International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-025-00780-7

GENERAL

Special Issue: SOAP 2024

# Interleaving static analysis and LLM prompting with applications to error specification inference

**Patrick J. Chapman**[1] · **Cindy Rubio-González**[1] · **Aditya V. Thakur**[1]

Accepted: 23 January 2025


**Abstract**

This paper presents a new approach to improve static program analysis using Large Language Models (LLMs). The approach *interleaves* calls to the static analyzer and queries to the LLM. The query to the LLM is constructed based on intermediate results from the static analysis, and subsequent static analysis uses the results from the LLM query. We apply our approach to the problem of *error-specification inference*: given systems code written in C, infer the set of values that each function can return on error. Such error specifications aid in program understanding and can be used to find error-handling bugs. We implemented our approach by incorporating LLMs into EESI, the state-of-the-art static analysis for error-specification inference. Compared to EESI, our approach achieves higher recall (from an average of 52.55% to 77.83%) and higher F1-score (from an average of 0.612 to 0.804) while maintaining precision (from an average of 86.67% to 85.12%) on real-world benchmarks such as Apache HTTPD and MbedTLS. We also conducted experiments to understand the sources of imprecision in our LLM-assisted analysis as well as the impact of LLM nondeterminism on the analysis results.

**Keywords** Static analysis · Large language model · Error handling · Error specifications

## 1 Introduction

This paper presents a new approach for using *Large Language Models (LLMs)* to improve static program analysis. LLMs [22, 35] have been shown to demonstrate impressive reasoning abilities in natural and programming languages tasks via few-shot [3] and chain-of-thought [38] prompting. The approach presented in this paper utilizes this reasoning ability of LLMs when the static analysis is unable to make progress; the results of the query to the LLM are used for subsequent analysis. Furthermore, the query (or prompt) to the LLM incorporates the current results of the static analysis, which enables it to provide more accurate results. In this way, our approach *interleaves* calls to the static analyzer and the LLM, with each utilizing the results of the other.

✉ P.J. Chapman
pchapman@ucdavis.edu

C. Rubio-González
crubio@ucdavis.edu

A.V. Thakur
avthakur@ucdavis.edu

[1] University of California, Davis, 1 Shields Ave., Davis, CA, USA

We apply this novel approach to the problem of *error-specification inference* of functions in systems code written in C, i.e., inferring the set of values returned by each function upon error (Sect. 2). The C language does not have built-in exception or error handling; thus, a common idiomatic practice for error handling is to check the return value of a function on error, i.e., the *return code idiom*. These return values indicate the functions' error specifications, which can aid in program understanding, as well as in finding error-handling bugs. EESI [9] has shown higher effectiveness and performance at inferring error specifications compared to prior approaches [1, 10, 17]. Our approach interleaves calls to the EESI static analyzer and the LLM (Fig. 1).

We evaluated our approach on six real-world C programs, such as MbedTLS and zlib (Sect. 4). Our approach improves recall and F1-score over EESI from 52.55% to 77.83% and 0.612 to 0.804, respectively, while maintaining a high precision of 85.12% compared to 86.67% in EESI. Our evaluation demonstrates that by interleaving static analysis and LLM prompting, we can significantly improve upon the error specification inference capabilities of just a static analyzer.

The contributions of this paper are as follows:

– We propose a technique for interleaving a static analysis with LLM prompting.

Published online: 04 February 2025

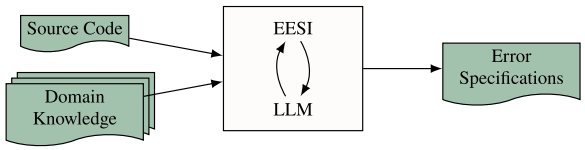



**Fig. 1** Our approach infers error specifications by interleaving calls to the EESI static analyzer and the LLM

- We designed a tool for error specification inference of C programs using our approach of combining the EESI static analyzer and LLM prompts.
- We evaluate our approach on 6 real-world C programs comparing it with prior state-of-the-art EESI. We provide an ablation study on the individual components of our approach. In addition, we also evaluate potential sources of imprecision in our interleaved analysis.

**Comparison to previous work** An earlier version of this paper was published at the *International Workshop on the State of the Art in Program Analysis (SOAP) 2024* [5]. In comparison with our prior work, this paper extends our interleaved analysis to track sources of inference (Sect. 3.3). We use sources of inference to identify the sources of imprecision in our interleaved analysis (Sects. 4.6 and 4.7), which reveal new key findings related to:

- The correlation between providing incorrect background knowledge error specifications and inferring error specifications incorrectly;
- The correlation between background knowledge from the LLM and incorrect error specification inference;
- The relationship between the number of calls to an LLM on an inference path and incorrect error specification inference;
- How often the incorrectly inferred error specifications are related to the $\emptyset$ value;
- How nondeterminism from the LLM affects the results of the analysis.

## 2 Background and motivating example

This section provides background on error specification inference for C programs, the state-of-the-art for error specification inference with focus on $EESI$, background on LLMs, and an example that motivates the need for an interleaved analysis for error specification inference.

### 2.1 Error specification inference for C programs

The C language does not feature programming constructs for exception handling. Instead, developers often use the *return code idiom* to indicate error. An *error specification* refers to the set of values returned by a function upon error. Because it is not possible to enforce compile-time rules regarding error code propagation and checking, the return code idiom often leads to error-handling bugs, e.g., developers may miss or incorrectly check the error return values of functions.

A few approaches for error specification inference have been presented [1, 9–11, 17, 40]. In this paper, we consider a state-of-the-art static program analysis that uses abstract interpretation for error specification inference named EESI [9].

### 2.2 EESI analysis for error specification inference

The EESI static analysis takes as input multiple forms of optional user-supplied *initial domain knowledge*: (1) initial error specifications, (2) error codes, (3) success codes, and (4) error-only functions (those only called in error paths). With this initial domain knowledge, EESI uses static analysis to infer new error specifications.

EESI incorporates three intraprocedural analyses and one interprocedural analysis:

- **Function call constraints analysis** finds constraints on function return values that must be met in order to execute a statement.
- **Returned constant value analysis** finds the constants that must be returned if a statement has executed.
- **Returned function call analysis** finds the call return values that must be returned if a statement has executed.
- **Error specification inference engine** facilitates the interprocedural analysis used to determine the error specification of a function based on a set of rules.

The *error specification inference engine* comprises five rules used for error specification inference. These rules rely of the facts learned from the three intraprocedural analyses and use propagated facts from the domain knowledge to infer error specifications:

- **Initial specification rule.** The analysis abstracts the error specification $\beta(f)$ from the initial specification supplied as domain knowledge.
- **Error code rule.** If a function returns an error code constant value $c$ from the supplied domain knowledge $EC$, then it is added as an error value.
- **Error only function rule.** If a function $f$ calls some constant error-only function $f_{eo}$ from the supplied domain knowledge and then some constant $c$ is returned, then $c$ is added as an error value.
- **Error constant return rule.** If a function $f$ returns some constant $c$ when a function call to $f'$ returns an error, then $c$ is added as an error value.
- **Function call propagation rule.** If a function $f$ returns a call to $g$ along an error path of $f'$, then the error values of $g$ are added to $f$. Note that $f'$ and $g$ may be the same function.

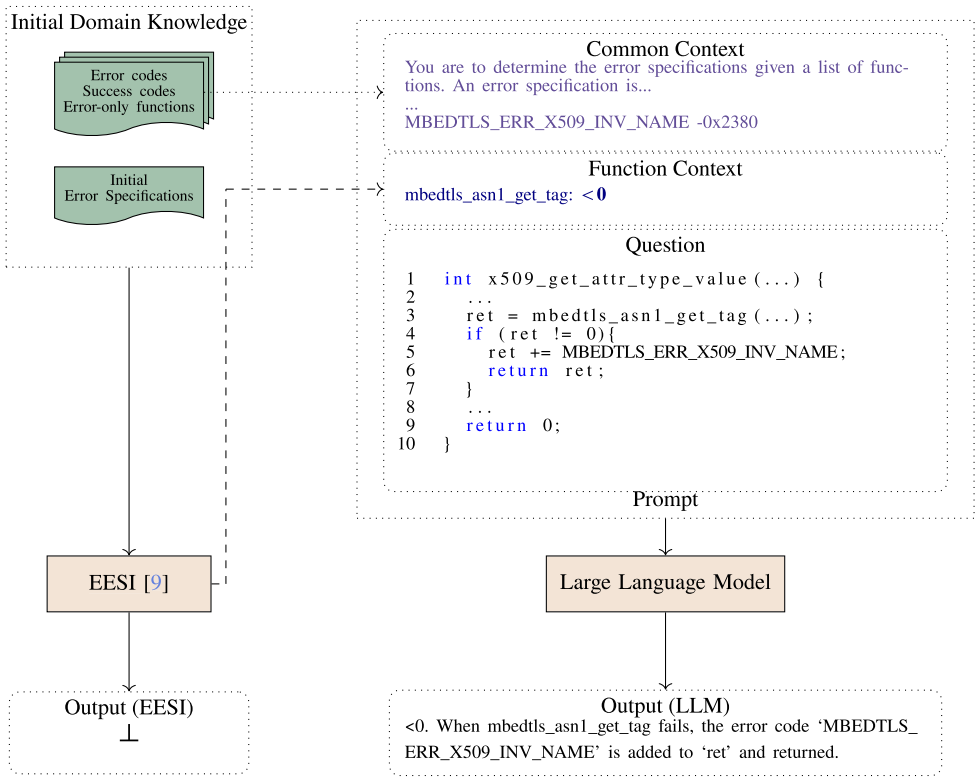



**Fig. 2** Using EESI and the LLM to infer error specifications in MbedTLS

While EESI has demonstrated success in error specification inference, it has inherent limitations due to the fact that *incomplete program facts* can affect the recall and precision of error specification inference. EESI provides approximations that may be insufficient in learning enough program facts. Furthermore, EESI is unable to reason about *third-party functions*, i.e., functions whose definition is not found within the program under analysis.

## 2.3 Large language models (LLMs)

LLMs are language models trained on large amounts of data for tasks such as text generation and language understanding. These models have been developed for both natural language [35] and programming languages [32], while some models are trained for both [22, 29, 36]. One of the key components of LLMs are the *prompts*, i.e., the input to the LLM. There has been considerable research done in recent years related to the generation of prompts that improve the performance of LLMs in various tasks [2, 29, 37, 38]. These approaches include concepts such as *chain-of-thought* [38], where LLMs are given question and answer as examples with the associated chain-of-though reasoning, and *self-consistency* [37] prompting, where LLMs are prompted with the same question multiple times, using the most consistent answer given.

## 2.4 A motivating example

This section illustrates our approach of interleaving calls to the EESI static analyzer and the LLM to infer error specifications. Consider the function `x509_get_attr_type_value` in MbedTLS. EESI alone is unable to infer its error specification $<\mathbf{0}$; EESI infers $\bot$ as the function error-specification as shown in Fig. 2.

The LLM alone is also unable to infer the error specification. We can construct a prompt to the LLM that includes the general description of the error specification inference problem (Common Context in Fig. 2), as well as the source code of the function (Question in Fig. 2). However, querying the LLM with just this information is not sufficient to give us the correct error specification $<\mathbf{0}$. In particular, the LLM infers that the error condition for `mbedtls_asn1_get_tag` is $\neq\mathbf{0}$ from the conditional check. Even when the value of the error code `MBEDTLS_ERR_X509_INV_NAME` is included in the Common Context, the incorrect assumption about the called function leads the LLM to incorrectly infer that the return value on the error path is the negative error code added with any nonzero value, that is, the LLM infers that the error value could be anything, and the error specification is $\top$, instead of $<\mathbf{0}$.

However, if we also include intermediate results from *EESI* in the LLM prompt, then the LLM is able to return the fact that `x509_get_attr_type_value` returns a

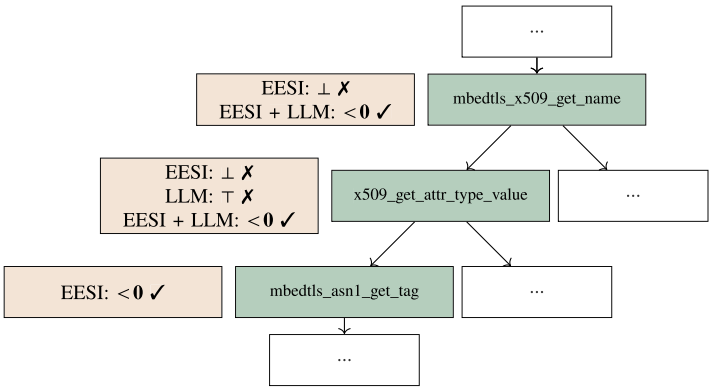



**Fig. 3** Interleaving the static analyzer EESI and LLM results in a new chain of correctly inferred error specifications compared to EESI for MbedTLS

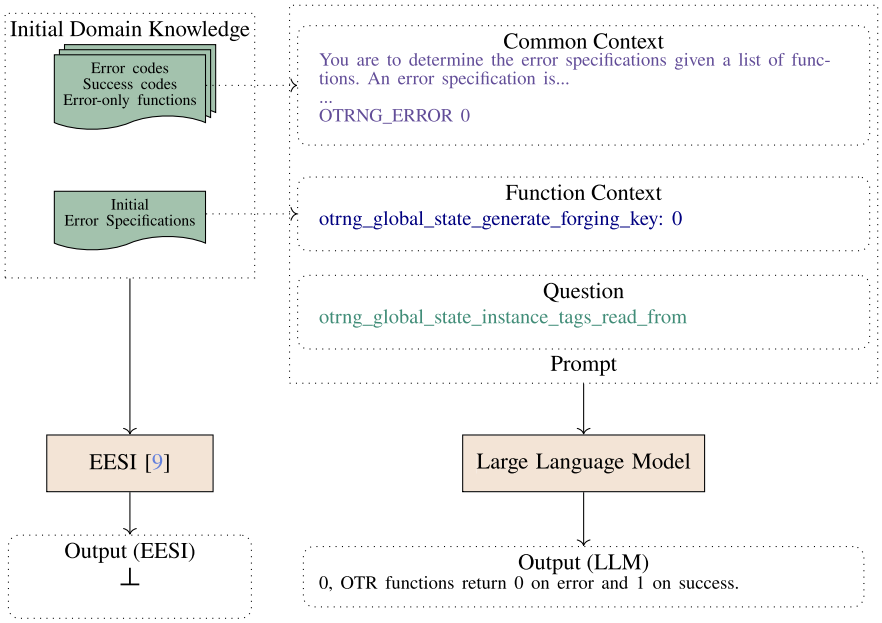



**Fig. 4** Using the LLM to infer the error specification of a third-party function called from Pidgin OTRv4

value <**0** on error. In particular, the LLM prompt includes the error specification of the function `mbedtls_asn1_get_tag` that is called from `x509_get_attr_type_value` (Function Context in Fig. 2); this error specification is inferred by the EESI static analyzer.

This example illustrates how our approach provides benefits over purely static analysis or LLM approaches by interleaving calls to the static analyzer and the LLM: the LLM is used only when the static analyzer is unable to make progress, and the LLM prompt includes intermediate information gleaned by the static analyzer. Furthermore, the output of the LLM is fed back into the EESI static analyzer. For example, the LLM's specification for `x509_get_attr_type_value` would allow EESI to subsequently find the error specification <**0** for `mbedtls_x509_get_name` from analyzing its implementation:

```
if((ret = x509_get_attr_type_value(...)) != 0)
    return ret;
```

We depict this inference chain in Fig. 3, where we demonstrate how leveraging the combination of EESI and the LLM can lead to entirely new chains of error specifications inferred in the call graph.

The specifics about the LLM prompt construction, viz, Common Context, Function Context, and Question, are deferred to Sect. 3.1.

Figure 4 shows another scenario that illustrates the benefits of an interleaved analysis. The function `otrng_global_state_instance_tags_read_from` is a third-party function called in Pidgin OTRv4. Because the source code of this function is not available, EESI is unable to infer its error specification and, consequently, it might not be able to infer the specifications of functions that call it. However, constructing an LLM prompt that includes information from the user-provided domain knowledge, the LLM is able to correctly infer the error specification for `otrng_global_state_instance_tags_read_from`.

## 3 Approach

We illustrate our approach for interleaving static analysis and LLMs in Fig. 1. The input is the program source code and optional domain knowledge, and the output is the function error specifications inferred by the analysis.

### 3.1 Building prompts

When interacting with the LLM, we construct a prompt that consists of the *Common Context*, *Function Context*, and *Question*, as mentioned in Sect. 2.4.

**Common context** The prompt *Common Context* used for error specification inference consists of a problem description and an explanation of the abstract domain used by the EESI static analyzer. We provide the explanation of the abstract domain, because we want the LLM to output its learned error specifications using this domain. Relating to the program under analysis, the *Common Context* also contains any error codes, success codes, and error-only functions from the domain knowledge input. We include additional observed idiomatic practices related to the return code idiom:

1. Error specification values must be a subset of the returned values of a function.
2. Unknown error specifications are $\perp$.
3. Success values are not part of the error specification.
4. The NULL return value is equal to 0.
5. Error codes from standard library functions are positive integers.
6. Macros may check return values and return if failing.

We also provide basic *chain-of-thought examples* that consist of a function definition and its associated error specification, with a chain-of-thought explanation. We do so to demonstrate the task of error specification inference and so that the LLM generates parse-able output. We do this, in addition to providing the explanation of the abstract domain, in order to limit the LLM from generating output that is unexpected. However, if the LLM output does not follow the expected format, then the related error specification will consist of the $\perp$ element, i.e., unknown. For example, the expected output for `malloc` would be *malloc: 0*.

**Function context** This prompt relates to any relevant function error specifications for the function that is being queried by the LLM. The *Function Context* that is generated depends on the selected LLM query function that will be explained further when introducing our algorithm, Algorithm 1. In all cases, these error specifications are provided as *few-shot examples* to the LLM, with the aim to generate parse-able output, as well as provide demonstrative examples to the LLM. These error specifications provide additional context that can assist the LLM when it comes to understanding returned error values. This is especially true when there are functions that exist in the same library as demonstrated with Fig. 2.

**Algorithm 1:** InferErrorSpecification($P$, $F$)

**INPUT:** Map of program facts $P$, set of functions $F$.
**OUTPUT:** Updated $P$ with new error specifications.

```
1:  CG ←CallGraph(F)
2:  for all f ∈ reverseTopologicalSort(CG) do
3:      if isThirdParty(f) then
4:          spec ←queryLLMThirdParty(P, f)
5:      else
6:          spec ←runAnalysis(P, f,EESI )
7:          if spec = ⊥ then
8:              spec ←queryLLMAnalysis(P, f)
9:          end if
10:     end if
11:     P ←updateFacts(P, f, spec)
12: end for
13: return P
14: Function queryLLMAnalysis(P, f)
15:     question ←getSourceCode(f)
16:     functionContext ← getCalledErrorSpecs(P, f)
17:     prompt ←buildPrompt(functionContext, question)
18:     spec ←parseOutput(queryLLM(prompt))
19:     return spec
20: EndFunction
21: Function queryLLMThirdParty(P, f)
22:     question ← getName(f)
23:     functionContext ←getErrorSpecs(P)
24:     prompt ←buildPrompt(functionContext, question)
25:     spec ← parseOutput(queryLLM(prompt))
26:     return spec
27: EndFunction
```

**Question** The *Question* in all constructed prompts asks for the LLM to return any error specification that it is confident in using the abstract domain used by EESI.

### 3.2 Error specification inference

For the task of error specification inference, we present Algorithm 1 to demonstrate how the static analyzer and LLM are used. Our algorithm takes as input the domain knowledge as a map of program facts $P$ and the set of functions from the source code $F$. The algorithm returns the updated facts $P$ after performing analysis.

The analysis begins by iterating over the functions $f \in F$ bottom-up in the Call Graph ($CG$) as demonstrated on Line 2. This ensures that called functions are inferred before their caller, because called functions provide additional context for error specification inference. Note that, for brevity, we

do not include in the algorithm that we perform a fixpoint on the Strongly Connected Components (SCC) in $CG$, as recursion may exist in the call chains. The algorithm will attempt to infer an error specification in one of three cases: (1) queryLLMThirdParty (Line 4), (2) runAnalysis (Line 6), or (3) queryLLMAnalysis (Line 8).

#### 3.2.1 Third-party function error specifications

For each function, we first check if it is a third-party function (Line 3). We perform queryLLMThirdParty (Line 4) in that case. Because the source code definition is not available for third-party functions, we cannot statically analyze it. As *Function Context* for the prompt, we provide the entire set of error specifications that are in $P$ on Line 23. The *Question* in this case just simply lists the name of the function of interest (Line 22). The LLM is then queried, where the output is then parsed (Line 25) and if any error specification is learned, the program facts are updated (Line 11).

#### 3.2.2 Error specification analysis

If the function is not third-party, then the EESI static analyzer will perform its own analysis. EESI will determine if the error specification of the function is infallible ($\emptyset$), unknown ($\perp$), or any other value (e.g., $<\mathbf{0}$) from runAnalysis on Line 6. If this result is $\perp$ (Line 7), then we query the LLM once for the function under analysis with queryLLMAnalysis on Line 8.

We only provide the known error specifications of functions called within the function definition (Line 16), unlike the *Function Context* provided in queryLLMThirdParty. We demonstrate an example of this in Fig. 2, where EESI infers an error specification for `mbedtls_asn1_get_tag`, which is provided as *Function Context* to the LLM, which results in the correct inference of the error specification for `x509_get_attr_type_value`.

Finally, the constructed *Question* as part of the prompt consists of the source code of the function being analyzed (Line 15), and the resulting output from the LLM is then parsed (Line 18) with any newly inferred error specification updated in the program facts (Line 11).

#### 3.2.3 Validating the LLM response

We requery the LLM for every generated prompt to limit the side effects of *hallucination*. Hallucination refers to when LLMs make up information to satisfy a prompt, even if the provided *chain-of-thought* reasoning is contradictory. We specifically ask the LLM to ensure that the error specifications provided match the given chain-of-thought description from itself. Additionally, we also limit some of the imprecision by identifying two inconsistencies with formal reasoning. First, we do not infer error specifications if the resulting error value from the LLM includes a known success value. Second, we do not infer an error specification if the LLM states that the error specification is an improper superset of the return range of the function. Third, we explicitly re-prompt the LLM and ask it to remove any success values from the error specification and any error specifications that it is not confident in. These particular success values may not be known as part of the interleaved analysis, but the LLM may still understand they are success values even if they misreport it as part of the error specification. As the program facts are obtained via an approximation during the analysis of EESI and that hallucinations are a very general problem for LLMs, we cannot guarantee that these inconsistencies are removed entirely, but we can utilize these rules to limit low-hanging fruit.

### 3.3 Tracking sources of inference

Error specifications can be inferred from initial domain knowledge given to $EESI$, from common and function context provided to the LLM, and/or from specifications inferred during the interleaved analysis. We refer to all these sources of inference as *background knowledge*. In order to better understand what our analysis leverages during inference, we track the following information:

- Whether the specification was directly inferred by either the static analyzer, or the LLM;
- The error specifications (if any) that were used as background knowledge or context to infer the target specification;
- The length of the specification's inference path, i.e., the number of functions along the call graph whose own specifications contributed to inferring the target specification;
- The number of LLM calls made along the specification's inference path.

We collect the above information by tracking the background knowledge functions that contribute to the target function's error specification inference along with the values that the background knowledge contributed to for inference. For the static analyzer, this means that we track the error specifications from called functions that contributed to error specification inference via inference rules (Sect. 2.1). For the LLM-based inference, we track the entire set of error specifications given as context, and associate this context with the inferred values given by the LLM. Unlike the static analyzer, it is infeasible to reason about what background knowledge error specifications actually contributed to the error specification inference. This is because we would have to determine the minimal subset of knowledge that would lead to the correct error specification inferred, which there may be multiple valid subsets.

**Table 1** Selected benchmarks with their LOC and selected domain knowledge – initial error specifications, error-only (EO) functions, error codes, and success codes

| Benchmark | KLOC | Ver. | Domain knowledge | | | |
|---|---|---|---|---|---|---|
| | | | Init. specs | EO | Codes | |
| | | | | | Error | Success |
| Apache HTTPD | 288 | 2.4.46 | 14 | 0 | 44 | 1 |
| LittleFS | 2 | 1.7.0 | 11 | 0 | 14 | 1 |
| MbedTLS | 255 | 2.21.0 | 21 | 1 | 221 | 1 |
| Netdata | 51 | 1.11.0 | 43 | 0 | 0 | 0 |
| Pidgin OTRv4 | 15 | 4.0.2 | 34 | 0 | 0 | 0 |
| zlib | 18 | 1.2.11 | 7 | 0 | 6 | 1 |

By tracking background knowledge used as sources of inference for each error specification, we can also track the inference path length. That is, we track how the inference proceeds across the call graph, tracking what background knowledge functions along the call graph can contribute to error specifications inferred later in the analysis. As we are tracking if error specifications are inferred from EESI or the LLM, we can also track the number of LLM calls along an inference path.

## 4 Experimental evaluation

For our experimental evaluation, we perform an ablation study. We propose five research questions with one baseline to target components of our approach:

**RQ0** How well does the static analysis of EESI perform? This is our baseline.
**RQ1** What is the impact of using the LLM to infer third-party error specifications, i.e., queryLLMThirdParty?
**RQ2** What is the impact of using the direct LLM analysis, i.e., queryLLMAnalysis?
**RQ3** What is the impact of interleaving EESI and the LLM?
**RQ4** What are the sources of imprecision of the interleaved analysis?
**RQ5** What is the performance cost in using the LLM in the interleaved analysis?

Our code and data are publicly available at https://github.com/ucd-plse/eesi-llm.

### 4.1 Experimental setup

**Benchmarks** We consider a data set of six benchmark programs that represent a variety of error-handling patterns and system types, as listed in Table 1.

**Table 2** Total number of functions, functions in $\mathcal{G}$, and third-party functions in $\mathcal{G}$

| Benchmark | Total | $\mathcal{G}$ | Third party $\in \mathcal{G}$ |
|---|---|---|---|
| Apache HTTPD | 1210 | 600 (49.6%) | 135 (22.5%) |
| Little FS | 60 | 60 (100.0%) | 9 (15.0%) |
| MbedTLS | 1211 | 598 (49.4%) | 15 (2.5%) |
| Netdata | 720 | 338 (47.6%) | 74 (21.9%) |
| Pidgin OTRv4 | 277 | 277 (100.0%) | 200 (72.2%) |
| zlib | 126 | 126 (100.0%) | 10 (7.9%) |

**Domain knowledge** For all approaches, we supply the same initial domain knowledge as input. *Initial error specifications* are identified via one of two strategies. The first is that we select applicable error specifications from a list of common and well-known standard library functions. The second is that we manually inspect a small subset of functions based on the program's call graph, supplying functions that appear lower in the call graph as initial domain knowledge. *Success* and *error codes* are mined automatically through pattern matching header files for patterns such as `ERR`, `err`, and `SUCCESS`. *Error-only functions* (only called on error paths) are selected via manual inspection. The manual effort involved in finding the above domain knowledge for all benchmarks took a total of one hour.

**Evaluation metrics and ground truth** We measure precision, recall, and F1 (F1-score) – where we only consider a true positive (TP) to be a learned error specification that matches the ground truth exactly; for example, $\leq \mathbf{0}$ and $< \mathbf{0}$ are not equivalent and would be considered a false positive (FP). If the analysis determines an error specification is unknown $\perp$, then that is considered a false negative (FN). As every function-under-analysis will have an error specification, even infallible $\emptyset$ functions, we do not have true negatives (TN). For all metrics, we calculate based on a manually inspected ground-truth $\mathcal{G}$ as depicted in Table 2. For smaller benchmarks, we inspected all functions, but for larger benchmarks we randomly sampled a subset. We did so, as manual inspection over all functions is not feasible due to time constraints, as some functions may consist of hundreds or thousands of lines. Note, numbers represented in Table 2 do not count initial error specifications from the domain knowledge.

*Precision*, *recall*, and *F1* are defined as:

$$Precision = \frac{TP_{\mathcal{G}}}{TP_{\mathcal{G}} + FP_{\mathcal{G}}}, \qquad Recall = \frac{TP_{\mathcal{G}}}{TP_{\mathcal{G}} + FN_{\mathcal{G}}},$$

$$F1 = \frac{2 * Precision * Recall}{100 * (Precision + Recall)}.$$

**Table 3** Specification counts, precision, recall, and F1-score for EESI

| Benchmark | $<0$ | $>0$ | 0 | $\leq 0$ | $\geq 0$ | $\neq 0$ | $\emptyset$ | Total | Precision | Recall | F1 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Apache HTTPD | 16 | 42 | 16 | 0 | 1 | 27 | 183 | 285 | 94.16% | 37.56% | 0.537 |
| Little FS | 40 | 0 | 7 | 0 | 0 | 0 | 10 | 57 | 91.30% | 75.00% | 0.824 |
| MbedTLS | 723 | 10 | 48 | 3 | 0 | 1 | 246 | 1031 | 90.64% | 84.55% | 0.875 |
| Netdata | 17 | 35 | 108 | 0 | 1 | 1 | 116 | 278 | 64.60% | 24.50% | 0.355 |
| Pidgin OTRv4 | 11 | 4 | 24 | 0 | 0 | 0 | 29 | 68 | 82.35% | 10.33% | 0.184 |
| zlib | 68 | 1 | 14 | 0 | 0 | 0 | 29 | 112 | 97.14% | 83.33% | 0.895 |

**Implementation details** EESI is implemented using the LLVM infrastructure [18] to analyze bitcode and our LLM error specification inference uses GPT-4 [22] as the LLM. Our experiments were run on a 2.10 GHz Xeon Silver 4216 CPU with 384 GB of RAM.

### 4.2 RQ0: how well does the static analysis of EESI perform?

For this task, we simply supply the initial domain knowledge and source code to EESI and receive its inferred error specifications. The number of inferred error specifications are represented in Table 3. From these, we can see that the most common error specification inferred across all benchmarks is $<\mathbf{0}$. Many standard library functions indicate that they return a negative error code on failure, which has been adopted by many other software programs. However, this cannot be assumed for all functions, as indicated with benchmarks such as Apache HTTPD, which often can return $<\mathbf{0}$, $>\mathbf{0}$, and $\neq\mathbf{0}$ on error. Additionally, some programs may have a considerable number of infallible ($\emptyset$) functions, e.g., MbedTLS.

EESI's precision ranges from 64.60% to 97.14% as seen in Table 3, averaging at 86.67% per benchmark. However, the recall varies even more depending on the benchmark, ranging from 10.33% to 83.33%, averaging 52.55%. The benchmark with the lowest recall, Pidgin OTRv4 (10.33%) is also notably the benchmark with the highest percentage of third-party functions at 72.2% as listed in Table 2.

### 4.3 RQ1: what is the impact of using the LLM to infer third-party error specifications?

We measure the impact of queryLLMThirdParty by running it in the first step of our interleaved error specification inference. We then run the static analysis of EESI through runAnalysis, however, we do not call queryLLMAnalysis when EESI infers $\perp$.

As we can see demonstrated in Fig. 5, we notice an average recall of 62.20% (Fig. 5c) and average increase of 29.17% (Fig. 5a) for inferred error specifications over EESI. Our precision remained similar to EESI (Fig. 5b). We notice the largest impact for the benchmark Netdata, which increased the most by 70.50%. This benchmark was impacted significantly, as it refers to many well-known libraries such as pthread. We do not see as much of an increase in Pidgin OTRv4, as many of the third-party libraries are for niche purposes, e.g., the GTK library. However, this is not the case for all library functions; for example, the error specification inference demonstrated in Fig. 4 occurs through queryLLMThirdParty.

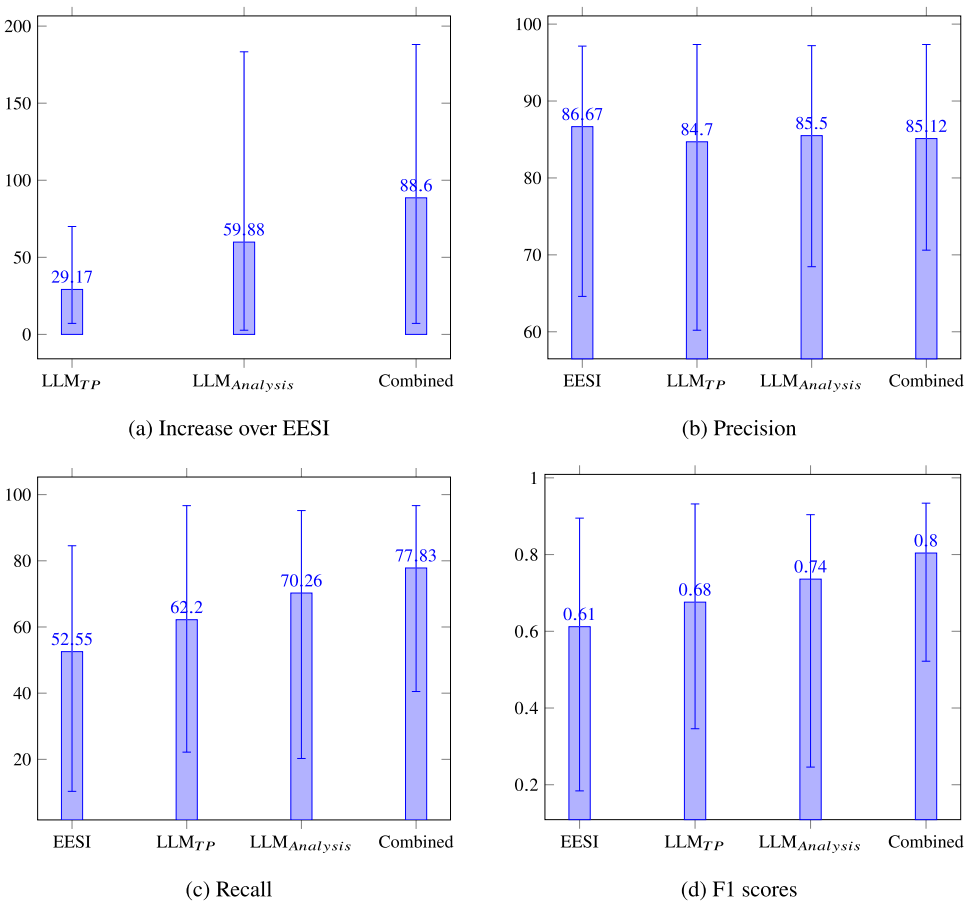
### 4.4 RQ2: what is the impact of using the direct LLM analysis?

To isolate the contributions of queryLLMAnalysis, we skip queryLLMThirdParty in the workflow. Instead, we proceed to running the static analysis of EESI, followed by querying the LLM if the result is $\perp$.

The results in Fig. 5 show an average increase of 59.88% (Fig. 5a) across all benchmarks, with an average recall of 70.26% (Fig. 5c). Our benchmark that saw the largest percentage increase was Apache HTTPD at 183.33%, which contains the second highest percentage of third-party functions (Table 2). In Fig. 2, we can see that the direct LLM analysis allows the LLM to reason about function bodies, even while the static analysis of EESI is insufficient.

### 4.5 RQ3: what is the impact of interleaving EESI and the LLM?

For our combined approach, we utilize the entire workflow, calling both queryLLMThirdParty and queryLLMAnalysis. We see in Table 4, that our combination of prompting strategies is extremely beneficial in applications such as Pidgin OTRv4, Netdata, and Apache HTTPD. Significantly improving the recall and F1 over EESI in Table 3. In fact, we see an increase over the average F1 of EESI by +0.192 (Fig. 5d). We also see the precision $\Delta$ on newly learned error specifications that were not inferred strictly via static analysis. With Netdata, we saw 144 new $<\mathbf{0}$ error specifications inferred, with our overall precision going up for the benchmark. We make note that even when we do lose some precision, as seen




(a) Increase over EESI

(b) Precision

(c) Recall

(d) F1 scores

**Fig. 5** Average increase, precision, recall, and F1-score for EESI, queryLLMThirdParty (LLM$_{TP}$), queryLLMAnalysis (LLM$_{Analysis}$), and out fully interleaved approach (Combined). The minimum and maximum benchmark results are represented as error bars for their respective metric

**Table 4** Specification counts, precision, recall, and F1-score when interleaving $EESI$ and the LLM

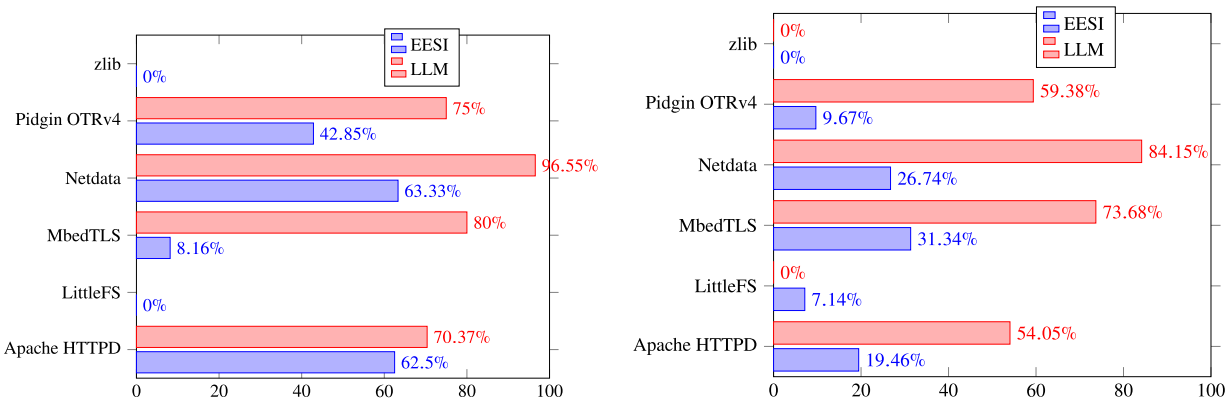
| Benchmark | $<0$ | $>0$ | 0 | $\leq 0$ | $\geq 0$ | $\neq 0$ | $\emptyset$ | Total | Increase | Precision | False positives | | Precision Δ | Recall | F1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | EESI | LLM | | | |
| Apache HTTPD | 46 | 98 | 42 | 2 | 6 | 93 | 534 | 821 | 188.07% | 85.92% | 15 | 24 | 75.20% | 66.85% | 0.752 |
| Little FS | 50 | 0 | 7 | 0 | 0 | 0 | 10 | 67 | 17.54% | 92.86% | 4 | 0 | 100.0% | 92.86% | 0.929 |
| MbedTLS | 818 | 15 | 64 | 4 | 0 | 1 | 272 | 1174 | 15.55% | 90.34% | 51 | 5 | 79.49% | 96.68% | 0.934 |
| Netdata | 161 | 72 | 222 | 2 | 4 | 1 | 234 | 696 | 150.36% | 70.59% | 57 | 28 | 75.40% | 80.63% | 0.753 |
| Pidgin OTRv4 | 16 | 4 | 95 | 0 | 4 | 0 | 53 | 172 | 152.94% | 73.68% | 16 | 19 | 70.71% | 40.50% | 0.522 |
| zlib | 76 | 1 | 14 | 0 | 0 | 0 | 29 | 120 | 7.14% | 97.35% | 3 | 0 | 100.0% | 89.43% | 0.932 |

with Apache HTTPD, we have an increase of 188.07% and still significantly improve our F1-score to 0.752.

In Fig. 5, our combination of prompting strategies to the LLM only improved upon the total number of inferred error specifications (Fig. 5a), obtaining the highest recall (Fig. 5c), and F1 (Fig. 5d), while maintaining a similar precision (Fig. 5b) to the analysis of EESI. We specifically highlight the advantages that each component has demonstrated, where queryLLMThirdParty demonstrated great success in assisting analyze benchmarks with a significant majority of third-party functions such as Pidgin OTRv4; where queryLLMAnalysis has demonstrated great success in analyzing function bodies directly, inferring error specifications in scenarios such as their called context.

### 4.6 RQ4: what are the sources of imprecision of the interleaved analysis?

As described in Sect. 3.3, we track the *background knowledge* that contributes to each function's error specification inference. The goal of this research question is to investigate various sources of imprecision from this tracked information: (1) incorrect background knowledge, (2) imprecision due to




(a) False positives (b) True positives

**Fig. 6** The percentage of true and false positives inferred by the interleaved analysis in the presence of incorrect background knowledge

the LLM, (3) incorrect inference of ∅ error specifications, and (4) nondeterminism from the LLM.

**(1) How much does incorrect background knowledge contribute to false positives?** As discussed earlier, the provided background knowledge can contribute to incorrectly inferred error specifications. To explore the relationship between false positives and the provision of incorrect background knowledge, we measure the percentage of false positives that had at least one incorrect error specification given as background knowledge at any stage during the analysis. Before calculating percentages, we first classify false positives depending on whether they come from the LLM or *EESI* and demonstrate these numbers in Table 4. Figure 6a shows the percentages for each benchmark. We observe in general a higher percentage of LLM false positives, ranging from 70.87% for Apache HTTP to 95.55% for Netdata, that involved incorrect background knowledge. In contrast, the percentage for *EESI* false positive ranges from 0% for zlib and LittleFS to 63.33% for Netdata. (Note that the absence of a percentage number in the graph means that no false positives at all were reported for that benchmark, e.g., LLM category for zlib and LittleFS). We believe that the difference observed between LLM and *EESI* false positives is potentially due to the fact that the sources of inference for the LLM includes either the entire set called function error specifications (queryLLMAnalysis), or the initial domain knowledge error specifications (queryLLMThirdParty).

As a reference, we also calculate the percentage of true positives in the presence of incorrect background knowledge, which is depicted in Fig. 6b. In general, we observe that a larger percentage of the false positives, compared to true positives, involved some incorrect background information. Nevertheless, the percentage is not consistent across benchmarks.

We investigate analysis imprecision due to the LLM based on two aspects: the percentage of incorrect background knowledge due to the LLM, and the number of calls made to the LLM during specification inference.

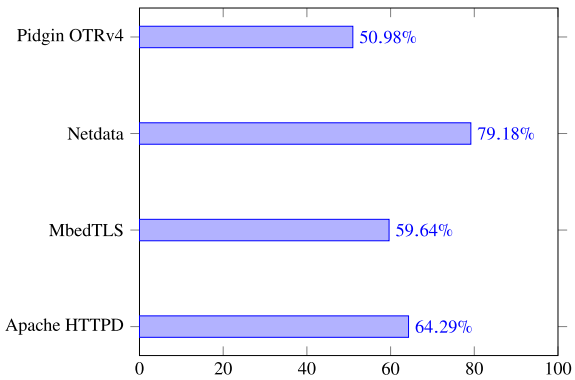



**Fig. 7** The average percentage of background knowledge that was incorrect for false positives

**(2a) What percentage of incorrect background knowledge is due to the LLM?** First, we calculate the average percentage of incorrect background knowledge for the false positives in each benchmark, as shown in Fig. 7. Second, as illustrated in Fig. 8, we calculate the percentage of this incorrect context that was inferred directly by the LLM. As a reminder, we do not infer false positives from the LLM for zlib and LittleFS. We see that for false positives, at least half of the total background knowledge used for inference is also incorrect ranging from 50.98% for Pidgin OTRv4 and 79.18% for Netdata. From both of these figures we cannot. However, the percentage of this incorrect background being from the LLM is much smaller and more varied across benchmarks. For example, MbedTLS has only 3.5% of the incorrect background knowledge coming from LLMs while Apache HTTPD has 35.57%. From these results it is hard to determine how much of the overall incorrect background knowledge coming from the LLM correlates with imprecision.

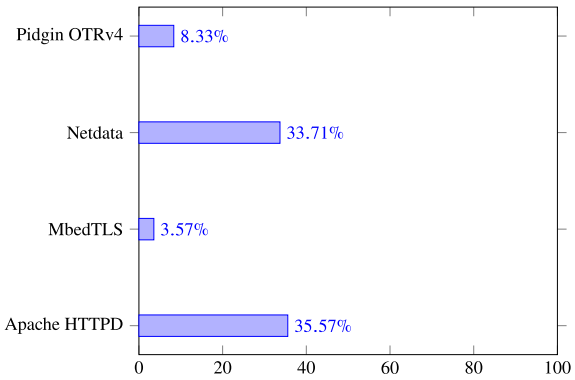



**Fig. 8** The percentage of incorrect background knowledge for false positives that was inferred by the LLM

**(2b) Is there a relationship between the number of LLM calls on an inference path, the total path length, and false positives?** So far, we have considered when the LLM is used directly to infer an specification when its background knowledge comes the LLM. Here we consider how much an error specification inferred by the LLM affects the overall inference path of other error specifications. For this, we measure the average number of LLM calls along an inference path for true positive (Fig. 9a) and false positive (Fig. 9b) error specifications. We notice that while the false positives have a higher average of LLM calls along an inference path when compared to the true positives in benchmarks such as Apache HTTPD, it also has extreme outliers. So while we can see a small trend when we try to consider specific examples of false positives, it can be difficult to find any obvious signal as to when an error specification may be incorrect.

We also show the average total path length for true positives (Fig. 10a) and false positives (Fig. 10b), where we observe a trend similar to that of the number of LLM calls. False positives also have a larger average in terms of path length, but a high number of outliers also exists. Finally, we observe similar results in Fig. 11 when comparing the average distance from a LLM call for true positives and false positives. We see that the average distance is relatively close except for in benchmarks such as zlib and MbedTLS. In general, while we notice certain benchmarks demonstrate distinct differences between false positives and true positives, these trends do not seem translate to all benchmarks.

**(3) How much of the imprecision is related to the $\emptyset$ error specification?** An $\emptyset$ (empty) error specification denotes that a function is infallible, i.e., it cannot fail and therefore does not return any error values. Because reasoning about infallible functions is especially challenging for static analysis, here we investigate how much of the imprecision is due to incorrectly inferring $\emptyset$ error specifications.

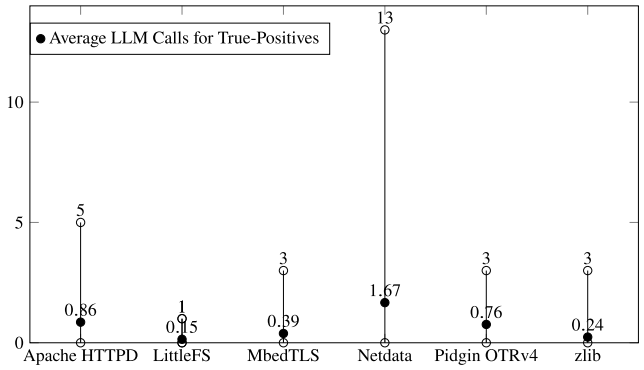
Specifically, our goal is to determine the impact that incorrect $\emptyset$ error specifications have on the precision of the analysis (see Table 4). For this, we calculate the percentage of the false positives where $\emptyset$ was incorrectly inferred (Fig. 12a), and the percentage of false positives where non-$\emptyset$ error specifications were incorrectly inferred instead of the $\emptyset$ specification (Fig. 12b). Note that LittleFS did not have any false positives related to $\emptyset$ and zlib had no false positives for error specifications inferred by the LLM. For both *EESI* and the LLM, we observe a noticeable amount of variance between how often the false positives are related to when our analysis infers $\emptyset$ in Fig. 12a. For example, in MbedTLS the $\emptyset$ contributed significantly to the overall percentage of false positives for both EESI and the LLM. Conversely, it was of little factor for Netdata. We also notice a significant amount of variance in Fig. 12b for error specifications where non-$\emptyset$ was inferred. In Netdata, this contributes to a high percentage of the false positives for EESI and low percentage for the LLM. However, for MbedTLS very few of the EESI false positives are related but a notable amount are for the LLM. By combining both of the figures in the number, we also see that the majority of the false positives between these benchmarks involves the $\emptyset$ error specification in some facet. This indicates that our interleaved analysis has the most difficulty in reasoning about the fallibility of a function when compared to other aspects of the analysis such as determining the correct specific error values.

**(4) How much does the nondeterminism of the LLM affect the results?** To measure the variance in the analysis results caused by LLM nondeterminism, we ran our interleaved analysis 5 times each on Apache HTTPD and LittleFS benchmarks using the GPT-4o-mini [23] model. We restricted the number of benchmarks and used this LLM to reduce monetary cost of conducting this experiment.

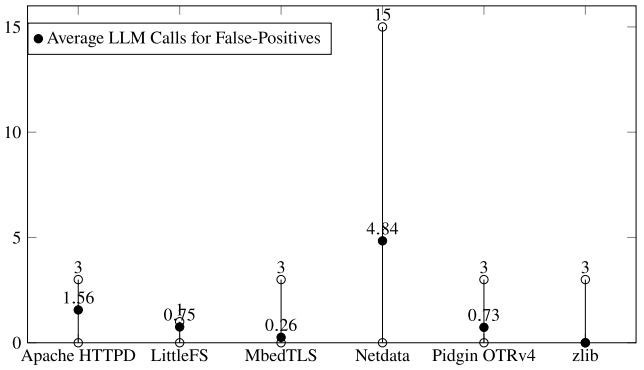
As seen in Table 5a, the interleaved analysis maintains a relatively high precision across runs for LittleFS in relation to the $\Delta$ precision, but with a reasonable degree of variance ranging from 84.62% to 100.0%. A similar trend is observed for third-party precision as indicated in the column labeled "3rd." For LittleFS, this precision does have a negative correlation with both the recall and number of inferred third-party error specifications, though the latter is a relatively small number ranging from 5 to 8.

In contrast, the variance observed in the precision and recall across the 5 analysis runs for Apache HTTPD is more significant, as seen in Table 5b. The $\Delta$ precision from the newly learned error specifications can vary from 51.19% to 70.49%. The third-party precision ranges from 14.48% to 72.31%. The overall recall of the interleaved analysis ranges from 64.74% to 81.23%, and the newly inferred third-party function error specifications range from 7 to 65.

The difference in the variance of the interleaved analysis could be due to the nature of the (ground-truth) error specifications for the functions in these two benchmarks. Unlike

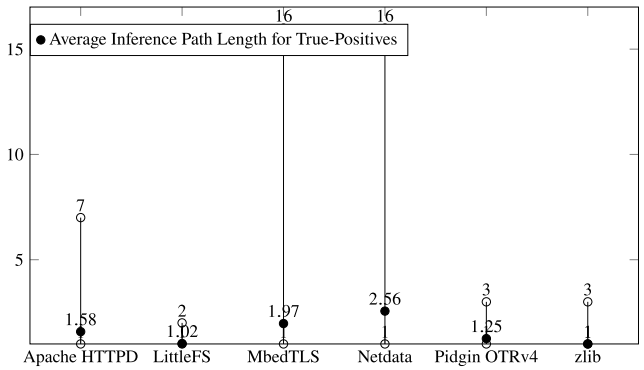



(a) The average number of calls for true positives.

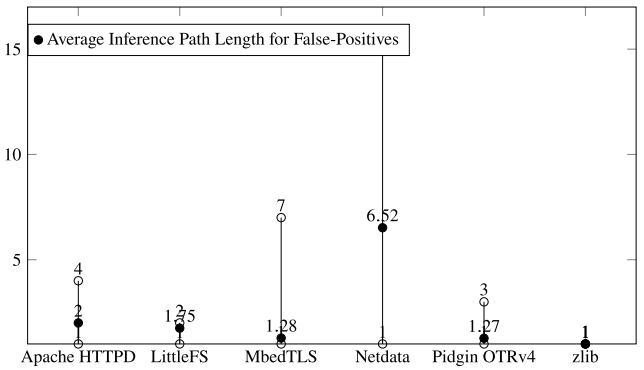



(b) The average number of calls for false positives.

**Fig. 9** The average number of calls to the LLM along an inference path for error specifications that our analysis inferred correctly and incorrectly. The minimum and maximum values are also displayed for each benchmark




(a) The average inference path length for true positives.




(b) The average inference path length for false positives.

**Fig. 10** The average path length for true and false positives error specifications inferred by our interleaved analysis

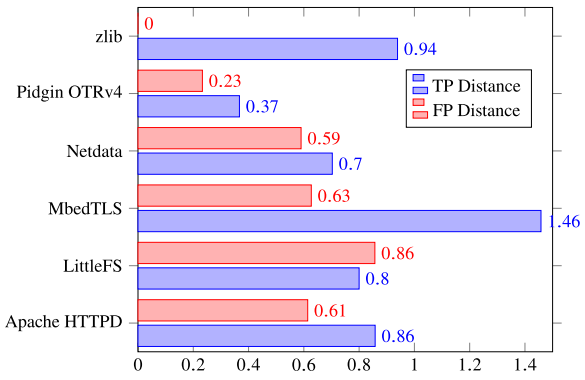


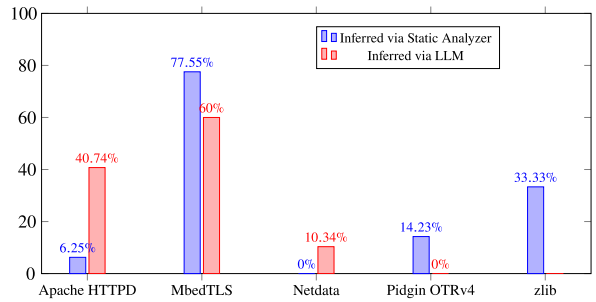

**Fig. 11** The average distance from a call to the LLM in the inference chain for true positives (TP) versus false positives (FP)

Apache HTTPD, most of the functions in LittleFS have the same error specification (<**0**) and it calls relatively few third-party functions. Further understanding and mitigating such variance is an interesting future research direction, and will be critical for deployment of any analysis that incorporates LLMs.

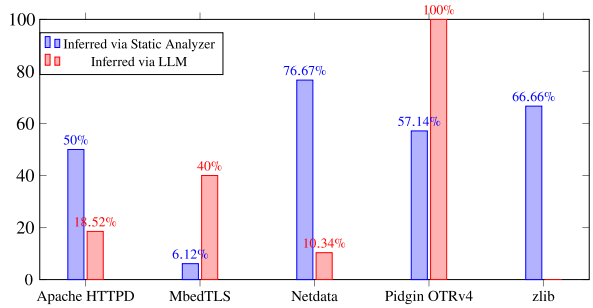
### 4.7 RQ5: what is the performance cost in using the LLM in the interleaved analysis?

To evaluate the performance cost of introducing LLMs into the interleaved analysis, we measure: (1) execution time of EESI versus the interleaved analysis, (2) number of requests to the LLM, (3) number of tokens given as input to the LLM, and (4) number of tokens given as output from the LLM. For this evaluation, we use GPT-4o-mini for monetary purposes. It should be taken into consideration that models' response time can vary significantly. The metrics for requests and tokens are gathered from usage reports from OpenAI [24].

The total number of requests to run the interleaved analysis on all benchmarks is 973. The total number of input and output tokens is 1,660,285 and 66,533, respectively. The cost using the 4o mini model is just 29 cents. As seen in Table 6, the interleaved analysis is much slower compared to EESI, but it still finishes in under 12 minutes for all benchmarks. Benchmarks for which the interleaved analysis inferred more error specifications compared to EESI observed a larger in-




(a) The percentage of false positives where $\emptyset$ was inferred via our interleaved analysis.




(b) The percentage of false positives where error values were inferred via our interleaved analysis, but the correct error specification is $\emptyset$.

**Fig. 12** Percentage of false positives that involve the $\emptyset$ error specification. Note that LittleFS has no false positives involving the $\emptyset$ error specification and the LLM did not infer any false positives for zlib

**Table 5** Repeated interleaved analysis results on Apache HTTPD and LittleFS. Measured results include total precision percentage, Δ precision percentage from the baseline static analysis, third-party (3rd) precision percentage, overall recall percentage, and newly inferred third-party error specifications

| Run | Precision | | | Recall | Inferred 3rd |
|---|---|---|---|---|---|
| | Total | Δ | 3rd | | |
| (a) LittleFS | | | | | |
| 1 | 89.83% | 84.62% | 75.00% | 98.15% | 8 |
| 2 | 91.23% | 90.91% | 83.33% | 94.55% | 6 |
| 3 | 89.83% | 84.62% | 75.00% | 98.15% | 8 |
| 4 | 91.38% | 91.67% | 85.71% | 96.67% | 7 |
| 5 | 92.86% | 100.00% | 100.00% | 92.86% | 5 |
| (b) Apache HTTPD | | | | | |
| 1 | 71.43% | 51.19% | 14.58% | 73.77% | 48 |
| 2 | 78.21% | 58.59% | 25.00% | 65.47% | 12 |
| 3 | 79.04% | 66.49% | 72.31% | 81.23% | 65 |
| 4 | 83.94% | 70.49% | 50.00% | 65.53% | 8 |
| 5 | 82.05% | 66.39% | 57.14% | 64.74% | 7 |

**Table 6** Total time to run interleaved analysis versus EESI

| Benchmark | Time (minutes:seconds) | |
|---|---|---|
| | EESI | Interleaved analysis |
| Apache HTTPD | 00:33 | 11:08 |
| Little FS | 00:25 | 00:25 |
| MbedTLS | 00:51 | 05:25 |
| Netdata | 01:09 | 11:30 |
| Pidgin OTRv4 | 00:25 | 02:00 |
| zlib | 00:40 | 01:00 |

crease in execution time. Using a local models without rate limits may help mitigate this performance cost.

## 5 Related work

**Error specification inference** Acharya and Xie [1] introduce techniques for mining error specifications for APIs using static traces. APEx [17] uses path-sensitive symbolic execution to find error-paths on the assumption that error paths are shorter than normal paths. Several other works [13, 27, 28, 34] find function error specifications via fault injection. MLPEx [40] is a machine-learning based approach that uses path-features to learn whether or not a program path is an error path. EESI [9] is a static analysis of C programs for error specification inference that allows the use of domain knowledge to bootstrap the analysis. Our task improves EESI by interleaving it with LLM prompting.

**Program analysis and LLMs** Ahmed and Devanbu [2] demonstrate that when an LLM is provided semantic information produced by static analysis, then tasks such as code summarization can be significantly improved. Li et al. [20] demonstrate that by carefully crafting questions using function-level behavior and summaries, LLMs can assist in removing false positives from a bug finding tool. Li et al. [21] also introduce a technique for combining static analysis using symbolic execution with LLMs to find *Use Before Initialization* (UBI) bugs, demonstrating that the LLM can be used to extract some program semantics and filter out false positives caused by the imprecision of the static analysis. Wen et al. [39] also report success in removing false positive warnings by using customized questions with domain knowledge from the Juliet [15] benchmark. LLMs have also been recently used to generate program invariants [26], including generating loop invariants [16] and subsequently ranking them using zero-shot prompting [4]. In contrast to all of the above, our work *interleaves* facts provided by both a static analysis and an LLM to improve the precision of an existing static analysis for error specification inference.

**Program analysis and machine learning** Seldon [8] is a tool using semisupervised learning through building and solving a constraint system from information flow constraints for taint specification inference. InspectJS [12] is an approach for taint specification inference that uses manual modeling from CodeQL [14], inferred specifications using an adaptation of Seldon, a ranking strategy using embeddings, and manual user feedback. As discussed previously in relation to error specification inference, MLPEx [40] uses machine learning for error specification inference. While these approaches combine machine learning and traditional program analysis techniques to improve analysis results, our technique differs in that we use LLMs, and in that both the static analysis and LLM-based inference results are interleaved throughout the entire analysis.

**Code LLMs** There are many LLMs that are dedicated to generating and reasoning about programming languages. Some of these models are general working for both natural language and programming languages [22, 29, 36]. Whereas some other models target programming languages specifically [32]. There has also been considerable work done towards how to effectively guide the model to generate useful output through prompting. One technique called *chain-of-thought* [38] provides examples with chain-of-thought reasoning to the models in an attempt to guide the model to utilize the same reasoning when answering future queries. There is also the notion of *self-consistency* [37], which is the process of querying the model multiple times and taking the most consistently given answer. While there has been significant work in trying to improve model performance, a separate challenge is how to effectively evaluate code LLMs generated code. A technique CodeBleu [30], builds on previous work for natural language evaluation BLEU [25], by introducing code syntax via AST and code semantics via data-flow analysis with the previous n-gram matching based approach in BLEU. While these approaches have demonstrated some success, one common criticism is that they do not evaluate the semantics of the generated program at a deeper level. One such approach to address this is by leveraging a *pass@k* rate [6], which measures the likelihood that code generated from a model will pass a set of tests.

**Retrieval and LLMs** The process of retrieving relevant context to provide to LLMs for generation tasks is called Retrieval-Augmented Generation (RAG) [19]. General retrieval strategies often incorporate retrieval strategies such as TF-IDF [33] or BM-25 [31], which can be extended to capturing source code in programming languages. In the context of code completion, retrieval is often leveraged to provide context about code entities that an LLM is unaware of from training, such as private code repositories. In order to bring the most relevant context as part of the retrieval, there have also been several works that introduce elements of static analysis to guide the retrieval. Cheng et al. [7] incorporate a dataflow analysis to track relationships between code entities in order to retrieve the most relevant code entity signatures as context, e.g., frequently used APIs. Our approach does *retrieve* already learned error specifications as context to the LLM during our interleaved analysis, but our approach is concerned with both the LLM and static analyzer providing learned error specifications to each other in order to increase the overall learned program knowledge.

## 6 Conclusion

We have presented an approach for interleaving static program analysis and LLMs for the task of error specification inference. We have demonstrated that providing program facts inferred by the EESI static analysis to the LLM helps the LLM to infer correct error specifications, and in-turn the results from the LLM can assist EESI to learn new error specifications. Evaluating our approach on real-world benchmarks shows that interleaving LLMs into static analysis using our approach improves average recall from 52.55% to 77.83% and improves F1-score from 0.612 to 0.804 without significantly reducing precision.

**Funding** This work was supported by the National Science Foundation under awards CCF-1750983, CCF-2119348, and CCF-2107592.



## References

1. Acharya, M., Xie, T.: Mining API error-handling specifications from source code. In: Chechik, M., Wirsing, M. (eds.) Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009. York, UK, March 22–29, 2009. Proceedings, Lecture Notes in Computer Science, vol. 5503, pp. 370–384. Springer, York (2009). https://doi.org/10.1007/978-3-642-00593-0_25
2. Ahmed, T., Devanbu, P.T.: Few-shot training llms for project-specific code-summarization. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10–14, 2022, pp. 177:1–177:5. ACM (2022). https://doi.org/10.1145/3551349.3559555

3. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6–12, 2020, virtual (2020). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html
4. Chakraborty, S., Lahiri, S.K., Fakhoury, S., Lal, A., Musuvathi, M., Rastogi, A., Senthilnathan, A., Sharma, R., Swamy, N.: Ranking LLM-generated loop invariants for program verification. In: Bouamor, H., Pino, J., Bali, K. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore December 6–10, 2023, pp. 9164–9175. Association for Computational Linguistics (2023). https://doi.org/10.18653/v1/2023.findings-emnlp.614
5. Chapman, P.J., Rubio-González, C., Thakur, A.V.: Interleaving static analysis and LLM prompting. In: SOAP@PLDI, pp. 9–17. ACM, New York (2024)
6. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating large language models trained on code. CoRR (2021). arXiv:2107.03374
7. Cheng, W., Wu, Y., Hu, W.: Dataflow-guided retrieval augmentation for repository-level code completion. In: ACL (2024)
8. Chibotaru, V., Bichsel, B., Raychev, V., Vechev, M.T.: Scalable taint specification inference with big code. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA June 22–26, 2019, pp. 760–774. ACM (2019). https://doi.org/10.1145/3314221.3314648
9. DeFreez, D., Baldwin, H.M., Rubio-González, C., Thakur, A.V.: Effective error-specification inference via domain-knowledge expansion. In: ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26–30, 2019, pp. 466–476. ACM, New York (2019). https://doi.org/10.1145/3338906.3338960
10. DeFreez, D., Thakur, A.V., Rubio-González, C.: Path-based function embedding and its application to error-handling specification mining. In: ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA November 4–9, 2018, pp. 423–433. ACM (2018). https://doi.org/10.1145/3236024.3236059
11. DeFreez, D., Thakur, A.V., Rubio-González, C.: Path-based function embeddings. In: Chaudron, M., Crnkovic, I., Chechik, M., Harman, M. (eds.) Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 – June 03, 2018, pp. 430–431. ACM (2018). https://doi.org/10.1145/3183440.3195042
12. Dutta, S., Garbervetsky, D., Lahiri, S.K., Schäfer, M.: Inspectjs: leveraging code similarity and user-feedback for effective taint specification inference for JavaScript. In: 44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22–24, 2022, pp. 165–174. IEEE (2022). https://doi.org/10.1109/ICSE-SEIP55303.2022.9794015
13. Fetzer, C., Högstedt, K., Felber, P.: Automatic detection and masking of non-atomic exception handling. In: 2003 International Conference on Dependable Systems and Networks (DSN 2003), June 22–25, 2003, San Francisco, CA, USA, Proceedings, pp. 445–454. IEEE Computer Society (2003). https://doi.org/10.1109/DSN.2003.1209955
14. GitHub: Codeql (2021). https://codeql.github.com
15. Jr, F.B., Black, P.: The Juliet 1.1 C/C++ and Java test suite **45**(10) (2012). https://doi.org/10.1109/MC.2012.345
16. Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S.K., Lal, A., Rastogi, A., Roy, S., Sharma, R.: Finding inductive loop invariants using large language models. CoRR (2023). https://doi.org/10.48550/arXiv.2311.07948. arXiv:2311.07948
17. Kang, Y.J., Ray, B., Jana, S.: Apex: automated inference of error specifications for C apis. In: Lo, D., Apel, S., Khurshid, S. (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3–7, 2016, pp. 472–482. ACM (2016). https://doi.org/10.1145/2970276.2970354
18. Lattner, C., Adve, V.S.: LLVM: a compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), San Jose, CA, USA, March 20–24, 2004, pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665
19. Lewis, P.S.H., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-augmented generation for knowledge-intensive NLP tasks. In: NeurIPS (2020)
20. Li, H., Hao, Y., Zhai, Y., Qian, Z.: Assisting static analysis with large language models: a ChatGPT experiment. In: Chandra, S., Blincoe, K., Tonella, P. (eds.) Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3–9, 2023, pp. 2107–2111. ACM (2023). https://doi.org/10.1145/3611643.3613078
21. Li, H., Hao, Y., Zhai, Y., Qian, Z.: Enhancing static analysis for practical bug detection: an LLM-integrated approach (2024)
22. OpenAI: GPT-4 technical report (2023). https://doi.org/10.48550/arXiv.2303.08774
23. OpenAI: GPT-4o (2024a). https://openai.com/index/hello-gpt-4o/
24. OpenAI: OpenAI developer platform (2024b). https://platform.openai.com/docs/overview
25. Papineni, K., Roukos, S., Ward, T., Zhu, W.: Bleu: a method for automatic evaluation of machine translation. In: ACL, pp. 311–318. ACL (2002)
26. Pei, K., Bieber, D., Shi, K., Sutton, C., Yin, P.: Can large language models reason about program invariants? In: Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J. (eds.) International Conference on Machine Learning, ICML 2023, Honolulu, Hawaii, USA, July 23–29, 2023. Proceedings of Machine Learning Research, vol. 202, pp. 27496–27520. PMLR (2023). https://proceedings.mlr.press/v202/pei23a.html
27. Prabhakaran, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Model-based failure analysis of journaling file systems. In: 2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June – 1 July 2005, Yokohama, Japan, Proceedings, pp. 802–811. IEEE Computer Society (2005). https://doi.org/10.1109/DSN.2005.65
28. Prabhakaran, V., Bairavasundaram, L.N., Agrawal, N., Gunawi, H.S., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: IRON file systems. In: Herbert, A., Birman, K.P. (eds.) Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23–26, 2005, pp. 206–220. ACM (2005). https://doi.org/10.1145/1095810.1095830

29. Radford, A., Narasimhan, K.: Improving language understanding by generative pre-training (2018). https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf
30. Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., Ma, S.: CodeBLEU: a method for automatic evaluation of code synthesis. CoRR (2020). arXiv:2009.10297
31. Robertson, S., Zaragoza, H.: The probabilistic relevance framework: BM25 and beyond. **3**(4), 333–389 (2009). ISSN 1554-0669. https://doi.org/10.1561/1500000019
32. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Canton-Ferrer, C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., Synnaeve, G.: Code Llama: Open foundation models for code. CoRR (2023). https://doi.org/10.48550/arXiv.2308.12950. arXiv:2308.12950
33. Sparck Jones, K.: A statistical interpretation of term specificity and its application in retrieval, pp. 132–142. Taylor Graham Publishing, GBR (1988). ISBN 0947568212
34. Süßkraut, M., Fetzer, C.: Automatically finding and patching bad error handling. In: Sixth European Dependable Computing Conference, EDCC 2006, Coimbra, Portugal, October 18–20, 2006, pp. 13–22. IEEE Computer Society (2006). https://doi.org/10.1109/EDCC.2006.3
35. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton-Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P.S., Lachaux, M., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E.M., Subramanian, R., Tan, X.E., Tang, B., Taylor, R., Williams, A., Kuan, J.X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., Scialom, T. (eds.): Llama 2: Open foundation and fine-tuned chat models. CoRR (2023). arXiv:2307.09288. https://doi.org/10.48550/arXiv.2307.09288
36. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, Long Beach, CA, USA, December 4–9, 2017, pp. 5998–6008 (2017). https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html
37. Wang, X., Wei, J., Schuurmans, D., Le, Q.V., Chi, E.H., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. In: The Eleventh International Conference on Learning Representations (2023). https://openreview.net/forum?id=1PL1NIMMrw
38. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models (2023). https://doi.org/10.48550/arXiv.2201.11903
39. Wen, C., Cai, Y., Zhang, B., Su, J., Xu, Z., Liu, D., Qin, S., Ming, Z., Tian, C.: Automatically inspecting thousands of static bug warnings with large language model: How far are we? ACM Trans. Knowl. Discov. Data (2024). https://doi.org/10.1145/3653718. ISSN 1556-4681
40. Wu, B., C, J.P. III, He, Y., Schlecht, A., Chen, S.: Generating precise error specifications for C: a zero shot learning approach. Proc. ACM Program. Lang. **3**(OOPSLA), 160:1–160:30 (2019). https://doi.org/10.1145/3360586