, or vice versa.

6.3. Limitations & Threats to Validity

While our approach is robust, we acknowledge potential problems:

Dataset We restricted our dataset collection to popular, open source Java projects which use the Maven build system. It is feasible that high quality open source code presents a distribution in execution semantics that is not representative of all Java code.

Tracing JACKAL is an experimental dynamic analysis tool that fails frequently. We were unable to collect many of the test cases we expected to. One plausible explanation is the time limit on tracing and data processing, per test case, of 72 hours. It may be, that the collected data represents only shorter, simpler methods. Other unknown factors may influence whether tracing succeeds or not, which would negatively affect generalizability.

CHAPTER 7

Conclusion and Future Work

7.1. Conclusion

In conclusion, the results of our experiments demonstrate the effectiveness of the STraceBERT approach for the code retrieval task in software reverse engineering. We find that embedding Java application traces using a BERT-style model allows for effective source code retrieval, outperforming traditional information retrieval approach BM25, and modern few-shot prompting of Codex in a known-libraries scenario. Future work includes further examination of robustness of our approach against obfuscation, and further exploration of the Java Trace Dataset.

7.2. Future Work

Proposed future work extends along multiple axes: (i) development of dynamic analysis infrastructure (ii) exploration of the Java Trace Dataset (iii) development of trace learning and representation (iv) examination of robustness.

During our data collection stage, we encountered frequent failure of the instrumentation framework Jackal. Future work may include examining these failures in detail and developing solutions for them. Another line may investigate how the performance overhead of this task can be minimized, as the tooling results in significant overhead, limiting its practical applications. These improvements alone would result in significantly more samples in the dataset, which in our setting would improve performance due to an increased candidate pool.

While we explored the data set characteristics in detail, we primarily centered our work on the task of embedding and retrieval of source code. Future work may investigate the applicability of trace sequences for tasks such as defect prediction, clone detection, autoregressive source code production, or dependency analysis to name a few.

During our study, we also posited whether we could exploit the tree structure of call *graphs*. Our token based trace sequences are linearized trees, yet the call graphs observed during runtime analysis contain rich information in their *edges* and implicit hierarchy. Future work may investigate alternative approaches to trace representations, such as using graph neural networks to embed call graphs rather than token-based models to embed linearized trees. Additionally, learning objectives beyond random token masking could be studied. We formulated various tasks in such a regime, such as parent-node prediction, child-node prediction, sibling-node prediction, cousin-node prediction, clique prediction, and prediction of sub-graphs *i.e.* masking of *several* nodes. Such tasks may help models learn better representations, and may be combined with tree transformer approaches to exploit the graph nature of code and call graphs. [2, 3, 17, 23, 31, 34, 36, 37, 38, 40]

APPENDIX A

Appendix

A.1. List of projects

Table A.1. Table of all gathered projects and commit hash

Organization	Project	Commit Hash
apache	activemq	9956dd6
eclipse	ajdt	a0e57f1
apache	flink	d0ba79e27f
apache	avro	b184c35
eclipse	birt	25528861
eclipse	californium	397f83a
eclipse	californium.actinium	f239096
eclipse	californium.tools	e26c92b
apache	camel-kamelets	572d488
eclipse	capella	bfda3061
eclipse	capella-basic-vp	ae9df94
apache	commons-daemon	af00cbc
apache	commons-io	f8a8069
apache	commons-lang	4753c6b
apache	cxf-fediz	0ba80c1
eclipse	dash-licenses	abac0f5
dbeaver	dbeaver	9faf767f71
eclipse	deeplearning4j	464f1fd
eclipse	dirigible	80a5f4a
eclipse	ditto	dfca359
apache	druid	b86f2d4
apache	dubbo	27fa230
eclipse	eclipse-collections	990295a
eclipse	deeplearning4j	029b84e2b7
eclipse-vertx	vert.x	8377be65e
apache	empire-db	69e9a48
eugenp	tutorials	d6e8da14cd
apache	geronimo-config	1e09399
apache	geronimo-health	fc5ec7b
apache	geronimo-metrics	8e3105d
apache	geronimo-opentracing	2015e31
apache	geronimo-txmanager	8969a23
apache	hadoop	ec2fd013
apache	hbase	5dc663e
eclipse	hono	b849842
apache	incubator-inlong	72c7e90
apache	incubator-kyuubi	3cbedea
apache	incubator-streampipes	a3fc041
jenkinsci	jenkins	c772d810f6
eclipse	jetty.docker	ed2952f
eclipse	jetty.project	34c21ff
apache	jmeter	6196e40
eclipse	jnosql	4a052b3
apache	kafka	3b534e1
eclipse	kapua	36190a2
eclipse	kitalpha	4a2be6c3
eclipse	lemminx	3ae1f16
apache	logging-log4j2	bf21070
eclipse	lyo	45b32ad
apache	maven-dist-tool	ffe2741
apache	maven-site	f0e4665
eclipse	microprofile-config	87ea205
eclipse	microprofile-graphql	d4ac51a
apache	mnemonic	eca3dd5
netty	netty	1fe29bd
eclipse	oneofour	9e6bc67
openzipkin	zipkin	7bd40d01e
eclipse	org.aspectj	0fe9c68
apache	ozone	93631a1
eclipse	packager	703bab6
apache	pinot	27d690d
eclipse	rdf4j	6c2fbb1
eclipse	reddeer	c5a50ef
apache	servicecomb-pack	98da54a
apache	spark	43c89dc
apache	storm	d48b1f0
apache	struts	115fef2
eclipse	tm4e	3d3f6a2
apache	tomcat	5515c06
apache	tomee-jakarta	6700451
apache	tomee-jakartaee-api	3aad2e4
apache	tomee-tck	09f756f
eclipse	tycho	6689f64
eclipse	winery	6320be7

Bibliography

[1] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr. Improving few-shot prompts with relevant static analysis products, 2023. arXiv: `2304.06815 [cs.SE]`.

[2] U. Alon, R. Sadaka, O. Levy, and E. Yahav. Structural language models of code. (arXiv:1910.00577), July 2020. DOI: `10.48550/arXiv.1910.00577`. URL: `http://arxiv.org/abs/1910.00577`. arXiv:1910.00577 [cs, stat].

[3] N. D. Q. Bui, Y. Yu, and L. Jiang. Treecaps: tree-based capsule networks for source code processing. (arXiv:2009.09777), Dec. 2020. DOI: `10.48550/arXiv.2009.09777`. URL: `http://arxiv.org/abs/2009.09777`. arXiv:2009.09777 [cs].

[4] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko. End-to-end object detection with transformers. (arXiv:2005.12872), May 2020. DOI: `10.48550/arXiv.2005.12872`. URL: `http://arxiv.org/abs/2005.12872`. arXiv:2005.12872 [cs].

[5] K. Chen, X. Du, B. Zhu, Z. Ma, T. Berg-Kirkpatrick, and S. Dubnov. Hts-at: a hierarchical token-semantic audio transformer for sound classification and detection. (arXiv:2202.00874), Feb. 2022. URL: `http://arxiv.org/abs/2202.00874`. arXiv:2202.00874 [cs, eess].

[6] Claudio Spiess. Java-trace-dataset (revision 905fe11). Hugging Face, 2023. DOI: `10.57967/hf/1020`. URL: `https://huggingface.co/datasets/claudios/java-trace-dataset`.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: pre-training of deep bidirectional transformers for language understanding. (arXiv:1810.04805), May 2019. DOI: `10.48550/arXiv.1810.04805`. URL: `http://arxiv.org/abs/1810.04805`. arXiv:1810.04805 [cs].

[8] S. H. H. Ding, B. C. M. Fung, and P. Charland. Asm2vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, May 2019. DOI: `10.1109/SP.2019.00003`.

[9] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: transformers for image recognition at scale. (arXiv:2010.11929), June 2021. DOI: 10.48550/arXiv.2010.11929. URL: http://arxiv.org/abs/2010.11929. arXiv:2010.11929 [cs].

[10] Y. Duan, X. Li, J. Wang, and H. Yin. Deepbindiff: learning program-wide code representations for binary diffing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: https://www.ndss-symposium.org/ndss-paper/deepbindiff-learning-program-wide-code-representations-for-binary-diffing/.

[11] A. Eghbali and M. Pradel. Crystalbleu: precisely and efficiently measuring the similarity of code. en. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 341–342, Pittsburgh, PA, USA. IEEE, May 2022. ISBN: 978-1-66549-598-1. DOI: 10.1109/ICSE-Companion55297.2022.9793747. URL: https://ieeexplore.ieee.org/document/9793747/.

[12] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi. Binclone: detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87, June 2014. DOI: 10.1109/SERE.2014.21.

[13] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: a pre-trained model for programming and natural languages. (arXiv:2002.08155), Sept. 2020. DOI: 10.48550/arXiv.2002.08155. URL: http://arxiv.org/abs/2002.08155. arXiv:2002.08155 [cs].

[14] M.-I. Georgescu, E. Fonseca, R. T. Ionescu, M. Lucic, C. Schmid, and A. Arnab. Audiovisual masked autoencoders. (arXiv:2212.05922), July 2023. URL: http://arxiv.org/abs/2212.05922. arXiv:2212.05922 [cs].

[15] Y. Gong, Y.-A. Chung, and J. Glass. Ast: audio spectrogram transformer. (arXiv:2104.01778), July 2021. URL: http://arxiv.org/abs/2104.01778. arXiv:2104.01778 [cs].

[16] Y. Gong, A. Rouditchenko, A. H. Liu, D. Harwath, L. Karlinsky, H. Kuehne, and J. Glass. Contrastive audio-visual masked autoencoder. (arXiv:2210.07839), Apr. 2023. URL: `http://arxiv.org/abs/2210.07839`. arXiv:2210.07839 [cs, eess].

[17] J. Harer, C. Reale, and P. Chin. Tree-transformer: a transformer-based method for correction of tree-structured data. en. (arXiv:1908.00449), Aug. 2019. URL: `http://arxiv.org/abs/1908.00449`. arXiv:1908.00449 [cs, stat].

[18] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu. Binmatch: a semantics-based hybrid approach on binary code clone analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 104–114, Sept. 2018. DOI: `10.1109/ICSME.2018.00019`.

[19] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019. arXiv: `1909.09436`. URL: `http://arxiv.org/abs/1909.09436`.

[20] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[21] A. Al-Kaswan, T. Ahmed, M. Izadi, A. Sawant, P. Devanbu, and A. van Deursen. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 260–271, Los Alamitos, CA, USA. IEEE Computer Society, Mar. 2023. DOI: `10.1109/SANER56733.2023.00033`. URL: `https://doi.ieeecomputersociety.org/10.1109/SANER56733.2023.00033`.

[22] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah. Transformers in vision: a survey. *ACM Computing Surveys*, 54(10s):1–41, Jan. 2022. ISSN: 0360-0300, 1557-7341. DOI: `10.1145/3505244`. arXiv:2101.01169 [cs].

[23] S. Kim, J. Zhao, Y. Tian, and S. Chandra. Code prediction by feeding trees to transformers. (arXiv:2003.13848), Mar. 2021. DOI: `10.48550/arXiv.2003.13848`. URL: `http://arxiv.org/abs/2003.13848`. arXiv:2003.13848 [cs].

[24] H. Koo, S. Park, D. Choi, and T. Kim. Semantic-aware binary code representation with bert. (arXiv:2106.05478), June 2021. DOI: `10.48550/arXiv.2106.05478`. URL: `http://arxiv.org/abs/2106.05478`. arXiv:2106.05478 [cs].

[25] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. L. Goues, G. Neubig, and B. Vasilescu. Dire: a neural approach to decompiled identifier naming. (arXiv:1909.09029), Oct. 2019. DOI: `10.48550/arXiv.1909.09029`. URL: `http://arxiv.org/abs/1909.09029`. arXiv:1909.09029 [cs].

[26] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. Roberta: a robustly optimized bert pretraining approach, 2019. arXiv: `1907.11692 [cs.CL]`.

[27] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[28] N. Nashid, M. Sintaha, and A. Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, 2023.

[29] 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*, pages 311–318, 2002.

[30] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray. Trex: learning execution semantics from micro-traces for binary similarity. (arXiv:2012.08680), Apr. 2021. DOI: `10.48550/arXiv.2012.08680`. URL: `http://arxiv.org/abs/2012.08680`. arXiv:2012.08680 [cs].

[31] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin. Integrating tree path in transformer for code representation. In *Advances in Neural Information Processing Systems*, volume 34, pages 9343–9354. Curran Associates, Inc., 2021. URL: `https://proceedings.neurips.cc/paper/2021/hash/4e0223a87610176ef0d24ef6d2dcde3a-Abstract.html`.

[32] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv:2009.10297 [cs]*, Sept. 2020. URL: `http://arxiv.org/abs/2009.10297`. arXiv: 2009.10297.

[33] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR'94: Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval, organised by Dublin City University*, pages 232–241. Springer, 1994.

[34] V. Shiv and C. Quirk. Novel positional encodings to enable tree-based transformers. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL: `https://proceedings.neurips.cc/paper/2019/hash/6e0917469214d8fbd8c517dcdc6b8dcf-Abstract.html`.

[35] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening. Concolic testing for deep neural networks. (arXiv:1805.00089), Aug. 2018. DOI: `10.48550/arXiv.1805.00089`. URL: `http://arxiv.org/abs/1805.00089`. arXiv:1805.00089 [cs, stat].

[36] Z. Tang, C. Li, J. Ge, X. Shen, Z. Zhu, and B. Luo. *AST-Transformer: Encoding Abstract Syntax Trees Efficiently for Code Summarization*, number arXiv:2112.01184. Dec. 2021. DOI: `10.48550/arXiv.2112.01184`. URL: `http://arxiv.org/abs/2112.01184`. arXiv:2112.01184 [cs] type: article.

[37] J. Villmow, A. Ulges, and U. Schwanecke. A structural transformer with relative positions in trees for code-to-sequence tasks. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–10, July 2021. DOI: `10.1109/IJCNN52387.2021.9533717`.

[38] Y.-S. Wang, H.-Y. Lee, and Y.-N. Chen. Tree transformer: integrating tree structures into self-attention. (arXiv:1909.06639), Nov. 2019. DOI: `10.48550/arXiv.1909.06639`. URL: `http://arxiv.org/abs/1909.06639`. arXiv:1909.06639 [cs].

[39] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi. Bertscore: evaluating text generation with bert. (arXiv:1904.09675), Feb. 2020. DOI: `10.48550/arXiv.1904.09675`. URL: `http://arxiv.org/abs/1904.09675`. arXiv:1904.09675 [cs].

[40] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann. Language-agnostic representation learning of source code from structure and context. (arXiv:2103.11318), Mar. 2021. DOI: `10.48550/arXiv.2103.11318`. URL: `http://arxiv.org/abs/2103.11318`. arXiv:2103.11318 [cs].