SpecGuru: Hierarchical LLM-Driven API Points-to Specification Generation with Self-Validation

Shuangxiang Kan
University of New South Wales
Kensington, NSW, Australia
shuangxiang.kan@unsw.edu.au

YueKang Li
University of New South Wales
Kensington, NSW, Australia
yuekang.li@unsw.edu.au

Xiao Cheng*
Macquarie University
Macquarie Park, NSW, Australia
xiao.cheng@mq.edu.au

Yulei Sui
University of New South Wales
Kensington, NSW, Australia
y.sui@unsw.edu.au

ABSTRACT

Static analysis is a fundamental technique for ensuring software safety and reliability. However, when analyzing client code that invokes third-party APIs, proper handling of API source code becomes critical. A common approach is to utilize API specifications that approximate the essential behaviors of the API while avoiding direct analysis of the implementation. Existing methods for constructing API specifications often fail to adequately address challenges presented by complex semantics, syntax, and edge cases.

In this paper, we introduce SpecGuru, a framework that leverages Large Language Models (LLMs) to automatically generate points-to API specifications. SpecGuru employs a novel bottom-up approach that begins with leaf functions (those without dependencies) and progressively builds specifications through rigorous validation. These validated specifications subsequently serve as abstractions at call sites of higher-level functions along the call hierarchy, effectively eliminating the need to process complex source code directly with LLMs. Our approach incorporates a robust self-validation mechanism using automatically synthesized test cases for differential testing throughout the specification inference process, which prevents error propagation and ensures incremental correctness of the generated specifications.

Our experimental evaluation on 15 third-party C libraries shows that SpecGuru significantly outperforms existing specification inference tools, generating 21% more specifications than Spectre and 46% more than c-summary. Moreover, our bottom-up abstraction technique combined with automated testing methodology enables the discovery of a more comprehensive set of specifications while effectively preventing error propagation, achieving analysis results comparable to those obtained using complete API source code.

KEYWORDS

Large language model, Specification generation, Static analysis

*Xiao Cheng is the corresponding author.

ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773209

ACM Reference Format:
Shuangxiang Kan, YueKang Li, Xiao Cheng, and Yulei Sui. 2026. SpecGuru: Hierarchical LLM-Driven API Points-to Specification Generation with Self-Validation. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773209

1 INTRODUCTION

Static analysis of code interfacing with external APIs presents a fundamental challenge: incorporating complete API source code with client code ensures analytical precision but introduces computational redundancy, while omitting API analysis compromises soundness. API specifications, concise abstractions that capture essential behaviors, offer an efficient solution by modeling critical properties without requiring implementation-level analysis [8, 68].

Points-to analysis [8, 9, 28, 54] constitutes a cornerstone of static analysis, determining potential memory locations referenced by pointer variables. This information forms the foundation for data-flow analysis, call graph construction, and vulnerability detection. In C/C++ programs, where pointers are pervasive, API points-to specifications enable precise cross-boundary analysis without examining implementation details, reduce analytical complexity through abstraction, enhance scalability through modular analysis, and facilitate reuse across diverse client applications. The generation of points-to specifications serves as an important complement to existing static analysis tools, helping improve their overall precision.

Despite the importance of API specifications for static analysis, existing approaches to inferring points-to specifications face significant limitations when addressing the complexities of modern software systems. Current techniques primarily rely on manual annotation [2, 15, 56], static analysis [8, 26, 30, 68], or dynamic execution [9, 10, 17] and fuzzing [27]. Manual approaches, while potentially precise, demand substantial domain expertise and impose prohibitive engineering costs at scale. Static analysis methods typically require intricate rule definitions to handle language-specific features and often struggle when faced with incomplete information, particularly at API boundaries where standard library functions like `memcpy` or `strcpy` may lack accessible implementations. Dynamic analysis approaches necessitate extensive instrumentation infrastructure, depend heavily on the quality of execution oracles, and suffer from coverage limitations that may omit critical pointer behaviors in rarely executed code paths. Given these limitations, there

remains a clear need for a more effective approach capable of generating accurate points-to specifications while handling complex language constructs (e.g., nested structures, multi-level function calls, and sophisticated control flows) and incomplete information scenarios. The desired approach would enable reliable and efficient analysis of client code that interacts with third-party APIs without requiring whole-program analysis to repeatedly re-analyze stable API implementations when client code changes (e.g., due to code modifications), thereby avoiding redundant computational overhead.

Recent advancements in Large Language Models (LLMs) have revealed significant capabilities across diverse software engineering tasks, including code generation, translation, and synthesis [11–13, 20, 33, 48]. The sophisticated ability of LLMs to analyze and comprehend source code presents a promising approach for generating API specifications through direct examination of API implementations. By systematically processing API source code, LLMs can infer complex pointer behaviors and data flows by leveraging their pre-trained knowledge of code semantics and common programming patterns. LLMs offer distinct advantages over traditional specification inference approaches. Unlike conventional static analysis techniques that require explicit rule formulation for handling language-specific features, LLMs inherently recognize various programming idioms and structural patterns through their extensive training on code repositories. In contrast to dynamic analysis methods that necessitate comprehensive instrumentation to capture execution traces, LLMs can reason about potential pointer behaviors across different control flow paths without requiring concrete execution scenarios. These capabilities directly address the fundamental limitations of existing approaches: they avoid the manual rule engineering required by static analysis and overcome the coverage constraints and runtime overhead inherent to dynamic analysis.

While the application of LLMs to API specification generation presents a promising direction, significant challenges remain in this domain. The effectiveness of LLM-generated specifications is constrained by two fundamental challenges inherent to applying these models in the context of program analysis:

Challenge#1: Reliable specification generation. LLMs exhibit inherent limitations in reasoning accuracy, often producing erroneous outputs or hallucinations [7, 25, 49]. These problems become especially challenging when analyzing complex code structures with intricate call hierarchies, sophisticated language features, and domain-specific programming idioms, which is precisely the context in which API specification generation operates. The difficulty is further amplified in languages such as C, where subtle implementation details impact pointer relationships and memory behavior, and thus the correctness of the resulting specification.

Challenge#2: Specification validation. Ensuring the correctness of generated specifications is crucial for static analysis applications. While test-driven validation is an effective approach [11–13, 20, 33, 48], it traditionally demands comprehensive, high-quality test suites that impose significant manual engineering overhead, and even carefully designed test suites may miss edge cases or subtle pointer behaviors [41]. This validation gap demands automated, reliable techniques for generating comprehensive test cases that can validate LLM-generated specifications effectively.

To address these challenges, we present SpecGuru, a novel framework that systematically leverages LLMs to generate precise API points-to specifications. For **Challenge#1**, we implement a hierarchical **divide-and-conquer** approach that resolves the fundamental tension between providing comprehensive API context and accommodating LLMs' context limitations. Beginning with leaf functions (those without API dependencies), we generate and validate specifications that subsequently serve as concise abstractions when analyzing higher-level functions in a **bottom-up** manner guided by the call graph structure. This progressive abstraction technique preserves essential semantic information while maintaining manageable input sizes, enabling effective specification inference across complex call hierarchies without exceeding the model's reasoning capacity. For **Challenge#2**, we introduce a novel **self-validation** methodology that builds upon recent findings demonstrating LLMs' proficiency in generating high-quality test cases [22, 32, 34, 61]. We employ **differential testing** with LLM-synthesized test cases to systematically compare execution behaviors between original implementations and generated specifications. This self-validation mechanism operates at each level of the call hierarchy, ensuring specification correctness throughout the entire bottom-up abstraction chain and preventing error propagation that would otherwise compound across successive layers of abstraction.

The bottom-up points-to specification inference combined with self-validation mechanism decomposes the complex task of library API specification inference into manageable components. This approach enables SpecGuru to efficiently generate and validate concise, precise specifications for third-party libraries incrementally, addressing the limitations of attempting to process entire library implementations at once. By systematically building specifications from leaf functions upward and validating each component through differential testing, SpecGuru achieves both analytical precision and scalability while maintaining strong correctness guarantees throughout the specification hierarchy. We conducted experiments on 15 third-party C libraries to validate the performance of SpecGuru. SpecGuru outperforms traditional static/dynamic analysis-based specification generation methods by inferring 33% more specifications successfully on average. Compared to using isolated or the complete source code of APIs under analysis, our bottom-up approach demonstrated better performance, and the automated testing strategy effectively prevented the spread of incorrect specification information. Additionally, the generated API specifications can achieve static analysis results comparable to those obtained using their source code.

In summary, this paper makes the following contributions:

- A new bottom-up approach for API specification generation using LLMs that first processes leaf functions, validates their specifications, and then progressively builds higher-level specifications using validated lower-level abstractions.
- A robust self-validation mechanism that employs differential testing with LLM-synthesized test cases at each abstraction level, ensuring specification correctness and preventing error propagation throughout the specification inference hierarchy.
- Empirical evaluation on 15 popular C libraries demonstrating that SpecGuru generates more comprehensive specifications than traditional static analysis methods while achieving comparable

analysis results to using complete API source code. Implementation available at: https://github.com/SVF-tools/SpecGuru.

2 PRELIMINARIES AND PROBLEM FORMULATION

This section introduces the preliminary knowledge for our work: specification inference, automated test case generation, and large language model capabilities for code, followed by our formulation of the API points-to specification inference problem.

2.1 Specification Inference

Specification inference is the task of inferring or modeling specifications that describe the behavior, constraints, and properties of programs or functions. Previous works have explored automated specification inference [6, 8, 9, 17, 42, 68], but mainly focused on modeling third-party library APIs for Java and Android. Regarding specification inference for C functions, existing approaches such as *c-summary* [30], which are based on static analysis, struggle to accurately model C function behaviors due to C's complex language features. Leveraging LLMs' strong understanding of C language semantics, we explore using LLMs to infer specifications for library APIs in the context of pointer analysis in this work.

2.2 Automated Test Case Generation

Automated test case generation aims to automatically create test cases to validate software functionality while reducing manual effort. Traditional approaches fall into two main categories: search-based methods [5, 24, 38, 44] that use techniques like genetic algorithms to evolve test cases, and model-based approaches [31, 46, 57] that leverage pre-trained models to generate tests. However, these approaches face significant limitations. Search-based methods often struggle to generate diverse and comprehensive test cases, particularly for complex APIs. Model-based approaches require extensive training data and labeled datasets, which are often unavailable for specialized third-party library specifications.

Recent research has shown the promising capabilities of Large Language Models (LLMs) in automated testing [11, 22, 32, 34, 61, 64, 66]. For third-party library specifications validation, LLMs offer several advantages: they have the ability to understand code context to generate more relevant test cases, require no additional training data, and can produce diverse test scenarios. Given these benefits, we leverage LLMs in our approach to validate generated specifications, addressing the limitations of traditional automated testing methods.

2.3 Large Language Models for Code Generation

Recently, Large Language Models (LLMs) like the GPT series have demonstrated promising capabilities to understand code [62, 67]. Leveraging LLMs for specification inference offers the following two unique advantages over traditional code analysis methods:

- **Code level**: LLMs can handle complex code structures, such as deeply nested loops and recursive calls, by recognizing their intent and functionality based on learned patterns. This allows LLMs to provide better analysis even for unfamiliar codebases, unlike traditional static analysis tools that require complex designs specifically tailored to recognize these structures.
- **Semantic level**: LLMs can understand the code snippets based on context and naming conventions, even without complete source code or documentation. For example, given a function call like `strcat(buf, input)`, LLMs can infer from the function name that this is a string concatenation operation that appends `input` to `buf` and returns the address of `buf`, even if the implementation of `strcat` is unavailable (e.g., provided only as a binary). This is particularly valuable when analyzing C libraries that invoke APIs from other libraries (e.g., C standard library or proprietary binaries) where traditional static analysis tools may fail to resolve semantics due to lack of source code, but LLMs can reason about functionality based on contextual clues, such as function and variable names, caller-callee relations, etc.

2.4 Problem Formulation

API points-to specification inference generates simplified API implementations that preserve essential pointer relationships while removing irrelevant details. These specifications maintain equivalent pointer interactions between function parameters and return values, supporting accurate static analysis with reduced complexity. SpecGuru leverages LLMs' code understanding to focus on preserving four fundamental pointer relationships that govern how data flows through API parameters and return values.

Definition 1 (Pointer Analysis Properties)**.** Given a library API f, let $V = P \cup R$, where P is the set of parameters and R is the return value of f. We define $\Phi = \{\phi_{\text{copy}}, \phi_{\text{load}}, \phi_{\text{store}}, \phi_{\text{field}}\}$ as a set of pointer analysis properties that describe pointer interactions between variables $x, y \in V$ in f:

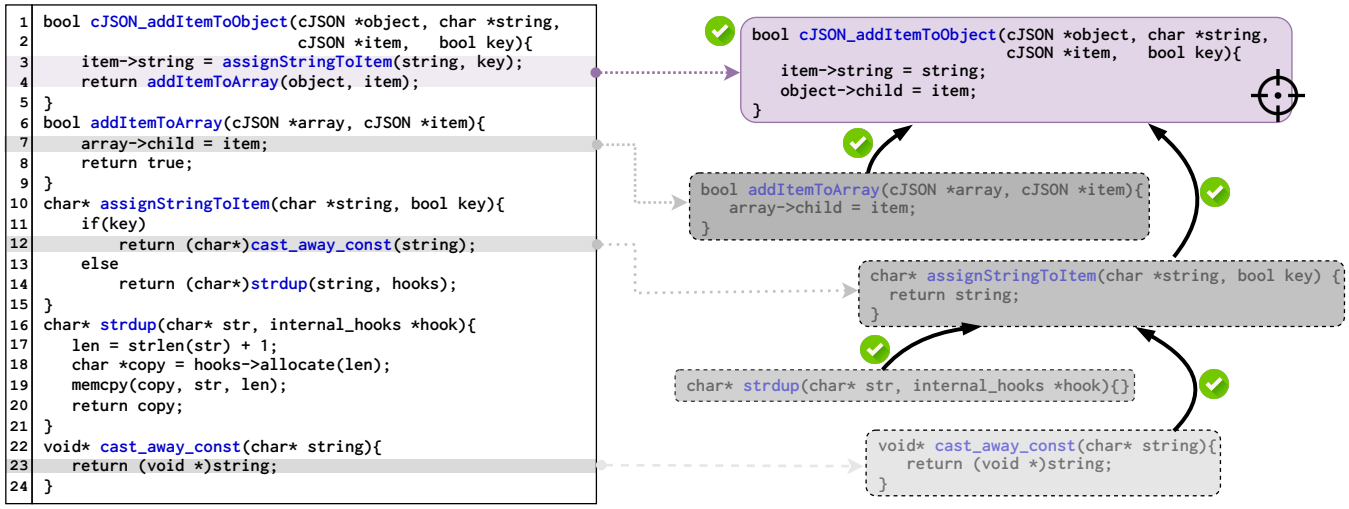
- $\phi_{\text{copy}} : p = q$, indicates that the value of q is assigned to p.
- $\phi_{\text{load}} : p = *q$, indicates that the value at the memory location pointed to by q is loaded into p.
- $\phi_{\text{store}} : *p = q$, indicates that the memory location pointed to by p is updated with the value of q.
- $\phi_{\text{field}} : p = \&q \rightarrow f$, indicates that the field f within the structure pointed to by q is accessed and assigned to p.

We formulate our points-to specification inference as follows:

> Given a library API's source code f and pointer properties Φ from Definition 1, our points-to specification inference task generates a simplified implementation S_f while preserving identical pointer behavior: $\forall \phi \in \Phi, \mathcal{B}(f, \phi) \Leftrightarrow \mathcal{B}(S_f, \phi)$, where $\mathcal{B}(f, \phi)$ represents f's behavior regarding property ϕ.

3 MOTIVATING EXAMPLE

Fig. 1 illustrates the key idea of SpecGuru's API points-to specification generation through an API function, `cJSON_addItemToObject` (ℓ_1-ℓ_5), extracted from the third-party C library `cJSON` [1]. The third parameter `item` of `cJSON_addItemToObject` is ultimately assigned to the `child` field of the first parameter `object` via ℓ_7 of



Simplified API Source Code **Bottom-Up Points-to Specification Inference and Validation**

Figure 1: A motivating example illustrating how SpecGuru performs bottom-up points-to specification inference and validation for the library API `cJSON_addItemToObject` extracted from the `cJSON` library [1].

`addItemToArray` when invoked at ℓ_4. Furthermore, the second parameter `string` propagates through a sequence of function calls—first to `assignStringToItem` at ℓ_3, then to `castAwayConst` at ℓ_{12}—before being assigned to the `string` field of the third parameter `item` at ℓ_{23}.

When analyzing client code that uses `cJSON_addItemToObject`, traditional static analyzers would need to process the complete implementation of this API function and all its callees to ensure analytical precision. However, this repeated analysis can be circumvented by employing a concise specification (depicted in the purple rounded rectangle in Fig. 1). This specification succinctly captures the essential pointer operations among parameters, representing the core behavior where a cJSON object (`item`) is added to another cJSON object (`object`) and assigned a name (`string`). Substituting this specification for the full implementation yields identical pointer analysis results while substantially reducing the analytical complexity from 24 lines of intricate code to a mere 4 lines of focused implementation.

Bottom-Up Specification Inference. SpecGuru implements a bottom-up approach integrated with divide-and-conquer principles for LLMs to perform points-to specification inference. This methodology leverages a key insight: points-to information naturally propagates from callees to callers within function hierarchies. Rather than attempting to analyze the entire implementation at once or inline all callees into the root function, SpecGuru systematically decomposes the problem according to the call graph. Specifically, SpecGuru initiates analysis with leaf functions (those making no calls to other functions within the same library) before ascending the call hierarchy. This strategic decomposition transforms the complex task of analyzing entire libraries into modular, function-level analyses that are significantly more manageable. As illustrated in Fig. 1, SpecGuru first synthesizes specifications for the leaf functions `castAwayConst`, `strdup`, and `addItemToArray`. These specifications are subsequently incorporated into the analysis of their respective callers—`assignStringToItem` and ultimately `cJSON_addItemToObject`—thereby facilitating efficient and comprehensive specification inference throughout the entire library API.

Specification Self-Validation. Another challenge is the validation of inferred specifications: third-party libraries serve diverse purposes across numerous domains, making it impractical to rely on pre-existing test cases for each library. Manual inspection or custom test case development would be prohibitively resource-intensive and undermine the automation benefits of our approach. To address this validation challenge, SpecGuru implements a self-validation strategy that leverages recent research demonstrating LLMs' effectiveness in generating high-quality test cases [22, 32, 34, 61]. This approach enables systematic validation of inferred points-to specifications without requiring domain-specific knowledge or pre-existing test suites.

For specification validation, SpecGuru employs a systematic differential testing approach. First, the test case generation LLM synthesizes `test_case_1` based on the function signature and the identified points-to relationships between parameters and return values. Subsequently, SpecGuru derives `test_case_2` by replicating `test_case_1` but substituting the function's implementation with its inferred specification. Through comparative execution of these paired test cases, SpecGuru evaluates whether the specification accurately captures the original implementation's pointer behavior.

```
*** test_case_1 ***:
int main(){
 ...
 # use source code
 addItemToArray(arr,it);
 if(arr == it)
 {...}
 ...
}
```

```
*** test_case_2 ***:
int main(){
 ...
 # use specification
 addItemToArray(arr,it);
 if(arr == it)
 {...}
 ...
}
```

As demonstrated in the listing above, SpecGuru generates paired test cases to verify each inferred points-to relationship. For instance, when validating the alias relationship between parameters `arr` and

`it` in `addItemToArray` from Fig. 1, the test cases specifically examine whether this pointer relationship is preserved. If the execution outputs of the testcase pair are equivalent, the specification is considered correct, as it accurately captures the pointer behavioral semantics of the original implementation. This systematic validation process extends throughout the entire call hierarchy, applying the same rigorous validation to every function from leaf nodes to their callers. Such comprehensive validation ensures the integrity of all specifications generated during SpecGuru's bottom-up analysis, effectively mitigating the risk of error propagation across the specification chain.

4 SPECGURU APPROACH

Fig. 2 illustrates the framework overview of SpecGuru: (a) Preprocessing, (b) Bottom-Up Specification Inference, (c) Specification Self-Validation, and (d) Clients. Stages (b) and (c) are interleaved during execution to ensure specification quality. Initially, SpecGuru preprocesses the source code to partition functions based on their call dependency relationships. The system then iteratively processes functions in bottom-up order: for each function, a specification is generated and immediately validated through differential testing before proceeding to functions that depend on it. This integrated approach of generation and validation ensures that only validated specifications propagate to the analysis of dependent functions. The resulting validated specifications are then provided to client applications for subsequent client applications, such as pointer alias analysis [18] and taint analysis [36].

4.1 Preprocessing

Given a set of functions $\mathbb{F}$ in a targeted library, since SpecGuru focuses on generating API specifications related to pointer analysis for functions in $\mathbb{F}$, it is required that the function's parameters or return value are of pointer types. SpecGuru excludes functions where either the return type and all parameter types are non-pointers, or only one of them is a pointer type, sets their specification function bodies to empty to indicate the absence of pointer relationships, and adds them to the summarized specification set $\mathbb{S}$. This helps avoid unnecessary LLM calls. The targeted functions are defined as:

$$\mathbb{F}' = \{f \in F \mid |\{t_i \mid \rho(t_i), i \in \{r\} \cup \{0, 1, ..., n\}\}| \geq 2\}$$

where t_r represents the return value type, $t_0, ..., t_n$ represent parameter types, and $\rho(t)$ indicates whether type t is a pointer type. Furthermore, we categorize functions in $\mathbb{F}'$ into leaf function set $\mathbb{L}$ and non-leaf function set $\mathbb{B}$ based on their callees for generating specifications in a bottom-up manner.

Definition 2 (Leaf Function)**.** A function ℓ in $\mathbb{L}$ is a leaf function if and only if the set of its callees $Callee(\ell)$ satisfies: $Callee(\ell) = \emptyset$ or, $Callee(\ell) = \{c \mid c \in Callee(\ell), c \notin \mathbb{F} \text{ or } c = \ell\}$.

Example 1. Leaf functions can only call functions defined outside the library (external APIs). Functions like `strdup`, `castAwayConst` and `addItemToArray` in Fig. 1 are leaf functions, as `strdup`'s call to `memcpy` from the C standard library does not affect its leaf status since `memcpy` is an external API.

Definition 3 (Non-Leaf Function)**.** A function δ in $\mathbb{B}$ is a non-leaf function if and only if its callee set $Callee(\delta)$ satisfies: $Callee(\delta) \cap \{m \mid m \in \mathbb{F}, m \neq \delta\} \neq \emptyset$.

Example 2. `cJSON_addItemToObject` and `assignStringToItem` in Fig. 1 are non-leaf functions since they call other functions within the same library.

Functions in $\mathbb{F}'$ may call external APIs without available source code. For such cases, we leverage LLMs' prior code training to infer external API functionalities and pointer behaviors from API signatures and calling context. For example, given `char* result = strstr(str, substr)`, LLMs can infer this is a string search function returning the first substring occurrence, without requiring source code or documentation. Even if LLMs make incorrect inferences, all generated specifications are validated through differential testing (Section 4.3), ensuring only correct and executable specifications are retained.

4.2 Bottom-Up Specification Inference

Based on the categorized sets of leaf functions $\mathbb{L}$ and non-leaf functions $\mathbb{B}$, SpecGuru employs a bottom-up analysis approach, starting with the leaf functions. As shown in Fig. 3, to get the candidate specification S'_ℓ for each leaf function $\ell \in \mathbb{L}$:

$$S'_\ell \leftarrow \mathcal{M}(D_{spec}, E, C_\ell)$$

the prompts fed into the language model $\mathcal{M}$ include specification description D_{spec}, examples of the specifications E, and the source code of the leaf function C_ℓ. Once the candidate specification S'_ℓ is generated, a differential test $\mathcal{D}\big(T^1(S'_\ell), T^2(C_\ell)\big)$ is conducted. If the test passes, the specification S_ℓ (which is the same as S'_ℓ but has successfully passed the differential test and been selected from the specification candidate set; see details in Section 4.3) is considered correct and added to the specification set $\mathbb{S}$. This process continues until all leaf function specifications have been added to $\mathbb{S}$.

Then SpecGuru proceed to handle each non-leaf function $\delta \in B$ recursively. Similar to obtaining specifications for leaf functions, to get candidate specification S'_δ for a non-leaf function δ, the prompt to the language model $\mathcal{M}$ not only includes the desired specification description D_{spec}, examples of the specifications E, the source code of the non-leaf function C_δ, but also include information about δ's callees S_{cs}:

$$S'_\delta \leftarrow \mathcal{M}(D_{spec}, E, C_\delta, S_{cs})$$

The callee information S_{cs} varies based on two types of non-leaf functions:

- *Non-leaf functions without mutual recursion:* For functions that do not involve mutual recursion, S_{cs} contains only the specifications of its callees that have already been summarized in $\mathbb{S}$:

  $$S_{cs} \leftarrow Callee_F(\delta) \cap \mathbb{S}$$

  where $Callee_F(\delta)$ represents the set of callees in F, excluding external API calls.
- *Non-leaf functions with mutual recursion:* For functions involving mutual recursion, $Callee_F(\delta)$ contains at least one unsummarized callee. Therefore, S_{cs} includes both the specifications of summarized callees and the source code of unsummarized callees:

  $$S_{cs} \leftarrow \{Callee_F(\delta) \cap \mathbb{S}\} \cup \{Callee_F(\delta) \cap \{F - \mathbb{S}\}\}$$

For mutually recursive functions, the combined source code may exceed the LLM's token limit. To address this, we first calculate the total tokens after including all callees' specifications or source code.

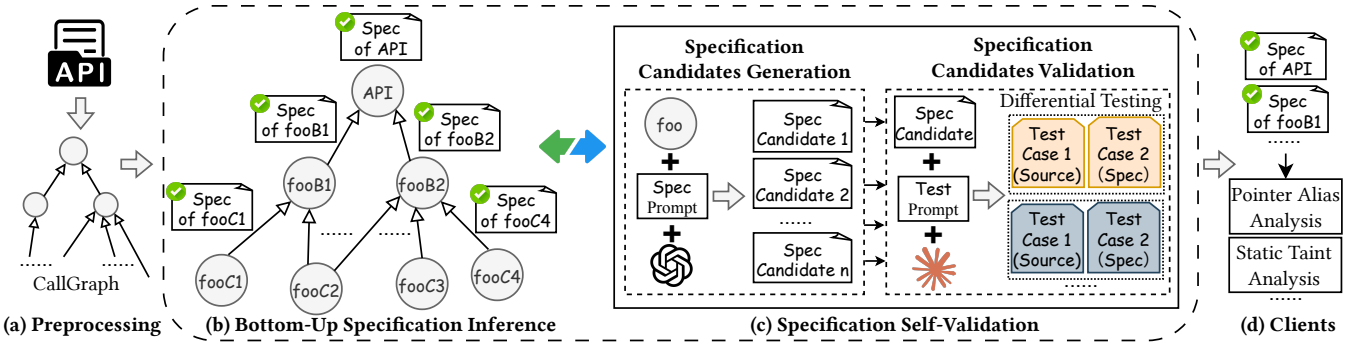


Figure 2: SpecGuru framework overview.

$$S'_\ell \leftarrow \mathcal{M}(D_{spec}, E, C_\ell) \quad (1) \qquad S'_\delta \leftarrow \mathcal{M}(D_{spec}, E, C_\delta, S_{cs}) \quad (4)$$

$$S_\ell \leftarrow \mathcal{D}(T^1(S'_\ell), T^2(C_\ell)) \quad (2) \qquad S_\delta \leftarrow \mathcal{D}(T^1(S'_\delta), T^2(C_\delta)) \quad (5)$$

$$\mathbb{S} \leftarrow \mathbb{S} \cup \{S_\ell\} \quad (3) \qquad \mathbb{S} \leftarrow \mathbb{S} \cup \{S_\delta\} \quad (6)$$

Figure 3: Leaf and Non-leaf function specification inference

If this exceeds the maximum limit, we iteratively remove the callee with the highest token count from the mutually recursive function set until the total falls within the token limit.

Once S'_δ is obtained, it undergoes differential testing to validate its correctness. If the test passes, S_δ (same as S'_δ but has passed the differential test) is added to $\mathcal{S}$. This recursive process ensures that specifications are generated in a bottom-up manner, with each non-leaf function building upon the validated specifications of its dependencies.

Example 3. For 5 functions in Fig. 1, SpecGuru firstly generates specifications for leaf functions `castAwayConst`, `strdup`, and `addItemToArray`. For non-leaf function `assignStringToItem`, since it calls `castAwayConst` and `strdup`, their specifications are combined with `assignStringToItem`'s source code as input to the LLM to generate its specification. Similarly, for another non-leaf function `cJSON_addItemToObject`, the specifications of `addItemToArray` and `assignStringToItem` are combined with its source code as input to generate its specification.

4.3 Specification Self-Validation

Since LLMs can make mistakes, we cannot rely on a single output S'_f from LLMs for identifying the ground truth $\hat{S}_f$ (the ideal specification that correctly captures all pointer relationships Φ for function f). Instead, SpecGuru generates multiple specification candidates for function f and compiles each generated specification S'_{fj}. We drop all specifications that fail to compile or execute properly. Only specifications that successfully pass both compilation and execution tests are then included in the candidate solution set $\mathcal{S}'_f = \{S'_{f_1}, S'_{f_2}, ..., S'_{f_n}\}$. This validation step ensures that all candidates in $\mathcal{S}'_f$ are syntactically correct and executable, eliminating potential issues with malformed or non-functional specifications. SpecGuru then selects from $\mathcal{S}'_f$ the candidate solution S'_{fi} that most closely matches the ground truth $\hat{S}_f$. Nonetheless, this approach presents two key challenges:

(1) How to validate that each candidate specification S'_{fi} correctly preserves the pointer relationships Φ?

(2) Since we have no access to the ground truth $\hat{S}_f$ as a reference, how can SpecGuru select the validated candidate specification S'_{fi} from $\mathcal{S}'_f$ that most closely matches $\hat{S}_f$?

4.3.1 ***Validation.*** In the validation stage, we continue to employ LLMs to generate test cases T_i for validating each specification candidate S'_{fi}. The feasibility of using LLM-generated test cases to validate LLM-identified specifications is based on the following assumptions [11]:

- The generation of specification candidate S'_{fi} and test case T_i are independent processes with distinct context prompts;
- The types of errors are diverse, making the probability that a faulty test case T_i correctly validates an incorrect specification candidate S'_{fi} very low.

To further reduce false positives caused by LLM hallucinations and to validate whether S'_{fi} preserves the same pointer analysis properties Φ as the API source code C_f, SpecGuru generates two test cases for each S'_{fi}: $T_i^1(S'_{fi})$ and $T_i^2(C_f)$. The only difference between $T_i^1(S'_{fi})$ and $T_i^2(C_f)$ is that the $T_i^1(S'_{fi})$ calls the specification candidate S'_{fi} while $T_i^2(C_f)$ calls the source code C_f. For simplicity, we use T_i^1 and T_i^2 to represent $T_i^1(S'_\ell)$ and $T_i^2(C_f)$ when it does not cause confusion.

Therefore, for candidate set $\mathcal{S}'_f$, SpecGuru generate a set of test case pairs:

$$T = \{\{S'_{f_1}, (T_1^1, T_1^2)\}, \{S'_{f_2}, (T_2^1, T_2^2)\}, ..., \{S'_{f_n}, (T_n^1, T_n^2)\}\}$$

to filter out specification candidates that fail differential testing. Specifically, for the generation of T_i^1, the language model $\mathcal{M}'$ is provided with a prompt that includes: (1) a test case description D_{test}, which outlines the requirements for creating test cases to validate the function's behavior; (2) the identified property instances description P for the specification candidate S'_{fi} (e.g., ϕ_{copy} property instance of `(array->child, item)` of `addItemToArray` function); and (3) the source code C_f of the function f. These components are combined to instruct $\mathcal{M}'$ to generate a test case T_i^1 that exercises the function and validates the specified properties:

$$T_i^1 \leftarrow \mathcal{M}'(D_{test}, P, C_f)$$

and T_i^2 is derived by **replacing** the function implementation C_f in T_i^1 with specification candidate S'_{fi}:

$$T_i^2 \leftarrow \mathcal{R}(T_i^1, S'_{fi})$$

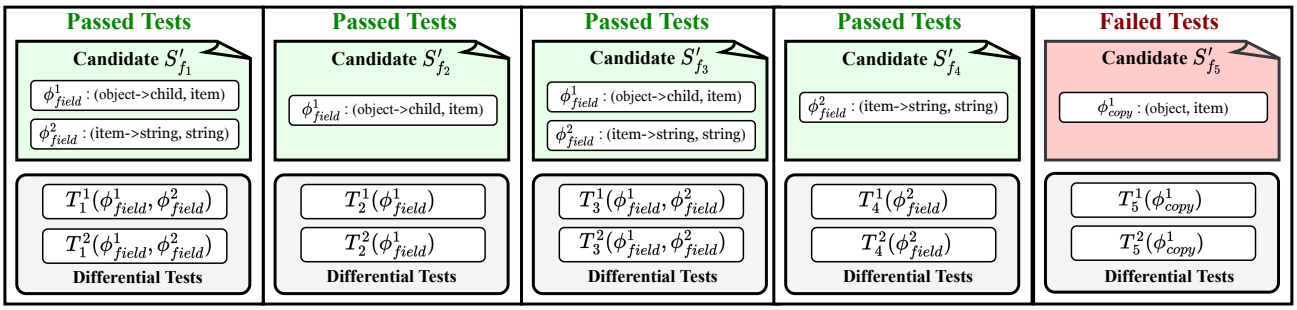


Figure 4: Specification candidates S'_{f_1}, S'_{f_2}, S'_{f_3}, and S'_{f_4} pass the differential test (inliers), candidate S'_{f_5} fails (outlier).

The only difference between T^1_i and T^2_i is the function body of f, which allows us to perform differential testing between them:

$$\mathcal{D}(T^1_i, T^2_i)$$

By comparing the outputs between T^1_i and T^2_i, SpecGuru can identify any discrepancies or inconsistencies, which may indicate potential inaccuracies in S'_{f_i} or flaws in the generated test cases T^1_i and T^2_i. If the outputs of T^1_i and T^2_i are consistent, SpecGuru consider S'_{f_i} to be a correct candidate specification. This differential testing process helps to increase our confidence in the correctness of the instance S'_{f_i} and the effectiveness of the generated test cases.

Example 4. Consider the `cJSON_addItemToArray` function from Section 3. SpecGuru first employs an LLM to detect the ϕ_{copy} property between the first parameter (`array`) and second parameter (`item`), generating a specification candidate S'_{f_1} containing corresponding ϕ_{copy} property instance. Next, SpecGuru makes an independent LLM call with a different prompt context to create `Test_Case_1` with two key components: (1) Invocations of `cJSON_addItemToArray` using the original implementation, and (2) Validation checks for pointer aliasing between `array` and `item`.

SpecGuru then derives `Test_Case_2` by duplicating `Test_Case_1` and replacing `cJSON_addItemToArray`'s implementation with specification candidate S'_{f_1}. After separately compiling and executing both test cases, SpecGuru compares their outputs. Consistent results between the test cases confirm that S'_{f_1} accurately captures the pointer property demonstrated in the original implementation.

4.3.2 ***Selection.*** To identify the candidate S'_{f_i} that most closely aligns with the target solution $\hat{S}_f$, SpecGuru employ a strategy inspired by the classic RANSAC algorithm from machine learning [11, 23]. A candidate S'_{f_i} is considered to best align with $\hat{S}_f$ if it satisfies both criteria: (1) it passes the differential test, confirming its behavioral consistency with the original function implementation, and (2) it contains the maximum number of pointer analysis property instances, indicating it captures the most comprehensive set of properties.

For candidate solutions $\mathcal{S}'_f$, SpecGuru generate a pair of test cases (T^1_i, T^2_i) for each S'_{f_i}. If S'_{f_i} successfully passes the differential test, it is deemed as an inlier. Conversely, failure to pass the differential test classifies it as an outlier. The inliers are then grouped based on identical sets of points-to property ϕ instances they contain. The objective is to identify the group (s) whose shared points-to property instance set has the maximum cardinality. If a single group possesses the largest set of property instances, a representative S'_{f_i} is selected from it. When multiple groups have property instance sets of equal size, a representative is randomly chosen from each maximal group. These representatives are merged to form the final candidate solutions, with the merged set containing all property instances from the selected groups. This approach ensures we prioritize specifications that not only have the most property instances, but also exhibit consistency in their property instance composition.

Example 5. As shown in Fig. 4, candidate specifications for the function `cJSON_addItemToObject` are evaluated: S'_{f_1}, S'_{f_2}, S'_{f_3}, S'_{f_4}, and S'_{f_5}. These candidates contain the following points-to property instances: S'_{f_1} and S'_{f_3} include $\{\phi^1_{field}, \phi^2_{field}\}$; S'_{f_2} includes $\{\phi^1_{field}\}$; S'_{f_4} includes $\{\phi^2_{field}\}$; and S'_{f_5} includes $\{\phi^1_{copy}\}$. After differential testing, S'_{f_1}, S'_{f_2}, S'_{f_3}, and S'_{f_4} pass and are classified as inliers, while S'_{f_5} fails due to its incorrect ϕ^1_{copy} property and is labeled as an outlier. The inliers are grouped based on identical sets of property instances: $\{S'_{f_1}, S'_{f_3}\}$ share $\{\phi^1_{field}, \phi^2_{field}\}$, $\{S'_{f_2}\}$ contains $\{\phi^1_{field}\}$, and $\{S'_{f_4}\}$ contains $\{\phi^2_{field}\}$. Since the group $\{S'_{f_1}, S'_{f_3}\}$ has the largest set of property instances (two instances: $\phi^1_{field}, \phi^2_{field}$), SpecGuru selects either S'_{f_1} or S'_{f_3} as the representative for the desired specification $\hat{S}_f$, which best captures the function's behavior.

5 EVALUATION

5.1 Experiment Setup

5.1.1 Research Questions. Our evaluation aims to answer the following research questions:

- **RQ1. (Comparison with SOTA)** Can SpecGuru outperform the state-of-the-art specification generation techniques?
- **RQ2. (Ablation: Bottom-Up Analysis)** How effective is the bottom-up API specification summarization?
- **RQ3. (Ablation: Specification Validation)** How does specification validation affect the performance of SpecGuru?
- **RQ4. (Practical Usefulness)** How do the SpecGuru-generated specifications enhance static analysis in practical applications?

5.1.2 Benchmarks. For our evaluation, we selected 15 widely used third-party C libraries as benchmarks, with their statistics presented in Table 1. These libraries provide specialized functionality beyond the C standard library. We include all 8 libraries from the dataset used by Spectre [27], along with 7 additional libraries we collected

Table 1: Statistics of 15 C third-party libraries. #APIs is the number of function definitions in the library, #LoC is the lines of code, #TSpec is the number of verified specifications, #Stars is the github/gitlab stars.

ID	Library	#Version	#APIs	#LoC	#TSpecs	#Stars
1	libvpx	1.15.2	31	362K	10	934
2	libtiff	4.7.0	172	108K	15	90
3	liblouis	3.34.0	34	37K	4	298
4	cJSON	1.7.15	112	10K	24	10.1K
5	zlib	1.3.1	84	32K	6	6.2K
6	libmagic	5.45	33	2.4K	10	1.2K
7	lcms	2.17	286	45K	19	634
8	cstr	1.0.0	83	18K	10	1.6K
9	parson	1.5.3	144	3.3K	30	1.3K
10	libpcap	1.10.5	84	69K	7	2.9K
11	utf8	1.0.0	38	4K	14	1.6K
12	miniz	3.0.2	175	26K	9	2.1K
13	bstring	1.0.0	130	9K	4	134
14	Jansson	2.14.1	95	13K	22	3.2K
15	lodepng	1.0.0	74	12K	9	2.2K

to expand our benchmark. To ensure representativeness and practical relevance, we only included libraries that either participate in Google's OSS-Fuzz [50] project or have garnered more than 1,000 stars on GitHub, indicating significant community adoption and real-world usage. Furthermore, these libraries have been extensively used as benchmarks in prior research [39, 65] on C library analysis and testing.

5.1.3 Baselines. We compared SpecGuru with several baselines to assess its performance.

- Spectre [27] is a recent fuzzing-based tool designed to automatically generate aliasing specifications for C library APIs, particularly when source code is unavailable. It leverages manually crafted API driver programs to capture aliasing relationships between parameters and return values during fuzzing, producing precise aliasing specifications to enhance static program analysis.
- *c-summary* [55] is a static analysis-based tool built on Infer [21] (a static analysis framework that enables modular analysis of functions). c-summary further processes Infer's results by converting the input/output states of C functions into semantic summaries, enabling automated generation of function specifications from source code. Although other recent automated specification generation techniques exist [9, 10, 17, 58], they are designed specifically for Java and rely on Java language features (RQ1). Consequently, these techniques are not applicable to our experimental context.
- **Isolated analysis (SpecGuru-IA)**: SpecGuru-IA serves as a baseline technique within the SpecGuru framework. It analyzes only the source code of the target function itself, without including any callee functions' source code, to generate specifications. This approach does not incorporate self-validation. (RQ2)
- **Complete code analysis (SpecGuru-CCA)**: SpecGuru-CCA functions as another baseline technique within the SpecGuru framework. It utilizes both the target function's source code and the source code of all its callees to generate specifications. Like SpecGuru-IA, it does not employ self-validation. (RQ2)
- **Bottom-up analysis without validation (SpecGuru-BUA)**: SpecGuru-BUA operates as a baseline within the SpecGuru framework. It employs the same bottom-up analysis strategy as the full SpecGuru implementation but lacks the self-validation component. (RQ2, RQ3)

5.1.4 Ground Truth of Specifications. To the best of our knowledge, there are no existing API specifications for the 15 benchmark C libraries. Therefore, to establish ground truth specifications, we collected all API specifications generated by all methods used in our evaluation and manually verified each specification against the actual source code implementation. We followed Definition 1 to construct the ground truth based on the fundamental pointer relationships. Through careful source code analysis and verification, we identified all correct specifications as the ground truth for each library. To ensure accuracy, the ground truth was cross-checked by a PhD candidate and a postdoctoral researcher. The final ground truth is shown in Table 1 under the column #TSpec. The number of specifications in #TSpec varies across libraries, mainly because we focus on points-to specifications generation between parameters and return values. It is worth noting that some libraries are centered on numerical computation or internal logic and involve few pointer interactions between arguments and return values, leading to fewer relevant specifications.

5.1.5 Evaluation Metrics. We use standard metrics from information retrieval [45] to evaluate the performance of specification generation techniques. If a generated specification matches the ground truth, it is considered a true positive (TP); otherwise, it is a false positive (FP). Specifications in the ground truth that are not generated are counted as false negatives (FN).

5.1.6 Implementation. We implemented SpecGuru using GPT-4-1106-preview [43] for specification generation and Claude-3.5-sonnet-20240620 [4] for test case generation and validation, both with temperature=1.0. The complete pipeline costed approximately $360 in API usage ($130 for specification generation, $230 for validation) and required 10 hours of computational time. Our approach is not tied to specific LLMs; any models with capabilities similar to GPT-4 should work effectively. For evaluating API points-to specifications, we employed a widely-used C/C++ static analysis framework SVF [2, 54] with Andersen's pointer analysis [3], an inclusion-based points-to analysis through constraint resolution. In the specification validation, we compare points-to relations derived from original API source code against those obtained using our specifications through dynamic differential testing. Experiments ran on a 64-bit Linux machine (Ubuntu-22.04) with a 1.8GHz 8-Core AMD Ryzen 7 5700U CPU and 16GB RAM, using OpenAI and Anthropic Claude APIs (500 RPM limit each). We repeated each experiment five times to mitigate randomness.

5.2 RQ1: Comparison with SOTA

Table 2 shows that SpecGuru generates 171 specifications across 15 libraries, which is 21% more than Spectre (141) and 46% more than c-summary (117). Additionally, SpecGuru identifies 33 unique specifications not found by the other two methods.

c-summary fails to generate many specifications due to its inability to handle complex language features. Based on static code analysis, c-summary can only process a subset of C language syntax, struggling with complex features such as arrays, arithmetic operations, and calls to external APIs without source code (Section 5.2

Table 2: The number of specifications generated by Spectre, c-summary, and SpecGuru across libraries. Numbers in parentheses indicate unique specifications not found by the other two methods.

Method	libvpx	libtiff	liblouis	cJSON	zlib	libmagic	lcms	cstr	parson	libpcap	utf8	miniz	bstring	Jansson	lodepng	Total
Spectre	4/(0)	8/(2)	1/(0)	23/(0)	5/(0)	8/(1)	16/(0)	4/(0)	18/(2)	2/(0)	14/(0)	8/(7)	2/(0)	22/(1)	6/(0)	141/(13)
c-summary	5/(2)	6/(0)	1/(0)	14/(0)	6/(0)	7/(0)	14/(1)	5/(1)	17/(2)	4/(1)	11/(0)	2/(0)	2/(1)	17/(0)	6/(0)	117/(7)
SpecGuru	8/(3)	13/(5)	4/(3)	23/(0)	6/(0)	8/(1)	18/(0)	8/(1)	28/(13)	6/(2)	14/(0)	3/(1)	2/(1)	21/(0)	9/(3)	171/(33)

in [30]). For example, consider the function `utf8codepoint` shown in Figure 5 (upper subfigure) for which c-summary fails to generate the specification: this function uses pointer arithmetic (`str += N`) and bitwise operations (ℓ_4-ℓ_5) to decode UTF-8 codepoints. c-summary cannot model these pointer updates and their relation to the return value, failing to infer that the return value points into the same string as the input parameter.

Spectre discovers 14 more specifications than c-summary. However, the limitation of Spectre's dynamic approach is that it can only observe results from executed paths, and cannot analyze corner cases that are not executed to generate specifications, resulting in fewer specifications generated compared to SpecGuru. For example, as shown in the lower subfigure of Figure 5, the `libvpx`'s API `vpx_codec_enc_init_multi_ver` contains 7 conditional branches (ℓ_{16}-ℓ_{20}), and the target field pointer property `ctx->iface=iface;` is nested within 4 levels of control structures `(else, if, for, else)`. This requires satisfying very complex conditions during execution to reach specific code paths.

However, Spectre identifies 13 specifications and c-summary identifies 7 specifications that SpecGuru fails to generate. Among these 20 specifications, 17 were not produced due to the self-validation mechanism conservatively filtering out candidates when the LLM could not generate executable test cases or when tests failed to achieve adequate branch coverage. While this introduces some false negatives, it improves overall generation accuracy and can prevent error propagation during bottom-up generation (detailed analysis in Section 5.4). The remaining 3 specifications involved mutual recursion cases where the combined code exceeded the LLM's context window, causing specification generation failure.

Answer to RQ1: SpecGuru significantly outperforms the state-of-the-art API specification inference tools, generating 21% more specifications than Spectre and 46% more specifications than c-summary across the 15 benchmark libraries. Additionally, SpecGuru identifies 33 unique specifications not found by the other two methods.

5.3 RQ2: Bottom-Up Analysis

Table 3 presents the results of three different strategies—SpecGuru-IA, SpecGuru-CCA, and SpecGuru-BUA—for generating specifications from 15 third-party libraries. The table shows the number of ground-truth specifications (#TSpecs), true positives (#TP), false positives (#FP), and tokens processed under each strategy.

SpecGuru-IA analyzes only the target function's source code without including callees. It identifies 175 (#TP + #FP = 129 + 46) specification candidates, of which 129 are true positives. Its false negatives of 64 out of 193 exceed those of SpecGuru-CCA (58 out of 193) and SpecGuru-BUA (40 out of 193). The false positives of 46 out of 175 for SpecGuru-IA are lower than SpecGuru-CCA (66 out of 201) but higher than SpecGuru-BUA (43 out of 196). Libraries such as `cJSON` and `cstr` frequently employ wrapper functions. When analyzing APIs like `cJSON_GetObjectItem` without access to its callee `get_object_item`, LLMs struggle to identify pointer properties Φ (Definition 1). Despite using fewer tokens (average 393.80), this limited context significantly reduces SpecGuru-IA's effectiveness.

```
1  utf8_int8_t* utf8codepoint (const utf8_int8_t* str,
2                              utf8_int32_t* out_codepoint) {
3    if (0xf0 == (0xf8 & str[0])) {
4      *out_codepoint = ((0x07 & str[0]) << 18) | ((0x3f & str[1]) << 12) |
5                       ((0x3f & str[2]) << 6) | (0x3f & str[3]);
6      str += 4;
7    ...
8    } else {
9      *out_codepoint = str[0];
10     str + =1;
11   }
12   return (utf8_int8_t *) str;
13 }
```

`utf8codepoint`

```
14 vpx_codec_err_t vpx_codec_enc_init_multi_ver (
15                  vpx_codec_ctx_t *ctx,vpx_codec_iface_t *iface, ... ) {
16   if (...) {...}
17   else if (...) {...}
18   // ... (3 more else if branches)
19   else if (...) {...}
20   else {
21     if (...) {
22       for (...) {
23         if (...) {...}
24         else {
25           ...
26           ctx->iface=iface;
27           ...
28 } } } } }
```

`vpx_codec_enc_init_multi_ver`

Figure 5: Case study.

SpecGuru-CCA includes both the target function and all its callees' source code. It generates 201 specification candidates (135 + 66), with 135 correct. While its false negatives decrease to 58, its false positives increase to 66, likely due to LLM hallucination when processing large code volumes. SpecGuru-CCA processes the most tokens among all strategies, averaging 2333.07 tokens per function.

SpecGuru-BUA employs a progressive approach, first summarizing specifications for leaf functions and then analyzing non-leaf functions using these summaries. It identifies 196 specification candidates (153 + 43) with 153 correct specifications, achieving the lowest false negatives (40) while maintaining moderate false positives (43). Processing an average of 608.13 tokens, SpecGuru-BUA requires more computation than SpecGuru-IA but substantially less than SpecGuru-CCA, offering an effective balance between accuracy and efficiency.

Figure 6 also shows that SpecGuru-BUA consistently outperforms the other methods across all evaluation metrics. It achieves the highest precision (0.781 vs. 0.737 for SpecGuru-IA and 0.672 for SpecGuru-CCA), recall (0.793 vs. 0.668 for SpecGuru-IA and 0.699 for SpecGuru-CCA), and F1 score (0.787 vs. 0.701 for SpecGuru-IA and 0.685 for SpecGuru-CCA). While these results establish

Table 3: Comparison of four strategies for specification inference: SpecGuru-IA/CCA/BUA, and SpecGuru. #Diff is the average number of test cases for each validated specification. #Tokens is the average number of tokens in the prompt for generating specifications for each API function.

Strategies	Metrics	Libraries															Average	Total (Precision)
		libvpx	libtiff	liblouis	cJSON	zlib	libmagic	lcms	cstr	parson	libpcap	utf8	miniz	bstring	Jansson	lodepng		
	#TSpecs	10	15	4	24	6	10	19	10	30	7	14	9	4	22	9	12.87	193
SpecGuru -IA	#TP	8	11	4	13	5	9	13	6	19	4	13	5	2	12	5	8.60	129 (73.7%)
	#FP	2	2	0	4	2	0	2	7	3	0	0	9	11	2	2	3.07	46
	#Tokens	723	619	534	187	416	287	528	98	188	573	267	577	289	198	423	393.80	5907
SpecGuru -CCA	#TP	10	12	3	16	4	6	15	6	22	4	13	2	1	14	7	9.00	135 (67.2%)
	#FP	3	5	2	4	1	0	3	7	9	3	0	11	11	4	3	4.40	66
	#Tokens	4132	4986	4218	1197	3432	1166	2783	945	1614	3212	682	1767	1410	465	2987	2333.07	34996
SpecGuru -BUA	#TP	7	12	4	22	7	7	17	7	20	6	14	5	4	14	7	10.20	153 (78.1%)
	#FP	0	2	0	7	1	0	3	5	5	1	1	7	9	0	2	2.87	43
	#Tokens	867	872	643	502	874	626	578	370	504	765	375	775	610	254	507	608.13	9122
SpecGuru	#TP	8	13	4	23	6	8	18	8	28	6	14	3	2	21	9	11.40	171 (91.4%)
	#FP	2	1	0	2	1	0	1	2	1	1	0	0	3	2	0	1.07	16
	#Diff	3.70	2.50	3.65	2.06	4.65	1.80	2.80	2.80	1.90	3.10	1.67	4.53	3.40	3.50	4.20	3.04	42.56
	#Tokens	834	683	543	425	853	646	564	319	662	712	366	790	648	234	536	587.67	8815

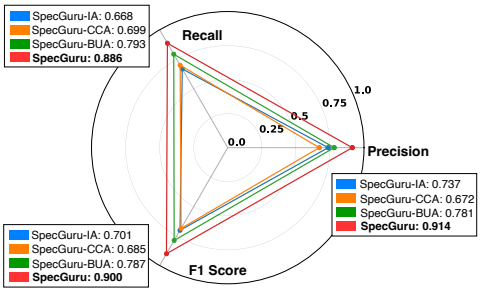


Figure 6: Comparison of four strategies in terms of precision, recall, and F1 Score.

SpecGuru-BUA as the most effective strategy, its lack of validation mechanisms remains problematic. Of its 196 generated specifications, 43 were false positives, underscoring the necessity of validation to ensure accuracy and prevent error propagation.

Answer to RQ2: The bottom-up analysis strategy is better than both isolated analysis and complete code analysis in terms of both quantity and quality of correct specification inference.

5.4 RQ3: Specification Validation

The self-validation mechanism significantly improved precision from 78.1% of SpecGuru-BUA to 91.4% of SpecGuru, as shown in Table 3. This demonstrates that self-validation effectively prevents the propagation and accumulation of erroneous specifications. SpecGuru maintains the highest precision (0.914), recall (0.886), and F1 score (0.900) compared to all other strategies in Figure 6.

For certain libraries (`zlib`, `miniz`, `bstring`), SpecGuru-BUA identifies more true positives than SpecGuru, indicating that the self-validation process occasionally fails to generate valid test cases and thus conservatively excludes some specification candidates. However, self-validation significantly reduces false positives while paradoxically enabling the discovery of more correct specifications in other libraries. This occurs because early error elimination prevents error propagation during the bottom-up summarization.

Among the 16 false positives from SpecGuru that passed differential testing, the primary issue stems from limitations in test-based validation, which occasionally fails to generate sufficiently diverse test cases that would expose incorrect specifications. Despite these false positives, the 171 correct specifications generated by our tool represent 88.6% of all validated specifications in our benchmark.

The validation process required an average of only 3.04 test case generation attempts per specification, demonstrating the efficiency of LLM-based test case generation for specification validation. For libraries with intricate data structures and complex call relationships, even LLMs with strong code comprehension capabilities encounter difficulties. Similar to human test engineers who require domain expertise, LLMs struggle with certain complex codebases. For instance, the `miniz` library—which implements sophisticated compression algorithms with complex data structures—presented significant challenges: although 9 APIs were identified as having property Φ, only 3 passed the validation process due to these inherent complexity barriers.

Answer to RQ3: The specification validation via LLM-generated test cases and differential testing effectively minimized the accumulation and propagation of incorrect specifications during bottom-up specification inference.

5.5 RQ4: Practical Usefulness

To evaluate the practical utility of SpecGuru, we conducted experiments using two widely-used static analysis applications: pointer alias analysis [18] and taint analysis [36]. These experiments assess how effectively the inferred specifications enhance bug detection by identifying additional tainted flows and improve the precision of alias information. The ground truth is defined as the results obtained using the original API source code (referred to as Alias/Taint-Source), which provides the most accurate and complete behavioral information derived directly from the API's implementation. We compared this ground truth against two configurations: (1) SpecGuru-generated API specifications (Alias/Taint-SpecGuru): Specifications inferred by our tool, designed to approximate the behavior of the original APIs. (2) No specifications (Alias/Taint-Empty): A baseline where no additional specifications are provided, relying solely on the analysis tool's default assumptions.

For alias analysis, we inserted alias checks at 193 API call points where ground truth specifications existed. As shown in Figure 7 (upper), Alias-Empty yielded limited alias results due to the absence of API points-to information. Alias-SpecGuru correctly identified aliases for 171 aliases. While Alias-Source covered all 193 alias checkpoints due to having access to the complete API source code, Alias-SpecGuru achieved 88% coverage (171/193) without requiring access to API implementation details.

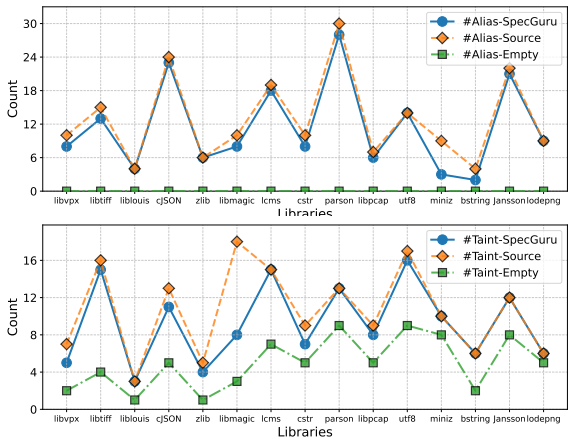


Figure 7: Comparison results of static pointer alias analysis (upper) and taint analysis (lower).

For taint analysis, we inserted source and sink points into the client code and compared three configurations to determine whether taint flows from the sources to the sinks can be identified. Figure 7 (lower) shows that Taint-Empty detected only 74 taint flows, which were flows that did not pass through APIs requiring pointer information. Taint-SpecGuru identified 139 flows, while Taint-Source found 159 flows due to having complete API implementation information. The improvement of Taint-SpecGuru over Taint-Empty demonstrates that SpecGuru-generated specifications significantly enhance the ability of static analysis tools to detect potential security vulnerabilities, even without access to API source code.

Answer to RQ4: The specifications generated by SpecGuru can replace the source code in static pointer analysis. Additionally, taint analysis demonstrates the practical benefits of SpecGuru, as it can enhance the detection of potential security vulnerabilities by providing points-to information.

6 THREATS TO VALIDITY

Datasets. A limitation of our study is that the evaluated open-source libraries may have been included in the training data of the employed LLMs, potentially introducing bias. Ideally, experiments would use novel, unseen libraries. However, specifications for rarely used libraries would provide limited practical value, somewhat mitigating this concern.

Correctness. While LLMs effectively generate specifications for complex libraries, they cannot guarantee soundness or completeness due to issues like hallucinations and limited test coverage. This is especially challenging for APIs with complex object parameters, where generating comprehensive test cases is difficult. Despite these limitations, our method produces more specifications than existing tools with high precision and a low false positive rate. In the future, we will focus on improving test coverage and handling complex parameters.

Model Selection. We used GPT-4 and Claude 3.5, and empirically found that LLMs with capabilities similar to GPT-4 (such as Gemini 1.5 Pro) perform comparably in specification inference and validation tasks. While more advanced models might yield better results, our focus is on the method for automatically generating and validating specifications, rather than the impact of different models.

7 RELATED WORK

Specification Inference. Existing methods for specification inference can be broadly categorized as static or dynamic. Static methods [6, 8, 14, 30, 37, 47, 51, 68] infer specifications by analyzing the source code's structure, variable types, and flow patterns. Dynamic methods [9, 10, 17, 27, 42], on the other hand, infer specifications by observing program behavior during execution using techniques like runtime monitoring and profiling. Some methods also leverage machine learning to infer specifications, such as unsupervised API aliasing inference [19] and semi-supervised taint specification inference[16]. The types of specifications inferred can range from taint analysis[17, 52], which monitors sensitive data flow, to points-to analysis[9, 10], which identifies potential object references for pointer variables. Many specification inference works focus on Java API specification inference, whereas our work targets C API specification inference. The distinct language features between Java and C present different challenges for specification generation. The efforts most relevant to our work, which target C API specification generation, are the static analysis-based c-summary [30] and the dynamic analysis-based (Fuzzing) Spectre [27]. However, c-summary struggles with complex language features, while Spectre fails to handle corner cases, limiting their ability to generate comprehensive C library specifications.

Automatic Test Case Generation. Automated test case generation can significantly cut down the time and resources spent on manual test case creation. Prior research in this area, mainly employing search-based heuristics [5, 24, 38, 44] or pre-trained language models like BART [31, 46, 57], has its drawbacks. While search-based heuristic methods often struggle with the diversity and volume of the test cases they generate, pre-trained language models usually necessitate extensive training and labeled data, which might not be easily accessible for different properties or new datasets.

LLM for Static Program Analysis. Recent research has leveraged LLMs for various static program analysis tasks, including verification strategy selection [53], software specification inference [29, 40], and loop invariant generation [35, 63]. Approaches like [59, 60] integrate LLMs with external tools to enhance dataflow analysis and bug detection. These orthogonal research directions and their innovations combining formal analysis with LLMs' code comprehension capabilities inspired our SpecGuru design.

8 CONCLUSION

This paper presents SpecGuru, a novel approach that automatically generates API points-to specifications using Large Language Models. Our bottom-up method optimizes LLM usage by providing callees' information without excessive code length. SpecGuru introduces an automated self-validation strategy that validates specification candidates through LLM-generated test cases and differential testing. Experiments on 15 popular third-party C libraries demonstrates that our approach significantly outperforms baseline methods in both the quantity and quality of inferred specifications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their reviews and suggestions. This work is supported by ARC Grants DP250101396 and DE250100192, and a Gift Grant from Google.

REFERENCES

[1] 2025. cJSON. https://github.com/DaveGamble/cJSON

[2] 2025. SVF: Static Value-Flow Analysis Framework for Source Code. https://github.com/SVF-tools/SVF.

[3] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. *PhD Thesis, DIKU, University of Copenhagen* (1994).

[4] Anthropic. 2024. Claude AI. https://claude.ai

[5] Andrea Arcuri. 2017. Many independent objective (MIO) algorithm for test suite generation. In *Search Based Software Engineering: 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings 9*. Springer, 3–17.

[6] 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*. 725–735.

[7] Owura Asare, Meiyappan Nagappan, and N Asokan. 2023. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* 28, 6 (2023), 129.

[8] Osbert Bastani, Saswat Anand, and Alex Aiken. 2015. Specification inference using context-free language reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 553–566.

[9] 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*. 678–692.

[10] Osbert Bastani, Rahul Sharma, Lazaro Clapp, Saswat Anand, and Alex Aiken. 2019. Eventually sound points-to analysis with specifications. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[11] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *International Conference on Learning Representations*.

[14] Xiao Cheng, Jiawei Ren, and Yulei Sui. 2024. Fast Graph Simplification for Path-Sensitive Typestate Analysis through Tempo-Spatial Multi-Point Slicing. *Proceedings of the ACM on Software Engineering* 1, FSE, Article 23 (2024).

[15] Xiao Cheng, Jiawei Wang, and Yulei Sui. 2024. Precise Sparse Abstract Execution via Cross-Domain Interaction. In *46th International Conference on Software Engineering (ICSE '24)*. ACM/IEEE.

[16] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 760–774.

[17] Lazaro Clapp, Saswat Anand, and Alex Aiken. 2015. Modelgen: mining explicit information flow specifications from concrete executions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 129–140.

[18] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 106–117. https://doi.org/10.1145/277650.277670

[19] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*. 745–759.

[20] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. 2019. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems* 32 (2019).

[21] Facebook. 2015. Infer Static Analyzer. https://fbinfer.com/. Accessed: 2025-07-01.

[22] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.

[23] Martin A Fischler and Robert C Bolles. 1981. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM* 24, 6 (1981), 381–395.

[24] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[25] Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. 2023. Large language models and simple, stupid bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 563–575.

[26] Shuangxiang Kan, Yuhao Gao, Zexin Zhong, and Yulei Sui. 2024. Cross-Language Taint Analysis: Generating Caller-Sensitive Native Code Specification for Java. *IEEE Transactions on Software Engineering* (2024).

[27] Shuangxiang Kan, Yuekang Li, Weigang He, Zhenchang Xing, Liming Zhu, and Yulei Sui. 2025. Spectre: Automated Aliasing Specifications Generation for Library APIs with Fuzzing. *ACM Transactions on Software Engineering and Methodology* (2025).

[28] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. *ACM SIGPLAN Notices* 48, 6 (2013), 423–434.

[29] Thanh Le-Cong, Bach Le, and Toby Murray. 2025. Can LLMs Reason About Program Semantics? A Comprehensive Evaluation of LLMs on Formal Specification Inference. *arXiv preprint arXiv:2503.04779* (2025).

[30] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 127–137.

[31] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[32] Tsz On Li, Wen yi Zong, Yibo Wang, Haoye Tian, Y. Wang, and Shing Chi Cheung. 2023. Nuances are the Key: Unlocking ChatGPT to Find Failure-Inducing Tests with Differential Prompting. *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2023), 14–26. https://api.semanticscholar.org/CorpusID:258298446

[33] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[34] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. 2024. LLM-Powered Test Case Generation for Detecting Tricky Bugs. *arXiv preprint arXiv:2404.10304* (2024).

[35] Ruibang Liu, Guoqiang Li, Minyu Chen, Ling-I Wu, and Jingyu Ke. 2024. Enhancing Automated Loop Invariant Generation for Complex Programs with Large Language Models. *arXiv preprint arXiv:2412.10483* (2024).

[36] Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*. https://www.usenix.org/legacy/events/sec05/tech/full_papers/livshits/livshits.pdf

[37] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. 2009. Merlin: Specification inference for explicit information flow problems. *ACM Sigplan Notices* 44, 6 (2009), 75–86.

[38] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[39] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3793–3807.

[40] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2024. Specgen: Automated generation of formal program specifications via large language models. *arXiv preprint arXiv:2401.08807* (2024).

[41] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[42] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing class specifications. In *Proceedings of the 44th International Conference on Software Engineering*. 1008–1020.

[43] OpenAI. 2024. ChatGPT. https://openai.com/chatgpt

[44] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[45] David M. W. Powers. 2011. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation. *Journal of Machine Learning Technologies* 2, 1 (2011), 37–63.

[46] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[47] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static specification inference using predicate mining. *ACM SIGPLAN Notices* 42, 6 (2007), 123–134.

[48] Baptiste Roziere, Jie M Zhang, Francois Charton, Mark Harman, Gabriel Synnaeve, and Guillaume Lample. 2021. Leveraging automated unit tests for unsupervised code translation. *arXiv preprint arXiv:2110.06773* (2021).

[49] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*. 1559–1575.

[50] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).

[51] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. 174–184.

[52] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting taint specifications for javascript libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 198–209.

[53] Jie Su, Liansai Deng, Cheng Wen, Shengchao Qin, and Cong Tian. 2024. CFStra: Enhancing Configurable Program Analysis Through LLM-Driven Strategy Selection Based on Code Features. In *International Symposium on Theoretical Aspects of Software Engineering*. Springer, 374–391.

[54] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[55] Sungho Lee. 2020. https://github.com/SunghoLee/c-summary.

[56] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1093–1105. https://doi.org/10.1145/3597926.3598120

[57] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[58] Chengpeng Wang, Jipeng Zhang, Rongxin Wu, and Charles Zhang. 2024. DAInfer: Inferring API Aliasing Specifications from Library Documentation via Neurosymbolic Optimization. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2469–2492.

[59] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. LLMDFA: Analyzing Dataflow in Code with Large Language Models. *Advances in Neural Information Processing Systems* 37 (2024), 131545–131574.

[60] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, and Xiangyu Zhang. 2024. Sanitizing Large Language Models in Bug Detection with Data-Flow. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 3790–3805.

[61] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).

[62] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting program specification synthesis by large language models using static analysis and program verification. *arXiv preprint arXiv:2404.00762* (2024).

[63] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 406–417.

[64] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764* (2023).

[65] Hanxiang Xu, Wei Ma, Ting Zhou, Yanjie Zhao, Kai Chen, Qiang Hu, Yang Liu, and Haoyu Wang. 2025. CKGFuzzer: LLM-Based Fuzz Driver Generation Enhanced By Code Knowledge Graph. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 243–254.

[66] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. CoRR abs/2305.04207 (2023). *arXiv preprint arXiv:2305.04207* 10 (2023).

[67] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. [n. d.]. LPR: Large Language Models-Aided Program Reduction. ([n. d.]).

[68] Haiyan Zhu, Thomas Dillig, and Isil Dillig. 2013. Automated inference of library specifications for source-sink property verification. In *Programming Languages and Systems: 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings 11*. Springer, 290–306.