Interleaving Static Analysis and LLM Prompting

Patrick J. Chapman
University of California, Davis
Davis, USA
pchapman@ucdavis.edu

Cindy Rubio-González
University of California, Davis
Davis, USA
crubio@ucdavis.edu

Aditya V. Thakur
University of California, Davis
Davis, USA
avthakur@ucdavis.edu

Abstract

This paper presents a new approach for using Large Language Models (LLMs) to improve static program analysis. Specifically, during program analysis, we *interleave* calls to the static analyzer and queries to the LLM: the prompt used to query the LLM is constructed using intermediate results from the static analysis, and the result from the LLM query is used for subsequent analysis of the program. 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, which can aid in program understanding as well as in finding error-handling bugs. We evaluate our approach on real-world C programs, such as MbedTLS and zlib, by incorporating LLMs into EESI, a state-of-the-art static analysis for error-specification inference. Compared to EESI, our approach achieves higher recall across all benchmarks (from average of 52.55% to 77.83%) and higher F1-score (from average of 0.612 to 0.804) while maintaining precision (from average of 86.67% to 85.12%).

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; *Error handling and recovery*; • **Computing methodologies** → **Natural language processing**.

Keywords: static analysis, large language model, error handling, error specifications

ACM Reference Format:
Patrick J. Chapman, Cindy Rubio-González, and Aditya V. Thakur. 2024. Interleaving Static Analysis and LLM Prompting. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3652588.3663317

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SOAP '24, June 25, 2024, Copenhagen, Denmark
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0621-9/24/06
https://doi.org/10.1145/3652588.3663317

1 Introduction

This paper presents a new approach for using *Large Language Models (LLMs)* to improve static program analysis. LLMs [18, 25] have been shown to demonstrate impressive reasoning abilities in natural and programming languages tasks via few-shot [3] and chain-of-thought [28] 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.

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 (Section 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 [6] has shown higher effectiveness and performance at inferring error specifications compared to prior approaches [1, 7, 14]. Our approach interleaves calls to the EESI static analyzer and the LLM (Figure 1).

We evaluated our approach on six real-world C programs, such as MbedTLS and zlib (Section 5). 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.
- We designed a tool for error specification inference of C programs using our approach of combining 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.

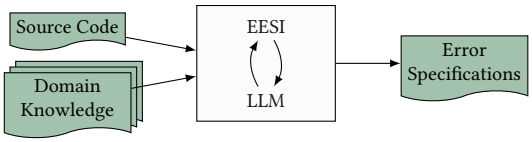


Figure 1. Our approach infers error specifications by interleaving calls to the EESI static analyzer and the LLM.

2 Background

Error Specification Inference. 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 bugs, e.g., developers may miss or incorrectly check the error return values of functions.

A few approaches have presented techniques for inferring error specifications [1, 6–8, 14, 30]. In this paper, we consider a state-of-the-art static program analysis using abstract interpretation for error specification inference named EESI [6]. As input, EESI takes in multiple forms of optional user-supplied *initial domain knowledge*: (1) initial error specifications, (2) error codes, (3) success codes, and (4) error-only functions (only called along error paths). With this initial domain knowledge, EESI uses static analysis to infer new error specifications.

While EESI has demonstrated success in error specification inference, it has two inherent limitations that affect its recall and precision: (1) *incomplete program facts*, and (2) *third–party* functions. As EESI is a static program analyzer using abstract interpretation to infer program semantics related to idiomatic practices, it provides approximations that may be insufficient in learning enough program facts for error specification inference. One important source of incomplete knowledge is *third-party functions*. Third-party functions are called within a program, but defined elsewhere. Because the analyzer does not have access to the source code, it cannot reason about their error specifications.

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 [25] and programming languages [23], while some models are trained for both [18, 22, 26]. 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, 22, 27, 28]. These approaches include concepts such as *chain-of-thought* [28], where LLMs are given question and answer as examples with the associated chain-of-though reasoning, and *self-consistency* [27] prompting, where LLMs are prompted with the same question multiple times, using the most consistent answer given.

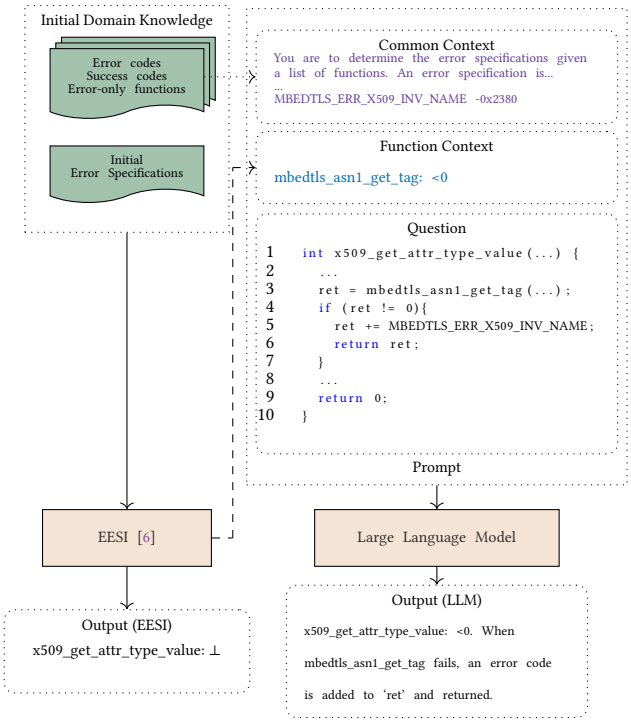


Figure 2. Using EESI and the LLM to infer error specifications

3 Overview Example

This section illustrates how our approach of interleaving calls to the EESI static analyzer and the LLMs to infer error specifications. Consider the function `x509_get_attr_type_value` in MbedTLS. EESI alone is unable to infer its error specification; EESI infers $\perp$ as the function error-specification as shown in Figure 2.

The LLM alone is also unable to infer its error specification. We can construct a prompt to the LLM that includes the general description of the error specification inference problem (Common Context in Figure 2) as well as the source code of the function (Question in Figure 2). However, querying the LLM with just this information is not sufficient to give us the correct error specification. 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 non-zero value; that is, the LLM infers that the error value could be anything, and the error specification is $\top$, instead of $< \mathbf{0}$.

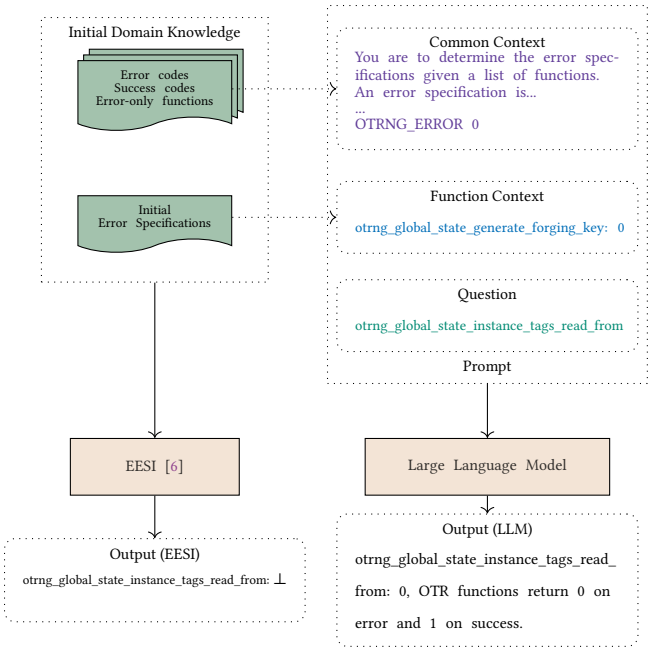


Figure 3. Using the LLM to infer error spefication of a third-party function

However, if we also include intermediate results from the EESI static analyzer in the LLM prompt, then the LLM is able to return the fact that `x509_get_attr_type_value` returns a value $<\mathbf{0}$ on error. In particular, the LLM prompt includes the error specification of the function `mbedtls_asn1_tag` that is called from `x509_get_attr_type_value` (Function Context in Figure 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 ($<\mathbf{0}$) for `mbedtls_x509_get_name` from analyzing its implementation:

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

The specifics about the LLM prompt construction; viz, Common Context, Function Context, and Question, are deferred to Section 4.1.

Figure 3 illustrates another scenario illustrating the benefits of incorporating calls to an LLM in the static analyzer. 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`.

4 Approach

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

4.1 Building Prompts

When interacting with the LLM, we construct a prompt that consists of the *Common Context*, *Function Context*, and *Question*, as mentioned in Section 3.

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 multiple, 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. The *Function Context* of the 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 providing 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 Figure 2.

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.

4.2 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 in 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 that called functions are inferred before their caller, because called functions provide additional context for error specification inference. Note, 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 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).

Algorithm 1: InferErrorSpecification(P, F)

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

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

4.2.1 Third-Party Function Error Specifications.

For each function, we first check if it is a third-party function (Line 3), and if it is, we perform queryLLMThirdParty as demonstrated on Line 4. 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 22. The *Question* in this case just simply lists the name of the function-of-interest (Line 21). The LLM is then queried, where the output is then parsed (Line 24) and if any error specification is learned, the program facts are updated (Line 10).

4.2.2 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.

Unlike the *Function Context* provided in queryLLMThirdParty, we only provide the known error specifications of called functions contained in the function definition (Line 15). We demonstrate an example of this in Figure 2, where error specification `mbedtls_asn1_get_tag` is learned from EESI and provided as *Function Context* to the LLM, correctly inferring `x509_get_attr_type_value`.

The constructed *Question* as part of the prompt consists of the source code of the function being analyzed (Line 14).

The resulting output from the LLM is then parsed (Line 17) and any newly learned error specification is updated in the program facts on Line 10.

4.2.3 Validating the LLM Response.

We re-query 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. As both of these program semantics are obtained via an approximation during the analysis of EESI, we cannot

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

			Domain Knowledge			
					Codes	
Benchmark	KLOC	Ver.	Init. Specs	EO	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

guarantee that these inconsistencies are removed entirely, but we can utilize these rules to limit low-hanging fruit.

5 Experimental Evaluation

For our experimental evaluation, we perform an ablation study. We propose three 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 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?

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

5.1 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.

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.

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%)

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

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

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

Implementation Details.

EESI is implemented using the LLVM infrastructure [15] to analyze bitcode and our LLM error specification inference uses GPT-4 [18] as the LLM. Our experiments were run on a 2.10 GHz Xeon Silver 4216 CPU with 384 GB of RAM.

5.2 Experimental Results

RQ0: How well does EESI perform in error specification inference?

For this task, we simply supply the initial domain knowledge and source code to the static analyzer of 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

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

Benchmark	$<\mathbf{0}$	$>\mathbf{0}$	$\mathbf{0}$	$\leq\mathbf{0}$	$\geq\mathbf{0}$	$\neq\mathbf{0}$	$\emptyset$	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

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 achieves a precision ranging 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.

RQ1: What is the impact of queryLLMThirdParty?

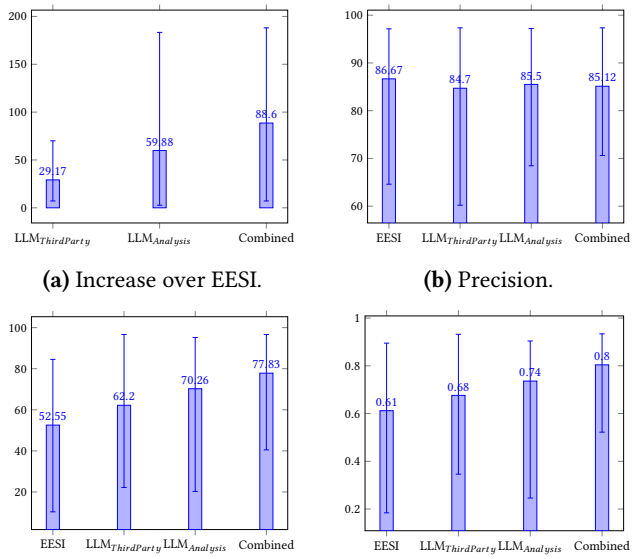
We measure the impact of queryLLMThirdParty by running it in the first step of our interleaved error specification inference. We then run the static analysis of EESI through runAnalysis, however, we do not call queryLLMAnalysis when EESI infers $\perp$.

As we can see demonstrated in Figure 4, we notice an average recall of 62.20% (Figure 4c) and average increase of 29.17% (Figure 4a) for inferred error specifications over EESI. Our precision remained similar to EESI (Figure 4b). 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 the 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 Figure 3 occurs through queryLLMThirdParty.

RQ2: What is the impact of using queryLLMAnalysis?

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 depicted in Figure 4 demonstrate an average increase of 59.88% (Figure 4a) across all benchmarks, with an average recall of 70.26% (Figure 4c). 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 Figure 2, we can see that the direct LLM analysis allows the LLM to reason about



(a) Increase over EESI. **(b)** Precision.

(c) Recall. **(d)** F1 scores.

Figure 4. Average increase, precision, recall, and F1-score for approaches. The minimum and maximum benchmark results are represented as error bars for their respective metric.

function bodies, even while the static analysis of EESI is insufficient.

RQ3: What is the impact of interleaving EESI and 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 (Figure 4d). We also see the precision Δ 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 with Apache HTTPD, we have an increase of 188.07% and still significantly improve our F1-score to 0.752.

Table 4. Specification counts, precision, recall, and F1-score for our framework interleaving static analysis and LLMs.

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

In Figure 4, our combination of prompting strategies to the LLM only improved upon the total number of inferred error specifications (Figure 4a), obtaining the highest recall (Figure 4c), and F1 (Figure 4d), while maintaining a similar precision (Figure 4b) 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.

6 Related Work

Error Specification Inference. Acharya and Xie [1] introduce techniques for mining error specifications for APIs using static traces. APEx [14] uses path-sensitive symbolic execution to find error-paths on the assumption that error paths are shorter than normal paths. Several other works [10, 20, 21, 24] find function error specifications via fault injection. MLPEx [30] is a machine-learning based approach that uses path-features to learn whether or not a program path is an error path. EESI [6] 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 a LLM is provided semantic information produced by static analysis, then tasks such as code summarization can be significantly improved. Li et al. [16] 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. [17] 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. [29] also demonstrate success in removing false positive warnings by using customized questions with domain knowledge from the Juliet [12] benchmark. LLMs have also been recently used to generate program invariants [19], including generating loop invariants [13] 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 a LLM to improve the precision of an existing static analysis for error specification inference.

Program Analysis and Machine Learning. Seldon [5] is a tool using semi-supervised learning through building and solving a constraint system from information flow constraints for taint specification inference. InspectJS [9] is an approach for taint specification inference that uses manual modeling from CodeQL [11], 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 [30] uses machine learning for error specification inference. While these approaches combined machine learning and traditional program analysis techniques to improve analysis results, our technique differs in that we are using LLMs and that both the static analysis and LLM-based inference results are interleaved throughout the entire analysis.

7 Conclusion

We have presented an approach for interleaving static program analysis and LLMs for the task of error specification inference. We have demonstrated that by providing program facts from the analysis of EESI to the LLM that it can infer error specifications correctly and in-turn can assist EESI to further learn new error specifications. We show this in our evaluation (Section 5), where our average recall grows from 52.55% to 77.83% and our F1-score improves from 0.612 to 0.804. Our evaluation also demonstrates a similar precision to the original static analysis, where the average only decreases from 86.67% to 85.12%.

Acknowledgments

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

References

[1] Mithun Acharya and Tao Xie. 2009. Mining API Error-Handling Specifications from Source Code. In *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), Marsha Chechik and Martin Wirsing (Eds.). Springer, 370–384. https://doi.org/10.1007/978-3-642-00593-0_25

[2] Toufique Ahmed and Premkumar T. Devanbu. 2022. 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*. ACM, 177:1–177:5. https://doi.org/10.1145/3551349.3559555

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, and Amanda Askell et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[4] Saikat Chakraborty, Shuvendu K. Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, 9164–9175. https://doi.org/10.18653/V1/2023.FINDINGS-EMNLP.614

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

[6] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V. Thakur. 2019. Effective error-specification inference via domain-knowledge expansion. In *ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 466–476. https://doi.org/10.1145/3338906.3338960

[7] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 423–433. https://doi.org/10.1145/3236024.3236059

[8] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based function embeddings. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 430–431. https://doi.org/10.1145/3183440.3195042

[9] Saikat Dutta, Diego Garbervetsky, Shuvendu K. Lahiri, and Max Schäfer. 2022. 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*. IEEE, 165–174. https://doi.org/10.1109/ICSE-SEIP55303.2022.9794015

[10] Christof Fetzer, Karin Högstedt, and Pascal Felber. 2003. Automatic Detection and Masking of Non-Atomic Exception Handling. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*. IEEE Computer Society, 445–454. https://doi.org/10.1109/DSN.2003.1209955

[11] GitHub. 2021. *CodeQL*. https://codeql.github.com

[12] Frederick Boland Jr. and Paul Black. 2012. The Juliet 1.1 C/C++ and Java Test Suite. 45 (October 2012). https://doi.org/10.1109/MC.2012.345

[13] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2023. Finding Inductive Loop Invariants using Large Language Models. *CoRR* abs/2311.07948 (2023). https://doi.org/10.48550/ARXIV.2311.07948 arXiv:2311.07948

[14] Yuan Jochen Kang, Baishakhi Ray, and Suman Jana. 2016. APEx: automated inference of error specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 472–482. https://doi.org/10.1145/2970276.2970354

[15] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[16] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *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*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 2107–2111. https://doi.org/10.1145/3611643.3613078

[17] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. (2024).

[18] OpenAI. 2023. GPT-4 Technical Report. https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[19] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 27496–27520. https://proceedings.mlr.press/v202/pei23a.html

[20] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. 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*. IEEE Computer Society, 802–811. https://doi.org/10.1109/DSN.2005.65

[21] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, Andrew Herbert and Kenneth P. Birman (Eds.). ACM, 206–220. https://doi.org/10.1145/1095810.1095830

[22] Alec Radford and Karthik Narasimhan. 2018. 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

[23] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950

[24] Martin Süßkraut and Christof Fetzer. 2006. Automatically Finding and Patching Bad Error Handling. In *Sixth European Dependable Computing Conference, EDCC 2006, Coimbra, Portugal, 18-20 October 2006*. IEEE Computer Society, 13–22. https://doi.org/10.1109/EDCC.2006.3

[25] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal

Bhargava, and Shruti Bhosale et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

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

[27] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=1PL1NIMMrw

[28] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. https://doi.org/10.48550/arXiv.2201.11903 arXiv:2201.11903 [cs.CL]

[29] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Cong Tian. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Trans. Knowl. Discov. Data* (mar 2024). https://doi.org/10.1145/3653718

[30] Baijun Wu, John Peter Campora III, Yi He, Alexander Schlecht, and Sheng Chen. 2019. Generating precise error specifications for C: a zero shot learning approach. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 160:1–160:30. https://doi.org/10.1145/3360586

Received 2024-03-07; accepted 2024-04-19