DAInfer: Inferring API Aliasing Specifications from Library Documentation via Neurosymbolic Optimization

CHENGPENG WANG, The Hong Kong University of Science and Technology, China
JIPENG ZHANG, The Hong Kong University of Science and Technology, China
RONGXIN WU, School of Informatics, Xiamen University, China
CHARLES ZHANG, The Hong Kong University of Science and Technology, China

Modern software systems heavily rely on various libraries, necessitating understanding API semantics in static analysis. However, summarizing API semantics remains challenging due to complex implementations or the unavailability of library code. This paper presents DAInfer, a novel approach for inferring API aliasing specifications from library documentation. Specifically, we employ Natural Language Processing (NLP) models to interpret informal semantic information provided by the documentation, which enables us to reduce the specification inference to an optimization problem. Furthermore, we propose a new technique called neurosymbolic optimization to efficiently solve the optimization problem, yielding the desired API aliasing specifications. We have implemented DAInfer as a tool and evaluated it upon Java classes from several popular libraries. The results indicate that DAInfer infers the API aliasing specifications with a precision of 79.78% and a recall of 82.29%, averagely consuming 5.35 seconds per class. These obtained aliasing specifications further facilitate alias analysis, revealing 80.05% more alias facts for API return values in 15 Java projects. Additionally, the tool supports taint analysis, identifying 85 more taint flows in 23 Android apps. These results demonstrate the practical value of DAInfer in library-aware static analysis.

CCS Concepts: • **Software and its engineering** → *Software libraries and repositories*; **Automated static analysis**; • **Applied computing** → *Document analysis*.

Additional Key Words and Phrases: specification inference, documentation mining, alias analysis

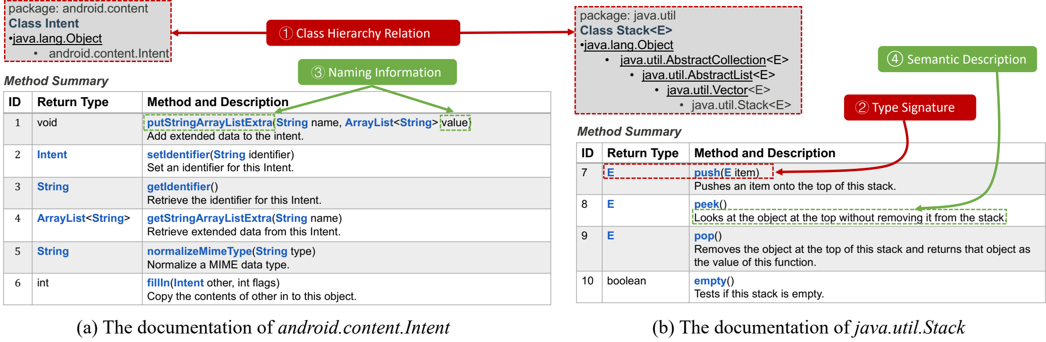
ACM Reference Format:
Chengpeng Wang, Jipeng Zhang, Rongxin Wu, and Charles Zhang. 2024. DAInfer: Inferring API Aliasing Specifications from Library Documentation via Neurosymbolic Optimization. *Proc. ACM Softw. Eng.* 1, FSE, Article 109 (July 2024), 24 pages. https://doi.org/10.1145/3660816

1 INTRODUCTION

In modern programming languages, programmers often develop their applications based on various libraries, which provide fundamental building blocks for client-side implementation. Undoubtedly, the behaviors of library APIs directly affect the functionality of the application code. As targeted by existing studies [8, 22], several library APIs are essentially generalized store and load operations, forming aliasing relations through store-load matches. For example, the APIs HashMap.put and HashMap.get conduct the store and load operations, respectively. When they are invoked upon the same HashMap object with the same first parameters successively, the return value of HashMap.get can be aliased with the second parameter of HashMap.put. To identify value flows in the application

Authors' addresses: Chengpeng Wang, The Hong Kong University of Science and Technology, Hong Kong, China, cwangch@cse.ust.hk; Jipeng Zhang, The Hong Kong University of Science and Technology, Hong Kong, China, jzhanggr@cse.ust.hk; Rongxin Wu, School of Informatics, Xiamen University, Xiamen, China, wurongxin@xmu.edu.cn; Charles Zhang, The Hong Kong University of Science and Technology, Hong Kong, China, charlesz@cse.ust.hk.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2994-970X/2024/7-ART109
https://doi.org/10.1145/3660816



(a) The documentation of *android.content.Intent*   (b) The documentation of *java.util.Stack*

Fig. 1. Examples of library documentation. We use m_i to denote the API with the ID i in the paper.

code, a static analyzer should be aware of such API aliasing specifications, which play critical roles for pointer analysis and other downstream clients. According to our investigation, many existing static analysis techniques rely on manually specified library API aliasing specifications [4, 6, 24]. However, the emergence of third-party libraries introduces a large number of APIs, making laborious effort unacceptable in practice.

This work targets the API aliasing specification inference problem to support library-aware alias analysis. Existing approaches infer API aliasing specifications from three perspectives. The first line analyzes the source code statically [5, 44]. Although it can derive the function summaries as the API aliasing specifications, the solution suffers the scalability problem due to deep call chains [50]. More importantly, the implementation of several library APIs can depend on native code, such as System.arraycopy in the implementation of java.util.Vector, which makes static analysis intractable [8]. The second line of the techniques constructs unit tests via active learning to trigger the execution of library APIs, so as to infer aliasing relations in the runtime [8]. Compared to static analysis-based inference techniques, they are more applicable when the source code of the library is unavailable. However, it can be infeasible to generate unit tests to trigger the target library APIs due to the difficulties of constructing the parameters with complex data structures and executing APIs in specific devices or environments. Third, several researchers learn the aliasing specifications from applications using libraries [22], which does not require the source code of the libraries or the execution of the programs. Unfortunately, their approach only discovers the API specifications used in the applications, finally causing the low recall in the inference.

This paper presents a new perspective on inferring API aliasing specifications. Unlike existing studies, we utilize another important library artifact, namely library documentation, to analyze the semantics of library APIs. As shown in Figure 1, library documentation contains formal semantic properties, e.g., class hierarchy relation and type signatures, and informal semantic information, e.g., semantic descriptions and naming information. Although the library documentation demonstrates the library API semantics in detail, it is far from trivial to derive API aliasing specifications from it. First, effectively understanding the informal semantic information is quite difficult. Even if we apply the recent advance in the large language models (LLMs), e.g., feeding the documentation of android.content.Intent to ChatGPT, we can only obtain nine API aliasing specifications, all of which are incorrect. Second, library documentation can be quite lengthy, which may introduce significant overhead. For example, feeding the lengthy documentation to ChatGPT not only demands much time but also introduces a high financial cost due to enormous token consumption.

To effectively achieve the inference with high efficiency, we propose our inference algorithm named DAInfer, which originates from three key insights:

- The class hierarchy determines the available APIs of a given class, while type signatures enable us to over-approximate aliasing facts based on the types of API parameters and returns. If two values can not be aliased, we do not need to analyze the naming information and semantic descriptions, which decreases the overhead by avoiding applying NLP models.
- The named entities in the names of APIs and parameters indicate the high-level semantics and narrow down aliasing relations between the parameters and return values. In Figure 1(a), the named entities in getIdentifier and the parameter name of Intent.setIdentifier are the same, indicating that the return value of Intent.getIdentifier can be aliased with the parameter of Intent.setIdentifier.
- Semantic descriptions reveal the conducted memory operations with specific verbs, supporting identifying store-load matches that may introduce aliasing facts. In Figure 1(b), "push" and "look" show that Stack.push and Stack.peek conduct the insertion and read operations, respectively.

Based on our insights, we propose DAInfer, an algorithm to infer API aliasing specifications. Technically, we introduce a graph representation to over-approximate the aliasing relations between parameters and return values based on type information. To interpret informal semantic information, we use a LLM and a tagging model to abstract memory operation kinds and high-level semantics of API parameters/return values, respectively. Then, we reduce the specification inference problem to an optimization problem that enforces the aliasing pairs between API parameters as many as possible for precise semantic abstraction. Particularly, the optimization problem poses constraints over the results of the two NLP models. To solve the problem efficiently, we propose the neurosymbolic optimization algorithm, which interacts with the two NLP models in a demand-driven manner, achieving low resource cost in the inference.

We implement our approach DAInfer and evaluate it upon Java classes in several popular libraries. Our evaluation shows that DAInfer achieves the specification inference with a precision of 79.78% and a recall of 82.29%, consuming 5.35 seconds per class on average. We also quantify the impact of the inferred API aliasing specifications on the pointer analysis and taint analysis. It is shown that DAInfer promotes the alias analysis by discovering 80.05% more aliasing facts for the API return values and enables the taint analysis to discover 85 more taint flows in the experimental subjects. Our main contributions of this work include:

- We introduce a new paradigm of inferring API aliasing specifications and reduce the inference problem to an optimization problem over a graph representation of the library documentation.
- We propose a novel technique, namely neurosymbolic optimization, to efficiently solve the optimization problem for the API aliasing specification inference.
- We extensively evaluate our approach over real-world libraries to demonstrate its superiority over existing techniques and quantify its impact on client analyses.

2 BACKGROUND AND OVERVIEW

In this section, we introduce the background of API aliasing specification inference and outline our key idea of inferring API aliasing specifications from documentation.

2.1 Library-Aware Alias Analysis

Modern software systems heavily depend on various libraries. A recent study found that a Java project can include an average of 48 libraries transitively [52]. This prevalence of library usage stimulates the demand for modeling API semantics in fundamental static analyses, such as alias analysis. However, the deep call chains and unavailable source code (e.g., native functions) complicate the scalability and applicability of static analysis. Many static analyzers use specifications to abstract the library API semantics to achieve library-aware analysis. Specifically, the API aliasing

specification for an API pair (m_1, m_2) indicates: When m_1 and m_2 conduct the store and load operations, respectively, the return value of m_2 may be aliased with the parameter of m_1 if m_2 is invoked after m_1 upon the same object. Based on the specification, a static analyzer can model the library API semantics without explicitly analyzing the implementation of m_1 and m_2, ultimately promoting the scalability and applicability of the overall analysis.

Example 1. Figure 1(a) indicates that when the first parameters of Intent.putStringArrayListExtra and Intent.getStringArrayListExtra are aliased, the return value of the latter can be aliased with the second parameter of the former if they are invoked successively upon the same Intent object.

2.2 Different Perspectives of Inferring API Aliasing Specifications

With the increasing number of third-party libraries, manually specifying the API aliasing specifications demands incredibly laborious effort [4, 6, 24]. To mitigate this problem, previous studies infer API aliasing specifications from different artifacts, including library implementation [5], application code using libraries [22], and unit tests constructed via active learning [8]. However, their solutions can be bothered with three main drawbacks. First, analyzing the library implementation suffers the scalability issue due to complex program structures, such as deep call chains, and even becomes inapplicable due to the unavailability of the implementation or the presence of native code. Second, inferring the specifications from application code using libraries may fail to achieve high recall when specific APIs are not utilized in the application code. Third, deriving the aliasing facts from dynamic execution of unit tests suffers the inapplicability issue when it is infeasible to construct executable unit tests in specific devices or environments.

To fill the research gap, our work proposes another perspective to infer the API aliasing specifications. We realize that there is another essential library artifact, i.e., library documentation, demonstrating the library API semantics in a semi-formal structure. As shown in Figure 1, the formal semantic properties, including class hierarchy relation and type signatures, are explicitly provided. Meanwhile, the naming information, e.g., the parameter names and API names, shows the intent of API parameters and return values, while semantic descriptions demonstrate the functionalities of the APIs informally. These ingredients permit us to understand how the library APIs manipulate the memory and further form aliasing relations between their parameters and return values. More importantly, the documentation is often available for analysis, as the developers tend to refer to it during the development. Hence, inferring the API aliasing specifications from documentation would exhibit better applicability than the existing techniques.

2.3 Overview of DAInfer

Although the documentation guides the developers in understanding the API semantics, there exists a gap between the API knowledge and API aliasing specifications. Concretely, we need to understand how the API parameters are stored and how the API return values are loaded. However, achieving this is quite complicated in front of informal semantic information. Even if we leverage the new advances in the LLMs, they cannot understand how the APIs manipulate memory and eventually fail to identify the aliasing relations between API parameters and return values. In the presence of lengthy documentation, a large number of interactions with LLMs can bring huge time overhead due to the time cost of LLM inference and also consume many tokens.

To address the challenges, we propose a novel inference algorithm named DAInfer, which effectively understands the API semantics and efficiently infers the API aliasing specification from library documentation. Our key idea originates from three critical observations on the aliasing relations between the parameters and return values of the library APIs as follows.

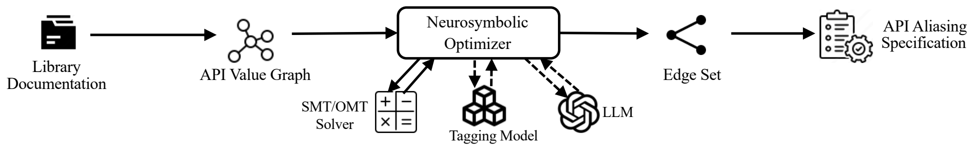


Fig. 2. Workflow of DAInfer

- The parameters and return values should be type-consistent if they are aliased. Specifically, their types should be the same, or one of them is the sub-type/super-type of the other. Such facts can be easily obtained from class hierarchy relation and type signatures in the documentation. In Figure 1, for example, we can obtain the potential aliasing relation between the return value of Intent.getIdentifier and the parameter of Intent.setIdentifier, while the second parameter of Intent.putStringArrayListExtra can not be aliased with the return value of Intent.getIdentifier.
- If the return values and parameters of two APIs are aliased, the named entities in their names tend to be the same, indicating the same high-level semantics. For example, the APIs Intent.setIdentifier and Intent.getIdentifier in Figure 1(a) share the same named entity identifier, indicating that they manipulate the same inner field. For general-purpose data structures, such as java.util.Stack in Figure 1(b), the API names of Stack.peek and Stack.pop do not have any named entities, indicating that their return values can be aliased with other parameters with consistent types.
- If a library API stores its parameters or loads the inner field as the return value, the verbs in its semantic description can reflect the memory operation kind intuitively. For example, the verbs "set" and "insert" are commonly used for the APIs storing their parameters, while the verbs "get" and "return" are prevalent in the semantic descriptions of the APIs loading inner fields.

Based on the observations, we realize that we can leverage type information to over-approximate aliasing relations and utilize named entities and verbs to understand the high-level semantic meanings of the APIs. For any store-load API pair, we can finalize an API aliasing specification as long as we discover the parameters and return values with the same semantic meanings and consistent types. According to these insights, we design our inference algorithm DAInfer, of which the workflow is shown in Figure 2. Our key technical design consists of three components.

- We introduce a new graph representation, namely the *API value graph*, to approximate aliasing relations. After converting a library documentation to a normalized documentation model, we encode the potential aliasing relations in the API value graph.
- We reduce the inference problem to an optimization problem upon the API value graph, where we aim to discover as many aliasing facts among parameters and return values as possible. Particularly, we leverage two NLP models, namely a tagging model and a LLM, to extract the named entities and interpret the semantic descriptions, respectively.
- We instantiate the optimization problem and propose an efficient neurosymbolic optimization algorithm to solve the problem, of which the solution induces the API aliasing specifications. Our neurosymbolic optimization algorithm interacts with the tagging model and the LLM in a demand-driven manner, significantly improving the efficiency of our algorithm.

Benefiting from our insights, our inference algorithm DAInfer simultaneously achieves high precision, recall, and efficiency. The high availability of library documentation also promotes the applicability of our approach in real-world scenarios. In the following sections, we will formulate our problem (§ 3) and provide our technical design (§ 4 and § 5) in detail.

3 PROBLEM FORMULATION

This section first formulates the documentation model (§ 3.1) and then defines the API aliasing specification (§ 3.2). Lastly, we provide the formal statement of the API aliasing specification inference problem and highlight the technical challenges (§ 3.3).

3.1 Documentation Model

Definition 1. (Documentation Model) Given a library, its documentation model is $\mathbf{L} := (\mathbf{H}, \mathbf{T}, \mathbf{N}, \mathbf{D})$:

- Class hierarchy model $\mathbf{H}$ maps a class c to a set of classes, which are the superclasses of c.
- Type signature model $\mathbf{T}$ maps (c, m, i) to a type, where m is an API of the class c and i is the index of the parameter. Without ambiguity, we regard the index of the return value as -1.
- Naming model $\mathbf{N}$ maps (c, m, i) to a string indicating the parameter name or API name, where m is an API of the class c and i is the index of the parameter. Without ambiguity, $\mathbf{N}(c, m, -1)$ indicates the name of the API m of the class c.
- Description model $\mathbf{D}$ maps (c, m) to a string indicating the API semantic description.

Example 2. According to the documentation of the class Intent in Figure 1, we have

$$\mathbf{H}(\text{Intent}) = \{\text{Object}\},\quad \mathbf{T}(\text{Intent}, m_1, -1) = \text{void},\quad \mathbf{T}(\text{Intent}, m_1, 1) = \text{ArrayList<String>}$$

$$\mathbf{N}(\text{Intent}, m_1, 0) = \text{name},\quad \mathbf{N}(\text{Intent}, m_1, 1) = \text{value},\quad \mathbf{N}(\text{Intent}, m_1, -1) = \text{putStringArrayListExtra}$$

$\mathbf{D}(\text{Intent}, m_1)$ is "Add extracted data to the intent". Here, m_1 is the API Intent.putStringArrayListExtra. Due to space limits, we do not discuss other APIs in detail.

Based on documentation, we can collect all the APIs offered by a specific class and its superclasses, forming the universe of available APIs when using the class. The naming information and API semantic descriptions are informal specifications, guiding the developers to use proper APIs in their programming contexts. Based on the documentation model, not only do developers achieve their program logic conveniently, but also analyzers can understand the behavior of each API.

3.2 API Aliasing Specification

To support the library-aware alias analysis, we concentrate on the API aliasing specification inference and follow an important form of aliasing specifications formulated in the prior study [22], which is defined as follows.

Definition 2. (API Aliasing Specification) An API aliasing specification is a tuple (m_1, m_2, P, t), where m_1 and m_2 are two APIs, $P := \{(i_1^{(1)}, i_1^{(2)}), \cdots, (i_j^{(1)}, i_j^{(2)})\}$ is a set of non-negative integer pairs, and t is an non-negative integer. It indicates that the return value of m_2 can be aliased with the t-th parameter of m_1 if

- m_1 is called before m_2 upon the same object
- The $i_k^{(1)}$ and $i_k^{(2)}$-th parameters of m_1 and m_2 are aliased accordingly.

Here, $0 \leq i_k^{(1)} \leq n_1$, $0 \leq i_k^{(2)} \leq n_2$, and $0 \leq k \leq j$. n_1 and n_2 are the parameter numbers of m_1 and m_2, respectively. Without ambiguity, we call m_1 and m_2 form a store-load API pair.

Definition 2 shows that the APIs m_1 and m_2 conduct the store and load operations upon the memory, respectively. Unlike simple load and store operations of pointers, storing and loading the values upon memory may depend on the values of other parameters, which are induced by the set P, determining the memory location where the values are stored and loaded, respectively. Essentially, the set P indicates the pre-condition of the aliasing relation between the return value of m_2 and the t-th parameter of the m_1. If P is empty, the parameters of m_1 and m_2 are not necessarily aliased to enforce the aliasing relation between the return value of m_2 and the t-th parameter of m_1.

Example 3. In Figure 1(a), we have two API aliasing specifications $(m_1, m_4, \{(0, 0)\}, 1)$ and $(m_2, m_3, \emptyset, 0)$. Specifically, the API aliasing specification $(m_1, m_4, \{(0, 0)\}, 1)$ indicates that the return value of Intent.getStringArrayListExtra and the second parameter of Intent.putStringArrayListExtra are aliased when they are invoked upon the same object and their first parameters are aliased.

The API aliasing specification in Definition 2 is more general than the one targeted by USpec [22]. Specifically, USpec only infers that calling m_2 may return a value aliased with the t-th parameter of a preceding call of m_1 on the same object *if all other parameters are aliased*. However, there exist many store-load API pairs in which not all the other parameters are aliased. For instance, the API createBitmap of android.graphics.Bitmap sets the values of DisplayMetrics, Config, width, and height simultaneously, while the API getConfig only fetches the value of Config. Our formulation in Definition 2 is expressive enough to depict such the store-load API pair.

3.3 Problem Statement

We aim to address the API aliasing specification inference problem from another perspective. As demonstrated in § 3.1, the library documentation provides various forms of semantic information of the library APIs. Hence, we can hopefully derive the API aliasing specifications from documentation without conducting deep semantic analysis upon the source code or program runtime information.

The API aliasing specification for a given store-load API pair may not be unique. In Example 3, for instance, $(m_1, m_4, \emptyset, 1)$ is also a valid specification, while it does not pose any restrictions upon the parameters of the two APIs as the pre-condition. In our work, we want to ensure that the inferred specifications exhibit as strong pre-conditions as possible, which implies the maximal size of the set P. Finally, we state the problem of the API aliasing specification inference as follows.

> Given a documentation model $\mathbf{L} = (\mathbf{H}, \mathbf{T}, \mathbf{N}, \mathbf{D})$, infer a set of API aliasing specifications S_{AS} such that $|P|$ is maximized for each $(m_1, m_2, P, t) \in S_{\text{AS}}$.

Technical Challenges. Although library documentation offers semantic information, solving the above problem is quite challenging. First, the naming information and semantic descriptions can be ambiguous. Without an effective interpretation, we can not understand how the APIs operate upon the memory and identify aliasing relations between parameters and return values. Second, there are often many available APIs offered by a single class and even its superclasses. It is non-trivial to obtain high efficiency in front of a large number of available APIs for each class.

Roadmap. In this work, we propose an inference algorithm DAInfer to address the two technical challenges. Specifically, we introduce the documentation model abstraction to formulate semantic information, which enables us to reduce the original problem to an optimization problem (§ 4). Furthermore, we propose the neurosymbolic optimization to efficiently solve the instantiated optimization problem (§ 5). We present the details of our implementation (§ 6) and demonstrate the evaluation quantifying the effectiveness and efficiency of DAInfer (§ 7).

4 DOCUMENTATION MODEL ABSTRACTION

This section presents the abstraction of documentation model. Specifically, we propose the concept of the API value graph (§ 4.1) and introduce two label abstractions over the graph (§ 4.2), which enables us to reduce the API aliasing specification problem to an optimization problem (§ 4.3).

4.1 API Value Graph

As shown in § 3.1, the formal semantic information, namely class hierarchy and the type signatures, reveals potential aliasing relations between API parameters and return values, while the informal semantic information, e.g., naming information and semantic descriptions, shows how parameters and return values are utilized. To depict aliasing relations that can be introduced by API invocations, we propose a graph representation, namely *API value graph*, as follows.

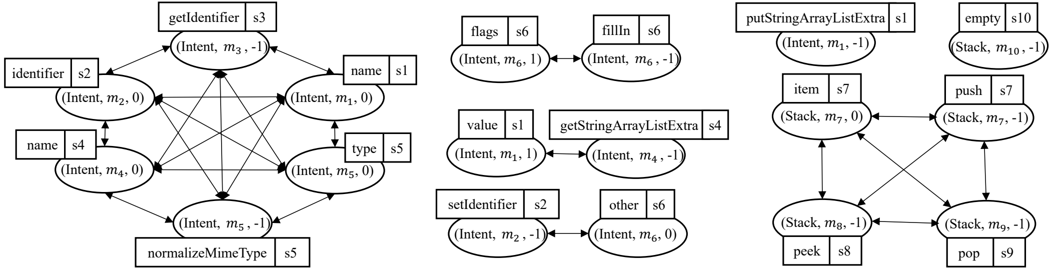


Fig. 3. The API value graph of the documentation model induced by the documentation in Figure 1

Definition 3 (API Value Graph)**.** Given a documentation model $\mathbf{L} = (\mathbf{H}, \mathbf{T}, \mathbf{N}, \mathbf{D})$, its API value graph is the labeled graph $G := (V, E, \ell_n, \ell_d)$, where

- The node set V contains API parameters and return values, which are referred to as *API values.* $(c, m, i) \in V$ if and only if $(c, m, i) \in \text{dom}(\mathbf{N})$ or there is $c' \in \mathbf{H}(c)$ such that $(c', m, i) \in \text{dom}(\mathbf{N})$.
- The edge set $E \subseteq V \times V$ indicates possible aliasing relations between API values. Specifically, $(v_1,\ v_2) \in E$ if and only if $\mathbf{T}(v_1) = \mathbf{T}(v_2)$, $\mathbf{T}(v_1) \in \mathbf{H}(\mathbf{T}(v_2))$, or $\mathbf{T}(v_2) \in \mathbf{H}(\mathbf{T}(v_1))$.
- The name label ℓ_n is a function that maps an API value to its name, i.e., $\ell_n(v) = \mathbf{N}(v)$.
- The description label ℓ_d is a function that maps an API value to the semantic description of the API, i.e., $\ell_d(v) = \mathbf{D}(c, m)$, where $v = (c, m, i)$.

The API value graph regards API values, namely API parameters and return values, as first-class citizens, and depicts their high-level semantics with labels. Intuitively, an edge from (c, m_1, i_1) to (c, m_2, i_2) indicates the fact that the two values may be aliased when m_2 is invoked after m_1 upon the same object. Meanwhile, the two labels attach the informal semantic information to API values, showing their usage intention. From a high-level perspective, the API value graph over-approximates aliasing relations according to class hierarchy relation and type signatures and still preserves informal semantic information as labels to support further specification inference.

Example 4. Figure 3 shows the API value graph for the documentation model induced by the classes in Figure 1, where the name labels and description labels are shown in the left and right boxes, respectively. s_i indicates the semantic description of m_i in Figure 1. Specifically, the edge from $(\text{Intent}, m_2, 0)$ to $(\text{Intent}, m_5, 0)$ indicates that the first parameters of Intent.setIdentifier and Intent.normalizeMimeType may be aliased when the two APIs are invoked successively.

4.2 Label Abstraction

Although the edges of the API value graph over-approximate aliasing relations over API values, not all the aliasing relations can hold when using APIs. In Figure 1, for example, the return value of getIdentifier and the first parameter of normalizeMimeType are unlikely to be aliased as the named entities in their names are different, revealing different usage intention of the two API values. To formulate this key idea, we first introduce the concept of the *semantic unit abstraction* as follows.

Definition 4. (Semantic Unit Abstraction) A semantic unit abstraction α_τ is a function mapping a string s to a set of named entities contained in s. We call the elements in $\alpha_\tau(s)$ as *semantic units.*

Example 5. The named entities in the API name of getStringArrayListExtra include string, array, list, and extra. Hence, we have $\alpha_\tau(\text{getStringArrayListExtra}) = \{\text{string},\ \text{array},\ \text{list},\ \text{extra}\}$.

Essentially, the semantic unit abstraction extracts the named entities from the names as semantic units, which shows the high-level semantics of API values, enabling us to refine aliasing relations

according to the following two intuitions: (1) If two API values v_1 and v_2 have the names with the same semantic units, we can obtain the confidence that they are very likely to indicate the same object in the memory; (2) If the name of an API value does not have any semantic units, we can conservatively regard that it can be aliased with any other API values with consistent types. Hence, we formally define the *semantic unit consistency* to formulate the two intuitions.

Definition 5. (Semantic Unit Consistency) Given a semantic unit abstraction α_τ upon an API value graph $G = (V, E, \ell_n, \ell_d)$, two nodes v_1 and v_2 are semantic-unit consistent if and only if (1) $\alpha_\tau(\ell_n(v_1)) = \alpha_\tau(\ell_n(v_2))$, or (2) $\alpha_\tau(\ell_n(v_1)) = \emptyset \vee \alpha_\tau(\ell_n(v_2)) = \emptyset$.

Example 6. Consider the API value graph in Figure 3. We have $\alpha_\tau(\text{getIdentifier}) = \{\text{identifier}\}$, so the return value of the API getIdentifier and the first parameter of setIdentifier are semantic-unit consistent for the class Intent. Also, we have $\alpha_\tau(\text{item}) = \{\text{item}\}$ and $\alpha_\tau(\text{peek}) = \emptyset$, so the return value of peek and the first parameter of push are semantic-unit consistent for the class Stack.

Lastly, we notice that semantic descriptions show how the API values are manipulated upon the memory. Specifically, we give a formal definition of the concept named *memory operation abstraction* as follows.

Definition 6. (Memory Operation Abstraction) A memory operation abstraction α_o maps a semantic description s to $\alpha_o(s) \subseteq M$, where $M = \{\text{I}, \text{D}, \text{R}, \text{W}\}$. The elements in M indicate the insertion (I), deletion (D), read (R), and write (W) operation upon the memory.

Notably, we classify common memory operations into four categories due to two major reasons. First, the write operation contains several sub-kinds, such as deletion and insertion. If we only categorize memory operations into read and write, we can not distinguish the APIs conducting the deletion and insertion, such as pop and add for java.util.Stack, which may yield wrong API aliasing specifications. For example, the APIs pop and peek of java.util.Stack would be wrongly identified to form a store-load pair and thus induce an incorrect API aliasing specification. Second, objects can be organized in various structural manners. When adding an object to a container-typed field, such as java.util.Stack and java.util.HashMap, the operation is an insertion. When storing an object in a non-container-typed field, the API writes a specific value to the field. The above operations are often described differently in the natural language, so we formulate the memory operation abstraction in a fine-grained manner.

Example 7. According to Figure 1, we have $\alpha_o(s_1) = \{\text{I}, \text{W}\}$, $\alpha_o(s_2) = \{\text{W}\}$ and $\alpha_o(s_3) = \alpha_o(s_4) = \{\text{R}\}$ for Intent. For Stack, we have $\alpha_o(s_7) = \{\text{I}, \text{W}\}$, $\alpha_o(s_8) = \{\text{R}\}$, and $\alpha_o(s_9) = \{\text{R}, \text{D}, \text{W}\}$.

To sum up, the above two label abstractions interpret the informal semantic descriptions with the sets of semantic units and memory operations, based on which we can refine potential aliasing relations indicated by the edges of the API value graph and identify store-load API pairs. In § 5.2, we will demonstrate how to instantiate the two abstractions to support the specification inference.

4.3 Problem Reduction

Based on the two label abstractions, we can interpret the high-level semantics of API values and the memory operations conducted by the APIs. According to our problem statement in § 3.3, we need to identify the store-load API pairs and find as many aliased parameters as possible, which determine a strong pre-condition of the aliasing relation between loaded and stored values. Hence, we reduce the specification inference to an optimization problem over the API value graph as follows.

Definition 7. (Optimization Problem) Given a semantic unit abstraction α_τ and a memory operation abstraction α_o upon an API value graph $G = (V, E, \ell_n, \ell_d)$, find an edge set $E^* \subseteq E$ with a maximal size $|E^*|$ satisfying the following constraints:

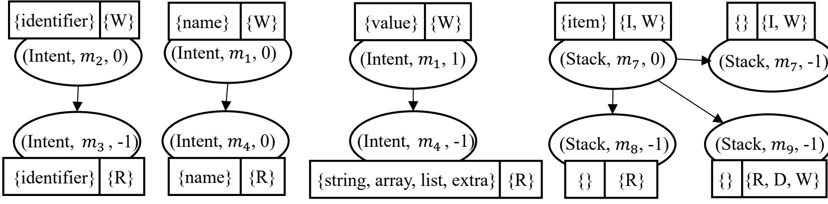


Fig. 4. An optimal solution to the problem instance over the API value graph shown in Figure 3

- (Degree constraint) For each $v \in V$, the in-degree and out-degree of v are not greater than 1.
- (Validity constraint) If $(v_1, v_2) \in E^*$, where v_1 and v_2 indicate parameters, there exist $u_1,\ u_2 \in V$ such that $(u_1, u_2) \in E^*$. where u_1 and u_2 indicate a parameter and a return value, respectively.
- (Semantic unit constraint) For any $(v_1, v_2) \in E^*$, where $v_1 = (c, m_1, i_1)$ and $v_2 = (c, m_2, i_2)$, the semantic unit abstraction of the names of v_1 and v_2 should satisfy
  - (S1) If $i_2 \neq -1$, v_1 and v_2 are semantic-unit consistent.
  - (S2) If $i_2 = -1$, v_1 or v_1' is semantic-unit consistent with v_2, where $v_1' = (c, m_1, -1)$.
- (Memory operation constraint) For any $(v_1, v_2) \in E^*$, the following two conditions are satisfied:
  - (M1) v_1 satisfies $\mathsf{I} \in \alpha_o(\ell_d(v_1)) \vee (\mathsf{W} \in \alpha_o(\ell_d(v_1)) \wedge \mathsf{D} \notin \alpha_o(\ell_d(v_1))$
  - (M2) v_2 satisfies that $\mathsf{R} \in \alpha_o(\ell_d(v_2))$

Definition 7 aims to maximize $|E^*|$ to discover all the aliased parameters of each store-load API pair, which corresponds to maximizing $|P|$ in original problem statement in § 3.3. The four kinds of constraints are posed upon the selected edges. Specifically, the degree and validity constraints ensure that the edges induce the API aliasing specification defined Definition 2. Besides, the parameters of the APIs m_1 and m_2 should be semantic-unit consistent if they are connected by a selected edge (S1). If a selected edge connects the parameter of m_1 and the return value of m_2, then the parameter of m_1 should be semantic-unit consistent with the return value of m_2 (S2). Lastly, the memory operation constraint ensures that the APIs m_1 and m_2 should form a store-load API pair (M1 and M2). Finally, we can obtain the specifications based on the optimal solution as follows.

Given the optimal solution E^* of the optimization problem defined in Definition 7, we can obtain the API aliasing specification $(m_1, m_2, P, t) \in S_{\mathsf{AS}}$, where

- $P = \{(i_1, i_2) \mid ((c, m_1, i_1),\ (c, m_2, i_2)) \in E^*, i_2 \neq -1\}$
- t satisfies $((c, m_1, t), (c, m_2, -1)) \in E^*$

Example 8. Figure 4 shows the optimal solution to the optimization problem over the API value graph in Figure 3, where the sets shown in the two boxes demonstrate the extracted semantic units and the identified memory operations under the label abstractions in Examples 6 and 7. We discover six possible aliasing relations. Notably, although the semantic units of $(\mathsf{Intent}, m_4, -1)$ are different from $(\mathsf{Intent}, m_1, 1)$, they are exactly the same as the ones of $(\mathsf{Intent}, m_1, -1)$, indicating that the second parameter of m_1 can have the same semantics as the return value of m_4. The optimal solution finally induces the API aliasing specifications in Example 3.

By reducing the original problem to the optimization problem in Definition 7, we only need to tackle two sub-problems for the specification inference. First, we have to instantiate two label abstractions so that we can precisely interpret the semantic meanings of names and the memory operation kinds. Second, we need to design an efficient optimization algorithm to solve the optimization problem. In § 5, we will provide the technical details of addressing the two sub-problems.

5 SPECIFICATION INFERENCE VIA NEUROSYMBOLIC OPTIMIZATION

This section presents the technical details of our algorithm DAInfer. Specifically, we demonstrate the overall algorithm in § 5.1 and detail the label abstraction instantiation in § 5.2. Besides, we present the neurosymbolic optimization in § 5.3 to instantiate and solve the optimization problem given in Definition 7. Lastly, we summarize our approach and highlight its advantages in § 5.4.

5.1 Overall Algorithm

Algorithm 1: Inference Algorithm

Input: $\mathbf{L}$: Documentation model;
Output: S_{AS}: API aliasing specifications;

```
1 G ← constructAVG(L);
2 α_τ ← getSemanticUnitAbs();
3 α_o ← getMemoryOperationAbs();
4 𝒫 ← (L, G, α_τ, α_o);
5 E* ← neuroSymOpt(𝒫);
6 S_AS ← convert(E*);
7 return S_AS;
```

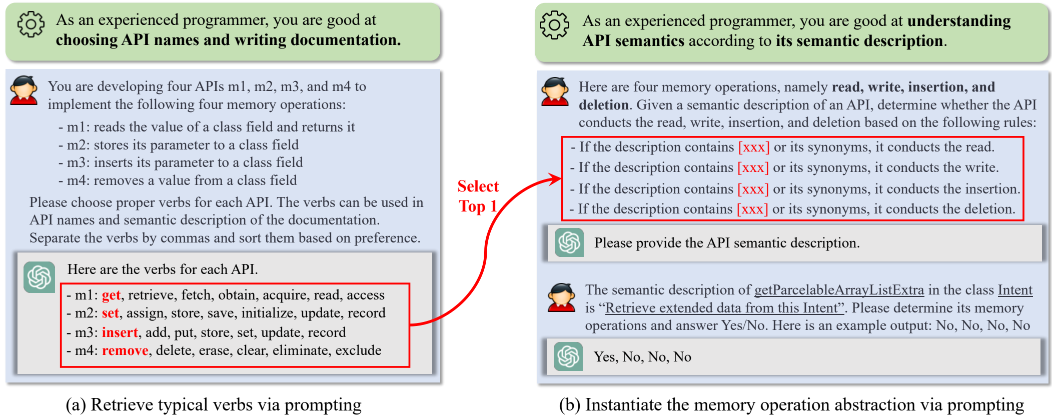
As demonstrated in § 4.3, we can reduce the API aliasing specification inference problem to an instance of the optimization problem given in Definition 7. Technically, we propose and formulate our specification algorithm in Algorithm 1, which takes as input a documentation model $\mathbf{L}$ and generates a set of API aliasing specifications S_{AS} as output. First, we derive the API value graph G from the documentation model $\mathbf{L}$ based on Definition 3 (Line 1). Second, we instantiate two label abstractions, i.e., α_τ and α_o, and further construct an instance of the optimization problem $\mathcal{P}$ defined in Definition 7 (Lines 2–3). Third, we propose the neurosymbolic optimization to solve the instance of the optimization problem $\mathcal{P}$ (Lines 4–5), and finally convert the optimal solution E^* to a set of API aliasing specifications S_{AS} (Line 6). Particularly, Definition 3 has demonstrated how to construct the API value graph, and converting the optimal solution to the specification is also explicitly formulated at the end of § 4.3. In the rest of this section, we will provide more details on the label abstraction instantiation (§ 5.2) and the neurosymbolic optimization algorithm (§ 5.3), which finalize the functions getSemanticUnitAbs, getMemoryOperationAbs, and neuroSymOpt in Algorithm 1, respectively.

5.2 Label Abstraction Instantiation

According to Definitions 4 and 6, the semantic unit abstraction requires attaching the grammatical tags, while the memory operation abstraction demands identifying how an API manipulates memory. In what follows, we will detail how to instantiate them with two different NLP models, respectively.

5.2.1 Instantiating Semantic Unit Abstraction. According to common programming practices, the developers of libraries tend to follow typical naming conventions [13], such as camel case, pascal case, and snake case. For example, userAccount is a parameter name using camel case, and get_account_balance is an API name using snake case. Notably, the sub-words are often separated with an underscore or begin with an uppercase letter. Hence, we can easily decompose each name s into the concatenation of several sub-words and further determine the tag of each sub-word.

However, the names of APIs or their parameters can hardly be valid phrases or sentences. Simply applying the part-of-speech (POS) tagging would tag almost all the words as the nouns. Also, the POS tagging targets tagging sentences, while the names of parameters and APIs are only the concatenation of words in phrases. To obtain more precise tagging results, we leverage an existing probability model trained in Brown Corpus [26], which can return all the possible grammatical tags of each word along with the occurrences. This enables us to determine whether a word is more likely to be a noun according to the existing probability model, which does not depend on the usage context of the word. Formally, we instantiate the semantic unit abstraction as follows.



(a) Retrieve typical verbs via prompting

(b) Instantiate the memory operation abstraction via prompting

Fig. 5. The prompt templates of two-staged prompting

Definition 8. (Instantiation of Semantic Unit Abstraction) Assume that g_τ maps a word w to a set of tag-occurrence pairs $\{(\tau_j, k_j)\}$. Given a sub-word w in a parameter/API name s, $w \in \alpha_\tau(s)$ if and only if $(\text{NOUN}, k^*) \in g_\tau(w)$ and k^* is the largest occurrence in $g_\tau(w)$.

Example 9. Consider the API setIdentifier in Figure 1. After splitting the API name into two sub-words, namely "set" and "identifier", we discover that "set" is more likely to be a verb than a noun, while "identifier" is very likely to be a noun. Hence, our instantiated semantic unit abstraction α_τ maps setIdentifier to {identifier}, identifying identifier as the semantic unit of the API.

5.2.2 Instantiating Memory Operation Abstraction. To instantiate an effective memory operation abstraction, we leverage an important programming practice: The developers often summarize the API functionality in a full sentence or a verb-object phrase as its semantic description. Particularly, the verbs in the semantic description intuitively depict the memory operations conducted by the API. Therefore, it is possible to instantiate an effective memory operation abstraction based on the verbs in the semantic description. However, the verbs used in the semantic descriptions can vary a lot, even if the APIs conduct the same kind of memory operation. For example, when describing an API conducting the memory insertion, developers can choose different verbs, e.g., "put", "insert", and "push". The diverse choices of the verbs describing a specific memory operation would make the inference suffer low recall if we just adopted a grep-like approach based on string matching.

Inspired by recent progress in the NLP community, we realize that the latest advances in the LLMs may provide new opportunities for resolving this issue [12, 37, 39]. Specifically, the LLMs have excellent abilities in text understanding, especially under the guidance of few-shot examples or descriptions of rules. Hence, we instantiate the memory operation abstraction via two-stage prompting, of which the prompt template is shown in Figure 5.

- First, we design the prompt in Figure 5 (a) to retrieve the verbs describing each memory operation and enforce the LLMs sort them based on the preference. Although the verb lists may overlap, the top-1 verbs are representative enough to distinguish different memory operations.
- Second, we select the top-1 verbs recommended in the first stage and then construct the prompt describing the rules for the memory operation abstraction, which is shown in Figure 5(b). Finally, we obtain an LLM response containing four "Yes"/"No" separated by commas.

It is worth noting that we identify memory operation kinds via a two-stage prompting instead of a one-stage prompting. If we manually specify the typical verbs describing memory operations, the second prompt may rely on our manual setting, which demands expert knowledge. If we do not offer typical words as hints, the result is not as interpretable as the current one. Our design actually

utilizes the ability of LLMs to predict method names for coding tasks, self-promoting the memory operation identification with generated typical verbs. Note that the first stage is only conducted once. The typical verbs are shared when analyzing library APIs. Hence, the extra cost introduced by the first stage is negligible. Based on the above prompting process, we can obtain an instantiation of the memory operation abstraction, which is formally formulated as follows.

Definition 9. (Instantiation of Memory Operation Abstraction) g_o is the function induced by the LLM via two-staged prompting in Figure 5. Then the memory operation abstraction α_o satisfies that op $\in \alpha_0(s)$ if and only if the corresponding answer of op in $g_o(s)$ is "Yes", where op $\in M$.

Example 10. In Figure 5(b), the output of the LLM is "Yes, No, No, No", indicating that Intent.getStringArrayListExtra only conducts the memory read. Hence, we have $\alpha_o(s_4) = \{\mathsf{R}\}$, where s_4 is the semantic description of the API Intent.getStringArrayListExtra. Similarly, for the API Intent.normalizeMineType, the verb "normalize" in its semantic description s_5 is not the synonym of four typical verbs, so $\alpha_o(s_5) = \emptyset$, indicating that it does not contribute to any load-store match.

Notably, our intuition of the label abstraction upon the API value graph is applicable for general libraries in real-world production. Typically, the developers of libraries are often in well-organized communities and cooperations, following good naming conventions and using proper verbs in semantic descriptions. That is, they are unlikely to use different nouns to indicate the objects with the same usage intention or describe memory operations conducted by the APIs with wrong verbs. Their good development habits permit us to correctly interpret the informal semantic properties of library APIs with the tagging model and the LLM, which can yield satisfactory precision and recall in the wild. Our evaluation also demonstrates the effectiveness of the label abstraction upon benchmarks used in existing studies [6, 8, 22]. Furthermore, such well-structured natural language descriptions, including documentation and comments, have been utilized in various software engineering tasks, such as API misuse detection [43, 61] and unit test generation [11]. These approaches, which share similar assumptions about natural language descriptions as ours, have demonstrated their practical impacts in understanding code semantics and benefiting downstream clients. We will provide a detailed discussion of these approaches in § 8.

5.3 Neurosymbolic Optimization

As shown in § 5.2, our two label abstractions are achieved with different overheads. Specifically, the semantic unit abstraction only relies on the tagging model that can be applied efficiently, while the memory operation abstraction has to relies on the LLM inference, which is considered to be much more costly than tagging. To achieve high efficiency, we propose a solving technique, named *neurosymbolic optimization*, for the optimization problem defined in Definition 7. For each API pair, we first check the satisfiability of degree constraint ϕ_d and the validity constraint ϕ_v (Lines 2–5). If both of them are satisfied, we apply the tagging model to derive the semantic unit constraint ϕ_s (Line 6) and examine the satisfiability of the conjunction of the three constraints (Line 7). If it is satisfiable, we apply the LLM to achieve the memory operation abstraction,

Algorithm 2: Neurosymbolic optimization

```
   Input: 𝒫: An optimization problem;
   Output: E*: The optimal solution;
 1 foreach (c, m1), (c, m2) do
 2     φd ← deriveDegreeConstraints(𝒫);
 3     φv ← deriveValidityConstraints(𝒫);
 4     if SMTSolve (φd ∧ φv)=UNSAT then
 5         continue;
 6     φs ← deriveSUConstraints(𝒫);
 7     if SMTSolve (φd ∧ φv ∧ φs)=UNSAT then
 8         continue;
 9     φo ← deriveMOConstraints(𝒫);
10     E′ ← Solve(obj(𝒫), φd ∧ φv ∧ φs ∧ φ0);
11     E* ← E* ∪ E′;
12 return E*;
```

and derive the memory operation constraint (Line 9). Based on OMT solving [10], we select the maximal number of edges connecting the API values (Line 10) and append them to the set E^* (Line 11), which is returned as the solution to the optimization problem.

Notably, the degree constraint and validity constraint do not depend on any NLP models and are instantiated *symbolically*, while the semantic unit constraint and memory operation constraint rely on the outputs of the tagging model and the LLM, respectively, being instantiated in a *neural* manner. By decoupling the *symbolic constraints* from *neural ones*, DAInfer applies NLP models with a lazy strategy. Noting that the LLM inference consumes much more time than SMT solving, our design can significantly reduce the time overhead and token cost.

Example 11. Consider the APIs of Intent in Figure 1(a). When processing the APIs Intent.fillIn and Intent.getIdentifier, the validity constraint is not satisfied as there are no type-consistent parameters or return values. Hence, we do not apply the tagging model or the LLM. For the APIs Intent.setIdentifier and Intent.normalizeMimeType, we find that their parameters and return values are not semantic-unit consistent, so we do not invoke the LLM with their semantic descriptions.

When we designed the label abstraction instantiation, we also considered directly prompting LLMs to validate semantic unit consistency. However, pairwise examining the names of an API and its parameters introduces a large number of LLM inferences, which increases time and token costs. We add more discussions on the possibility of utilizing LLMs to improve DAInfer in § 7.5.

5.4 Summary

DAInfer is the first trial of inferring API aliasing specifications from documentation. It demonstrates the promising potential of utilizing new advances in the community of natural language processing, especially the LLMs, to solve traditional static analysis problems. Similar to traditional pointer analyses upon source code, such as Andersen-style pointer analysis [3], DAInfer establishes a constraint system over library documentation to pose restrictions upon pointer facts. In order to precisely understand the natural language, it utilizes the NLP models as documentation interpreters to abstract informal semantic information, which supports instantiating an optimization problem for the specification inference. Our insight into utilizing NLP models for documentation interpretation can be generalized in other tasks, such as program synthesis [58] and test case generation [35].

6 IMPLEMENTATION

We implement the approach DAInfer as a prototype and release the source code online [20]. Specifically, we implement the documentation parser by using *soup* Python package. For each documentation page describing the API semantics, we can extract the four kinds of information, including class hierarchy relation, API type information, naming information, and API semantic descriptions, Since library documentation pages almost have a uniform format, we do not have to make major changes to the implementation of the parser to adapt to different libraries. To instantiate the semantic unit abstraction, we utilize the conditional frequency distributions tool with Brown Corpus provided by Natural Language Toolkit [36] to determine whether or not a word is the most likely to be a noun. To instantiate the memory operation abstraction, we adopt gpt-3.5-turbo model with chat completions API to interpret the API semantic descriptions [38]. Specifically, we invoke the interface ChatCompletion.create to feed the constructed prompts to the LLM and fetch its response. In our implementation, we set the temperatures of the two stages of prompting to 0.7 by default.

We implement the neurosymbolic optimization based on Z3 solver [10, 21]. For any pair of APIs, we introduce $(n_1 + 1) \cdot (n_2 + 1)$ boolean variables to indicate whether the two API values are aliased, where n_1 and n_2 are the numbers of the API parameters. We directly encode the degree constraint

and validity constraint symbolically, while the semantic unit constraint and memory operation constraint are constructed and solved on demand, relying on the outputs of the tagging model and the LLM. We count the number of boolean variables that are assigned with True and set it as the objective function. For better preformance, we parallelize the invocations of the LLM in eight threads, and introduce the memorization technique to store the tagging result and the response of the LLM upon each semantic description. If a word or an API semantic description has been processed before, we directly reuse the previous result.

7 EVALUATION

We evaluate DAInfer by investigating the following research questions:

- **RQ1:** How effectively and efficiently does DAInfer infer API aliasing specifications?
- **RQ2:** How does DAInfer benefit library-aware static analysis clients?
- **RQ3:** How does DAInfer compare against other approaches?

7.1 Experimental Setup

All the experiments are performed on a 64-bit machine with 40 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20 GHz and 512 GB of physical memory. We invoke the Z3 SMT solver with its default options.

Subjects. To show the superiority of DAInfer, we evaluate Atlas [8], USpec [22], and DAInfer upon the same set of Java classes. Specifically, the Java classes are collected from: (1) The classes of which the specifications are manually specified in FlowDroid [6]; (2) The classes appearing in the inference results of USpec [22]. Since the dataset of Atlas [8] is not publicly available, we cannot conduct experiments on it. In total, our benchmark contains 167 Java classes offering 8,342 APIs, which range from general-purpose libraries, including Android framework and Java Collections Framework, to specific-usage libraries, such as Gson. Without ambiguity, we call the first and the second kinds of the classes from FlowDroid benchmark and USpec benchmark, respectively.

7.2 Effectiveness and Efficiency

Effectiveness. Although USpec offers the raw data and the source code of Atlas is available, the ground truth used in the two previous studies is not published. Also, the specifications offered by FlowDroid are manually specified by the developers, and thus, may contain several flaws and miss several correct ones. Hence, we have to label the specifications of the benchmarks manually. Meanwhile, investigating all the classes demands tremendous manual effort. Following the recent study [22], we randomly select 60 classes that offer 2,771 APIs in total. For each API, we examine whether it forms store-load API pairs with other APIs offered by the same class, of which the number can reach 50 on average. To make the manual examination more reliable, we invite five experienced engineers from the industry as volunteers to specify the specifications independently. Specifically, they refer to the specifications specified by the developers of FlowDroid and inferred by existing works (i.e., USpec and Atlas), and meanwhile, investigate the library documentation and implementation simultaneously. In the end, we merge the specifications specified by the five volunteers and resolve the inconsistent parts following the principle of max voting, eventually obtaining 988 API aliasing specifications as the ground truth.

According to our investigation, we find that DAInfer achieves high precision and recall upon the experimental subjects. In total, it successfully infers 2,680 API aliasing specifications. For the randomly selected 60 classes, DAInfer infers 1,019 API aliasing specifications, 813 of which are correct, achieving a precision of 79.78%. After examining all the APIs of the selected classes, we discover that DAInfer misses 175 specifications, achieving a recall of 82.29%. Interestingly, we collect the specifications where the API names contain “get” or “set”, and discover that such specifications

Table 1. Efficiency of DAInfer and its ablations

Tool	# Tagging	# LLM	Token Cost	Time Cost (sec)
DAInfer	32,325	2,950	726,425	892.93
DAInfer-Type	32,325	5,164	1,276,254	1,734.63
DAInfer-Exhausive	58,846	8,090	1,994,017	2,844.26

only take up 33.49% of all the inferred ones. It shows that DAInfer can understand how APIs operate upon the memory even if diverse verbs are used. We also compare our results with the specifications in the FlowDroid and USpec benchmarks. It is shown that DAInfer infers 170 out of the total 210 specifications in FlowDroid benchmark and 65 out of the total 82 specifications inferred by USpec, achieving 81.0% and 79.3% recall upon the two benchmarks, respectively. The above results show that DAInfer can effectively infer the API aliasing specifications from documentation.

Efficiency. We quantify the efficiency of DAInfer with four metrics, including the number of applying the tagging model, the number of applying the LLM, the token cost, and the time cost. As shown in Table 1. DAInfer applies the tagging model 32,325 times and interacts with the LLM 2,950 times using 726,425 tokens, and the overall time cost is 892.93 seconds (around 15 minutes). According to the billing strategy of OpenAI, we only need to pay 1.09 USD in total. We also conduct the ablation study to demonstrate the benefit of the neurosymbolic optimization algorithm. Specifically, the ablation DAInfer-Exhausive applies the two NLP models to all the APIs while the ablation DAInfer-Type applies the NLP models to the APIs satisfying the degree constraint and the validity constraint. As shown in Table 1, DAInfer-Type invokes the LLM 5,164 times with 1,276,254 tokens in total and finishes analyzing all the subjects in 1,734.63 seconds. Besides, DAInfer-Exhausive has to apply the tagging models 58,846 times and invoke the LLM 8,090 times using 1,994,017 tokens, of which the whole process finishes in 2,844.26 seconds. The key reason for the differences between the ablations is that the solving steps at Lines 4 and 7 in Algorithm 2 can effectively reduce the numbers of applying the tagging model and the LLM, respectively, when the conjunctions of the constraints are unsatisfiable. Compared to DAInfer-Type and DAInfer-Exhausive, DAInfer achieves the inference with 1.94× and 3.19× speed-ups. Hence, our neurosymbolic optimization can efficiently support the specification inference.

7.3 Effects on Client Analysis

Following existing studies [8, 22], we choose alias analysis and taint analysis as two fundamental clients of DAInfer to quantify its effects.

Effect on Alias Analysis. We conduct the field and context-sensitive alias analysis by running a static analyzer Pinpoint [46, 55] upon 15 Java projects in two settings. In the setting Alias-Empty, we provide empty specifications of library APIs, i.e., discarding all the possible alias facts introduced by library API calls. In the setting Alias-Infer, we apply the inferred correct API aliasing specifications to the pointer analysis. For each given pointer, Pinpoint computes its alias facts in a sound manner. We quantify the alias set sizes of the return values of library APIs and compute $\frac{size_{\text{infer}}}{size_{\text{empty}}}$ for each library API invocation, where $\text{size}_{\text{infer}}$ and $\text{size}_{\text{empty}}$ are the alias set sizes of the return value under the settings Alias-Infer and Alias-Empty, respectively. Figure 6 is the histogram showing the distribution of the ratios of alias set sizes. According to the ratios of alias set sizes, we can discover that the average increase ratio reaches 80.05% with the benefit

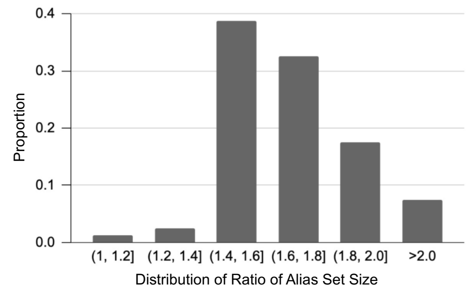


Fig. 6. The results of pointer analysis

of our inferred specifications. Except for the intervals (1, 1.2] and (1.2, 1.4], the size increase ratio is larger than 40% as the ratio is larger than 1.4. The proportion of such library API invocations reaches 96.25%. Because our pointer analysis is sound and we investigate the same set of return values of library API calls, the increases in the alias set sizes demonstrate that DAInfer promotes the alias analysis in discovering more alias facts in the applications using libraries.

Effect on Taint Analysis. We choose three different settings of specifications for FlowDroid to conduct the taint analysis, namely Taint-Empty, Taint-Manual, and Taint-Infer. Here, Taint-Empty and Taint-Infer are similar to the two settings in the pointer analysis, and the sources and sinks are specified based on the default taint specification offered by FlowDroid. Under the setting Taint-Manual, we apply the manual specifications provided by FlowDroid directly. We select 23 popular Android applications in F-Droid [23], which cover different program domains, including navigation, security, and messaging applications. Figure 7 shows the numbers of the taint flows discovered under the three settings. Specifically, FlowDroid discovers 225 taint flows under Taint-Empty, while it finds 304 taint flows under Taint-Manual. Notably, 79 out of 304 taint flows are induced by the aliasing relations among API parameters and returns. When we run FlowDroid under Taint-Infer, it discovers 310 taint flows, 85 of which are discovered based on the correct API aliasing specifications inferred by DAInfer. There are six taint flows in three apps not discovered by FlowDroid under the setting Taint-Infer due to false negatives of our inference algorithm. However, 12 taint flows discovered under Taint-Infer are not discovered under Taint-Manual. The results demonstrate that DAInfer promotes the taint analysis in discovering more taint flows. We do not seek the confirmations of taint flows, which may depend on the developers' subjective intentions and the choices of taint specifications. However, the ability to discover more taint flows has shown the practical impact of DAInfer in detecting potential taint-style vulnerabilities. This evaluation principle is also applied in many existing studies [8, 22, 48].

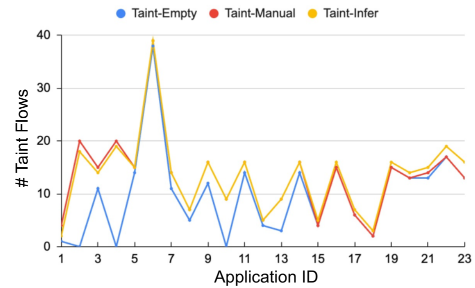


Fig. 7. The results of taint analysis

7.4 Comparison with Existing Techniques

We compare DAInfer with two most recent studies on API aliasing specification inference, i.e., Atlas [8] and USpec [22]. Besides, we construct another baseline, LLM-Alias, which feeds the documentation to ChatGPT and generates API aliasing specifications via in-context learning.

Comparison with Atlas. We run Atlas [7] upon the total 167 classes and finish the inference in 74.48 minutes. Note that the output of Atlas is the library implementation derived from the execution of unit tests. Automatically converting it into the specifications defined in Definition 2 requires static analysis techniques. Hence, we analyze the library implementation generated by Atlas with a field-sensitive pointer analysis, which matches the store-load operations upon the same fields, and eventually convert the output of Atlas to the API aliasing specifications defined in Definition 2. For the classes labeled with ground truth in § 7.2, Atlas infers 546 specifications and 454 correct ones, achieving 83.15% precision and 45.95% recall. After investigating the results, we find that Atlas fails to generate the specifications for 111 classes in the experimental subjects, such as android.os.Intent and android.os.Configuration. The root cause is that Atlas fails to infer the specifications when the creation of library function parameters is non-trivial, or the unit test execution demands a specific environment, such as an Android emulator. In contrast, DAInfer can derive the API aliasing specifications for such classes. Also, the aliasing specifications generated

by DAInfer only depict the potential aliasing relations between parameters and return values, while they all miss the pre-conditions under which such aliasing relations hold. For example, Atlas only obtains that the return value of HashMap.get can be aliased with the second parameter of HashMap.put, missing the pre-condition over their first parameters. The restrictive templates used in the inference introduce the imprecision, which is also reported in the prior study [22].

Comparison with USpec. USpec is not open-sourced due to its commercial use [22]. To make the comparison, we asked the authors for the raw data of their evaluation. According to their results, USpecs successfully obtains 124 API aliasing specifications upon 62 classes. Unfortunately, the precision of USpec only reaches 66.1% (82/124). For instance, USpec generates the incorrect aliasing specification (HashMap.put, HashMap.get, $\{(0, 1)\}$, 0) for the class java.util.HashMap. The root cause is that USpec infers possible aliasing relations according to the usage events, while the keys and values of HashMap objects may have the same types, making the inference algorithm unable to distinguish them with usage events only. However, DAInfer successfully infers the specification via neurosymbolic optimization. We also quantify the recall of USpec based on our labeled specifications in § 7.2. It is shown that USpec misses 370 API aliasing specifications. The recall of inferring API aliasing specifications is only 18.14%. The root cause of its low recall is that USpec can only generate the aliasing specifications for the APIs that are used in the applications.

Comparison with LLM-Alias. We compare DAInfer with LLM-Alias, which directly queries ChatGPT with the documentation. The response of ChatGPT is a natural language sentence indicating an API aliasing specification. Due to laborious effort, we only examine the inference results for 60 classes that we randomly selected in § 7.2. The results show that LLM-Alias generates 801 API aliasing specifications for examined classes, only 113 of which are correct, yielding a precision of 14.11% and a recall of 11.44%. Among 688 incorrect specifications, 60 specifications indicate the correct aliasing relations between parameters and return values, while they do not pose any restrictions over API parameters as the pre-conditions. The results show that vanilla LLMs without special designs have poor performance in understanding the concept of aliasing relation. In contrast, DAInfer achieves quite satisfactory precision and recall, which benefits from our insightful problem reduction and efficient neurosymbolic optimization.

Comparison upon Client Analyses. We also compare the effects of baselines on client analyses with the same settings as the ones in § 7.3. Specifically, Atlas introduces 43.26% increase of the alias set sizes on average, which is lower than the one introduced by DAInfer. USpec and LLM-Alias introduce 14.52% and 12.17% of increase in the alias set sizes on averagely, respectively. Although LLM-Alias infers slightly more API aliasing specifications than USpec, the specifications inferred by USpec contribute more to the aliasing facts, which might be caused by more frequent usage of the involved library APIs in the application code. DAInfer can introduce the highest average increase ratio of the alias sets among different approaches. Similarly, we find that Atlas, USpec, and LLM-Alias discover fewer taint flows than DAInfer, which is shown by Figure 8. Specifically, DAInfer newly discovers 85 taint flows, while Atlas, USpec, and LLM-Alias detect 60, 29, and 35 taint flows in total, respectively. Therefore, DAInfer has overwhelming superiority over existing techniques in assisting client analyses, including alias analysis and taint analysis.

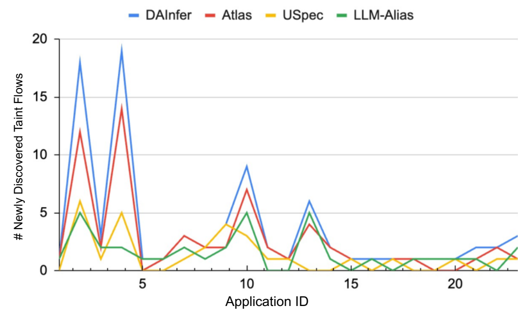


Fig. 8. The results of taint analysis assisted with Atlas, USpec, and LLM-Alias

7.5 Limitations and Future Work

Our approach has several drawbacks that demand further improvements. First, DAInfer can not determine whether an API creates a new object. When the developers create any new objects, our inferred specifications can only depict data flow facts instead of aliasing relations. For example, DAInfer infers an API aliasing specification for java.util.Map that the return value of Map.computeIfPresent can be aliased with the second parameter of Map.put when their first parameters are aliased. This is a wrong specification as computeIfPresent returns null value or a newly computed value instead of any existing values stored in the fields. Second, the semantic unit consistency requires two strings to be equal. In our evaluation, however, we notice that several semantic units are not the same strings while they indicate the same concept in several rare cases. For example, the first parameters of SparseArray.set and SparseArray.valueAt in the class android.util.SparseArray are key and index, respectively. The two different strings are actually the indicators of the same semantic concept. Hence, DAInfer can not infer the correct specification for the two APIs. Although there are several traditional ways to extract synonyms for natural languages, such as WordNet [34] and word embedding [59], they may fail to identify similar semantic units in the programming languages, for example, the similarity between key and index is measured to be even smaller than 0.1 by WordNet. Even if we utilize several code models, such as code2vec [1] and CodeBERT [25], they can still lead a false negtive/positive when the similarity of the names in a correct/wrong specification is below/above the preset threshold.

To further improve DAInfer, we can explore several directions in the future. First, we can leverage domain-specific LLMs for code, which allow local deployment, to validate the semantic unit consistency. If the inference of general-purpose LLMs, such as GPT-4, becomes much more efficient and cheaper in the future, we can also prompt them directly without introducing significant overhead. The above models can hopefully support us in identifying the semantic units indicating the same concept even if they are not the same string. Second, it is promising to obtain a domain-specific LLM via fine-tuning. Specifically, we can leverage existing static analyzers to scan the source code of open-source libraries, obtain their memory operation kinds, and, particularly, determine whether the APIs create objects, which can improve the accuracy of the memory operation abstraction. Third, our current implementation of DAInfer requires the documentation parsing with a manually specified parser. If the LLMs exhibits an enhanced ability of understanding documentation with lower cost, we can derive the documentation model with LLMs via prompting, which further automates the whole process of API aliasing specification inference.

8 RELATED WORK

Library Specification Inference. The inference of library function specifications has always been a central topic in program analysis. Typically, IFDS/IDE-based approaches summarize the data-flow facts of libraries as their semantic abstractions [5, 44], which can be reused across various clients of data-flow analysis. Established upon a symbolic memory model, shape analysis computes the memory state for each statement of a library function as invariants, and derives the pre/post conditions of each library function as its specification [14, 30, 45]. While the inferred specification accurately depicts the semantics of the library function, the analysis suffers from scalability problems, especially in the presence of complex program structures [15]. Mining-based approaches leverage the program facts derived from applications to infer specific forms of specifications, e.g., points-to [8], aliasing [22], taint [17], and commutativity specifications [27], which support specific static analysis clients, e.g., taint analysis [6] and Andersen-style pointer analysis [24]. Another mining-based approach AutoISES automatically infers security specifications from high-quality application code and then guides the detection of security policy violations [49]. Our work concentrates on the aliasing specification inference, which shares the same motivation with the existing

studies [8, 22]. Unlike the previous studies, DAInfer provides a new paradigm of inferring aliasing specifications, which leverages documentation to promote its potential value in different scenarios.

Natural Language Specification Understanding. Natural language specifications, such as comments and documentation, are widely utilized in various software engineering tasks, including test case generation [11, 35, 60], bug detection [43, 60, 61], and code search [40]. Typically, C2S [60] employs semantic parsing to derive formal specifications from comments, which aids in test case generation and taint bug detection. Similarly, Jdoctor [11] and Swami [35] translate natural language specifications to formal ones to facilitate the generation of test cases covering exceptional behavior and boundary conditions, while they only focus on specific patterns, such as exceptions and numeric relations. Similar to DAInfer, Doc2Spec utilizes keywords, such as nouns and verbs indicating resource names and actions, respectively, to infer the resource specifications, which promote the resource misuse detection [61]. Targeting a different problem, DAInfer utilizes NLP models to interpret documentation, which abstracts informal semantic information into semantic units and memory operations, reducing API aliasing specification inference to an optimization problem. Although DAInfer shares similarities with existing works [40, 61] in terms of technical choices, such as named-entity recognition [18], our effort explores a new paradigm of deriving pointer facts from documentation, which can be generalized for other static analysis problems.

Large Language Models. The Large Language Models (LLMs) [37, 39], based on the decoder-only transformer architecture [51], are typically pre-trained on massive text corpora containing trillions of tokens. They exhibit exceptional zero/few-shot performance in a wide range of highly specific downstream tasks, including complex text generation [16], interactive decision making/planning [56, 62], and tool utilization [57]. Among various downstream tasks, reasoning task has traditionally been regarded as a typical challenge for LLMs [19], which has attracted significant research interests. Specifically, there has been a line of literature exploring the use of LLMs in automated theorem proving within formal logic. Pioneering studies [31, 32, 54] have focused on employing LLMs to generate proofs for theorems expressed in formal logic. Several recent efforts aimed to integrate advanced LLMs that have demonstrated impressive zero/few-shot performance in code completion tasks into formal logic reasoning tasks [33, 53]. Inspired by these studies, our work leverages the LLMs to interpret the memory operation kinds, which is one of the sub-problems in our approach.

9 CONCLUSION

We proposed a new approach DAInfer to infer API aliasing specifications from the documentation. DAInfer adopts the tagging model and the LLM to interpret informal semantic information in the documentation and reduces the inference problem to an optimization problem, which can be efficiently solved by our neurosymbolic optimization algorithm. The inferred specifications are further fed to static analysis clients for analyzing the applications using libraries. Our evaluation demonstrated the high precision and recall of DAInfer in the inference and also showed its significant impact in promoting the library-aware pointer analysis and taint analysis.

ACKNOWLEDGMENT

We thank the reviewers for their valuable comments on this work. This work is supported by the PRP/004/21FX grant from the Hong Kong Innovation and Technology Commission, Natural Science Foundation of China (62272400), and research grants from Huawei and TCL. The work was done when Chengpeng Wang was with Hong Kong University of Science and Technology. He is currently with Purdue University and available via email at wang6590@purdue.edu. Rongxin Wu is the corresponding author and works as a member of Xiamen Key Laboratory of Intelligent Storage and Computing in Xiamen University.

REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (jan 2019), 29 pages. https://doi.org/10.1145/3290353

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[3] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. (1994).

[4] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 794–807. https://doi.org/10.1145/3385412.3386026

[5] Steven Arzt and Eric Bodden. 2016. StubDroid: automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 725–735. https://doi.org/10.1145/2884781.2884816

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[7] ATLAS. 2023. Soure code of ATLAS. https://github.com/obastani/atlas. [Online; accessed 13-Sept-2023].

[8] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2018. Active learning of points-to specifications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 678–692. https://doi.org/10.1145/3192366.3192383

[9] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 2933–2942. http://proceedings.mlr.press/v48/bielik16.html

[10] Nikolaj S. Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. νZ - An Optimizing SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9035)*, Christel Baier and Cesare Tinelli (Eds.). Springer, 194–199. https://doi.org/10.1007/978-3-662-46681-0_14

[11] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 242–253. https://doi.org/10.1145/3213846.3213872

[12] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[13] Simon Butler, Michel Wermelinger, and Yijun Yu. 2015. A survey of the forms of Java reference names. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, Andrea De Lucia, Christian Bird, and Rocco Oliveto (Eds.). IEEE Computer Society, 196–206. https://doi.org/10.1109/ICPC.2015.30

[14] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26:1–26:66. https://doi.org/10.1145/2049697.2049700

[15] Bor-Yuh Evan Chang, Cezara Dragoi, Roman Manevich, Noam Rinetzky, and Xavier Rival. 2020. Shape Analysis. *Found. Trends Program. Lang.* 6, 1-2 (2020), 1–158. https://doi.org/10.1561/2500000037

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power,

Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[17] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 760–774. https://doi.org/10.1145/3314221.3314648

[18] Nancy Chinchor. 1998. Appendix E: MUC-7 Named Entity Task Definition (version 3.5). In *Seventh Message Understanding Conference: Proceedings of a Conference Held in Fairfax, Virginia, USA, MUC 1998, April 29 - May 1, 1998.* ACL. https://aclanthology.org/M98-1028/

[19] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *CoRR* abs/2110.14168 (2021). arXiv:2110.14168 https://arxiv.org/abs/2110.14168

[20] DAInfer. 2024. Implementation of DAInfer. https://github.com/DAInfer/DAInfer. [Online; accessed 28-Apr-2024].

[21] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[22] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin T. Vechev. 2019. Unsupervised learning of API alias specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 745–759. https://doi.org/10.1145/3314221.3314640

[23] F-Droid. 2023. F-Droid. https://f-droid.org/. [Online; accessed 1-Sept-2023].

[24] Pratik Fegade and Christian Wimmer. 2020. Scalable pointer analysis of data structures using semantic models. In *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*, Louis-Noël Pouchet and Alexandra Jimborean (Eds.). ACM, 39–50. https://doi.org/10.1145/3377555.3377885

[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[26] W Nelson Francis and Henry Kucera. 1967. Computational analysis of present-day American English. *Providence, RI: Brown University Press. Kuperman, V., Estes, Z., Brysbaert, M., & Warriner, AB (2014). Emotion and language: Valence and arousal affect word recognition. Journal of Experimental Psychology: General* 143 (1967), 1065–1081.

[27] Timon Gehr, Dimitar K. Dimitrov, and Martin T. Vechev. 2015. Learning Commutativity Specifications. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 307–323. https://doi.org/10.1007/978-3-319-21690-4_18

[28] Jingxuan He, Cheng-Chun Lee, Veselin Raychev, and Martin T. Vechev. 2021. Learning to find naming issues with big code and small supervision. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 296–311. https://doi.org/10.1145/3453483.3454045

[29] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[30] Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. 2010. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.* 32, 2 (2010), 5:1–5:52. https://doi.org/10.1145/1667048.1667050

[31] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygózdz, Piotr Milos, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding Hammers to Integrate Language Models and Automated Theorem Provers. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2022/hash/377c25312668e48f2e531e2f2c422483-Abstract-Conference.html

[32] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal

Proofs. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=SMa9EAovKMC

[33] Guillaume Lample, Timothée Lacroix, Marie-Anne Lachaux, Aurélien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. 2022. HyperTree Proof Search for Neural Theorem Proving. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2022/hash/a8901c5e85fb8e1823bbf0f755053672-Abstract-Conference.html

[34] George A. Miller. 1995. WordNet: a lexical database for English. *Commun. ACM* 38, 11 (nov 1995), 39–41. https://doi.org/10.1145/219717.219748

[35] Manish Motwani and Yuriy Brun. 2019. Automatically generating precise Oracles from structured natural language specifications. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 188–199. https://doi.org/10.1109/ICSE.2019.00035

[36] NLTK. 2023. Natural Language Toolkit. https://www.nltk.org/index.html. [Online; accessed 7-Sep-2023].

[37] OpenAI. 2022. Introducing ChatGPT. (2022). https://openai.com/blog/chatgpt

[38] OpenAI. 2023. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5. [Online; accessed 7-Sep-2023].

[39] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[40] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit M. Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 815–825. https://doi.org/10.1109/ICSE.2012.6227137

[41] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2019. Predicting program properties from 'big code'. *Commun. ACM* 62, 3 (2019), 99–107. https://doi.org/10.1145/3306204

[42] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 419–428. https://doi.org/10.1145/2594291.2594321

[43] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-Misuse Detection Driven by Fine-Grained API-Constraint Knowledge Graph. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 461–472. https://doi.org/10.1145/3324884.3416551

[44] Atanas Rountev, Mariana Sharp, and Guoqing Xu. 2008. IDE dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*. Springer, 53–68.

[45] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.* 24, 3 (2002), 217–298. https://doi.org/10.1145/514188.514190

[46] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 693–706. https://doi.org/10.1145/3192366.3192418

[47] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, David S. Rosenblum and Sebastian G. Elbaum (Eds.). ACM, 174–184. https://doi.org/10.1145/1273463.1273487

[48] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java (Artifact). *Dagstuhl Artifacts Ser.* 2, 1 (2016), 12:1–12:2. https://doi.org/10.4230/DARTS.2.1.12

[49] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specification and Detecting Violations. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, Paul C. van Oorschot (Ed.). USENIX Association, 379–394. http://www.usenix.org/events/sec08/tech/full_papers/tan_l/tan_l.pdf

[50] John Toman and Dan Grossman. 2017. Taming the Static Analysis Beast. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.18

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[52] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 319–330. https://doi.org/10.1145/3236024.3236056

[53] Sean Welleck, Jiacheng Liu, Ximing Lu, Hannaneh Hajishirzi, and Yejin Choi. 2022. NaturalProver: Grounded Mathematical Proof Generation with Language Models. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2022/hash/1fc548a8243ad06616eee731e0572927-Abstract-Conference.html

[54] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with Large Language Models. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2022/hash/d0c6bc641a56bebee9d985b937307367-Abstract-Conference.html

[55] Peisen Yao, Jinguo Zhou, Xiao Xiao, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Efficient Path-Sensitive Data-Dependence Analysis. *CoRR* abs/2109.07923 (2021). arXiv:2109.07923 https://arxiv.org/abs/2109.07923

[56] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *CoRR* abs/2305.10601 (2023). https://doi.org/10.48550/arXiv.2305.10601 arXiv:2305.10601

[57] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=WE_vluYUL-X

[58] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. 2021. Semi-automated protocol disambiguation and code generation. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 272–286. https://doi.org/10.1145/3452296.3472910

[59] Hamed Zamani and W Bruce Croft. 2017. Relevance-based word embedding. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 505–514.

[60] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: translating natural language comments to formal program specifications. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 25–37. https://doi.org/10.1145/3368089.3409716

[61] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring Resource Specifications from Natural Language API Documentation. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 307–318. https://doi.org/10.1109/ASE.2009.94

[62] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=WZH7099tgfM

Received 2023-09-28; accepted 2024-04-16