An Empirical Study of LLM-Generated Specifications for VeriFast

Wen Fan
Purdue University
West Lafayette, Indiana, USA
fan372@purdue.edu

Minh Tran
Purdue University
West Lafayette, Indiana, USA
tran299@purdue.edu

Sanya Dod
Purdue University
West Lafayette, Indiana, USA
sdod@purdue.edu

Xin Hu
University of Michigan
Ann Arbor, Michigan, USA
hsinhu@umich.edu

Marilyn Rego
University of Michigan
Ann Arbor, Michigan, USA
mrego@umich.edu

Danning Xie
Meta
Menlo Park, California, USA
xiedn1105@gmail.com

Jenna DiVincenzo
Purdue University
West Lafayette, Indiana, USA
jennad@purdue.edu

Lin Tan
Purdue University
West Lafayette, Indiana, USA
lintan@purdue.edu

arXiv:2606.26490v1 [cs.SE] 25 Jun 2026

***Abstract*—Static verification tools can assure industrial scale software, but require significant human labor to write specifications. This is particularly true of static verifiers based on separation logic (SL verifiers), which excel at verifying heap-manipulating programs, but require many complex auxiliary specifications to reason about heap structure. Recent work applies large language models (LLMs) to generate code, tests, and proofs, including specifications for verifiers, but mostly targeting non-SL verifiers. To address this gap, this paper thoroughly evaluates how well LLMs perform when prompted to generate specifications for verifying 303 C functions with the SL verifier VeriFast. We explored eight prompting approaches, ten LLMs, and three input types in two stages. Quantitative and qualitative analyses are used to assess the LLM-generated code and specifications for functional behavior, verifiability and errors. The results show that LLMs preserve functional behavior in source code and specifications (both over 91%), but achieve modest verification success (31.4%). Using Gemini 2.5 Pro and providing formal contracts lead to higher success rates in our setting. Moreover, most errors (94%) come from LLMs' mistakes in the domain-specific knowledge of SL verifiers such as VeriFast. These findings provide guidance for optimizing LLM-generated specifications for SL verifiers.**

***Index Terms*—formal verification, large language models, separation logic, prompt engineering, VeriFast, qualitative analysis**

I. INTRODUCTION

Advancements in *satisfiability modulo theories (SMT) solvers* [1] have driven the development of automated deductive verification tools—also known as auto-active, static, Hoare-logic [2], or design-by-contract verifiers. Such verifiers work by taking specifications written in a first-order mathematical logic on program components (*e.g.* pre- and post-conditions on functions), which denote the intended program behavior, and then statically analyzing the program to prove its adherence to the specifications. The analysis builds up proof obligations (*e.g.* at the end of a function the analysis's data should imply the postcondition) that are sent to an SMT solver to discharge. Successful verification of a program means that it will not violate its specifications at run time. Mature static verifiers, such as Dafny [3] and VeriFast [4], have been successfully applied in industry. Dafny was used to verify the functional correctness of Amazon Web Services' (AWS) authorization engine [5], and VeriFast was used to verify the absence of safety errors (*e.g.* dividing by zero or illegal memory accesses) in two smart card applets, a Linux device driver, and an embedded Linux network management component [6].

While promising, static verifiers have not achieved the ideal in automation where users only need to specify their intended behavior for code. Instead, verifiers require auxiliary specifications, such as loop invariants, inductive lemmas, and folds and unfolds of predicates to make up for limitations in proof automation. Furthermore, writing these specifications requires internal knowledge of the verifier's analysis and its status throughout the verification process. It should not come as a surprise then that it took 4 expert person-years to verify the AWS authorization engine [5]. In response, and with the success of large language models (LLMs) at generating code [7], [8], test-cases [9], [10], program invariants [11]–[15], and proofs in proof assistants [16]–[20], there has been a surge of recent work (all in the last 3 years) using LLMs to successfully generate specifications for such verifiers [21]–[37].

Only one of these works [35] concerns static verifiers based on *separation logic* [38], [39] (SL verifiers), such as Viper [40], Gillian [41], and VeriFast [42].

This understudy is unfortunate as SL verifiers in particular excel at verifying both the functional and safety properties of heap-manipulating programs at scale [6], but they suffer from requiring auxiliary specifications that are further complicated by reasoning about the heap. This leads to specification patterns that are unique to SL verifiers, *e.g.* numerous folds and unfolds, loop invariants, and lemmas breaking down or connecting predicates describing parts of the heap, and are worth studying as a result.

Therefore, this paper seeks to empirically and thoroughly evaluate how well existing LLMs perform when prompted to generate specifications for verifying C programs by the SL verifier VeriFast [42]. In particular, we examine both their

ability to preserve functional behavior and produce verifiable specifications across settings, and analyze the errors that arise when verification fails.

To conduct the study, we develop a dataset of 303 functions from 60 C programs with three different input versions of each function containing their functional behavior specified in natural language or VeriFast's specification language. Our dataset also includes fully specified and verified (ground truth) versions of these functions. Throughout our study, LLMs are prompted to generate missing specifications needed to verify a function given its input files; the resulting output files are then quantitatively and qualitatively analyzed. In particular, our study proceeds in two stages: 1) we conduct pilot studies on 45 functions to select the best performing prompting approach out of eight and three best performing LLMs out of ten, and then 2) we conduct a full study on the full dataset with the selected prompt approach and LLMs. For the pilot studies, we make selections based on verification success rates, the number of verification errors and their severity, and the number of functional behavior changes made to function pre- and postconditions and source code. In the full study, we manually code source code and pre- and postconditions in the output files to assess their functional behavior. We also calculate the percentage of functions passing verification with VeriFast. Finally, we sample 50 failing functions and manually record and fix errors until the function is verified. During this process, we record the error severity and the fix applied.

We find that LLMs largely preserve functional behavior in both source code (91.0%) and pre- and postconditions (92%), but have a fairly modest verification success rate at 31.4%. Moreover, we find Gemini 2.5 Pro and providing stronger pre- and postconditions in the input achieve the best performance for verifiability. Lastly, our error analysis shows that the majority of errors (94%) arise from limitations in the knowledge of SL verifier (*e.g.* syntax and heap reasoning), rather than general logical reasoning. We also provide actionable insights from our results, such as the suggestion to provide better heap related debugging information from a SL verifier to LLMs when verification fails and enabling the auto-feature of VeriFast–both to reduce verification errors and improve success rates.

In summary, this paper makes the following contributions:

- The first empirical and thorough study of LLMs' ability to generate specifications for program verification with an SL verifier using prompt engineering. In particular, we consider 8 different prompting approaches, 10 different LLMs, and 3 different input types.
- A dataset of 60 C programs with 303 functions that each have three versions with specifications describing their functional behavior in different formats. The dataset also includes a fourth fully specified and verified version of each program and function.
- Analysis results and actionable insights (§IV) that provide a baseline and guidance for building LLM-based verification environments for proof synthesis in SL verifiers, particularly VeriFast.
- 354 LLM specified C functions failing to verify with VeriFast along with human coded errors for each function that capture their severity, location, and fix/root cause.

II. Background & Motivation

A. The Benefits of Separation Logic Based Static Verifiers

Static verifiers verify programs by checking that each function adheres to its intended behavior written as pre- and postconditions in a mathematical logic. For heap-manipulating functions, this logic is a first-order logic extended with *separation logic* (SL) [38] or its variants, such as *implicit dynamic frames* (IDF) [43]. SL allows one to express how the shape of the heap or its parts change during execution; and, static verifiers supporting SL (*i.e.* SL verifiers) ensure the function respects this behavior. For example, Fig. 1 contains a simple C implementation of an `append_tail` function for singly-linked lists, which inserts a new node with a given value at the tail end of a given list. It also contains specifications highlighted in gray leading to the successful verification of `append_tail` with the SL verifier VeriFast [42]. The precondition `llist(head)` (line 26) specifies the input list starting at `head` must be acyclic, and the postcondition `llist(result)` (line 27) similarly denotes the returned list must be acyclic. In particular, the recursive predicate `lseg` (lines 4-7), which defines `llist` (line 9), uses *separation logic arrow* (*i.e.* the points-to assertion or singleton heap) to express heap locations and the values they hold, *e.g.* `from->val |-> ?val` states the heap location `from->val` holds the value `val`. The points-to assertion also denotes unique ownership of the given heap location, *e.g.* `from->val`. The `lseg` predicate also relies on the *separating conjunction* `&*&` to denote disjointness in memory. For example, `y->next |-> ?ynext &*& x->next |-> ?xnext` implies `y->next` and `x->next` are distinct heap locations, *i.e.* `y != x`. Then altogether, `lseg` specifies that all the heap locations for nodes in a list segment from the `from` node to the `to` node are separate from one another in memory, *i.e.* that the list segment is acyclic. It also denotes ownership of these heap locations. As a result, `append_tail`'s pre- and postconditions state: `append_tail` only accesses the heap locations for the given acyclic list, and returns the heap locations for an acyclic list.

By supporting SL, SL verifiers are able to assure heap-manipulating programs for functional properties involving the heap, such as preservation of list acyclicness. Further, SL verifiers utilize the ownership and separation constraints from SL to guarantee the absence of access errors, such as dangling or null-pointer dereferences, and memory leaks [1].

B. The Specification Burden of SL Verifiers

Unfortunately, SL verifiers not only require traditional auxiliary specifications (*e.g.* predicates, loop invariants, and inductive lemmas), but also additional complexity from ownership tracking and checking unique to SL verifiers. Notice, in Fig.

[1] VeriFast uses malloc block (*e.g.* at line 7) to help detect memory leaks.

```
1  struct node { int val; struct node *next; };
2  typedef struct node* node;
3
4  /*@  predicate lseg(node from, node to) =
5       from == to ? true : from != 0
6         &*& from->val |-> ?val &*& from->next |-> ?next
7         &*& lseg(next, to) &*& malloc_block_node(from);
8
9    predicate llist(node head) = lseg(head,0);
10
11   lemma void lseg_merge(node x, node y, node z)
12   requires lseg(x,y) &*& lseg(y,z) &*& lseg(z,0);
13   ensures lseg(x,z) &*& lseg(z,0); {
14     open lseg(x,y);
15     if (x == y) {
16     } else {
17       lseg_merge(x->next,y,z);
18       open lseg(z,0);
19       close lseg(x,z);
20       close lseg(z,0);
21     }
22   }
23 @*/
24
25 node append_tail(node head, int val)
26   //@ requires llist(head);
27   //@ ensures llist(result);  {
28     //@ open llist(head);
29     //@ open lseg(head,0);
30     node new_node = malloc(sizeof(struct node));
31     if (new_node == 0) abort();
32     new_node->next = 0;
33     new_node->val = val;
34     //@ close lseg(new_node->next,0);
35     //@ close lseg(new_node,0);
36     if (head == 0) {
37       //@ close llist(new_node);
38       return new_node;
39     } else {
40       node curr = head;
41       //@ close lseg(head,curr);
42       while (curr->next != 0)
43         /*@  invariant curr != 0 &*& lseg(head,curr)
44              &*& curr->val |-> ?v &*& curr->next |-> ?cn
45              &*& lseg(cn,0) &*& malloc_block_node(curr);
46         @*/ {
47         node tmp = curr;
48         curr = curr->next;
49           //@ close lseg(cn,cn);
50           //@ close lseg(tmp,cn);
51           //@ lseg_merge(head,tmp,cn);
52           //@ open lseg(cn,0);
53       }
54       curr->next = new_node;
55         //@ close lseg(curr,0);
56         //@ lseg_merge(head,curr,0);
57         //@ open lseg(0,0);
58         //@ close llist(head);
59       return head;
60     }
61 }
```

Fig. 1: Static verification in VeriFast of the `append_tail` function for singly-linked lists.

1, verifying `append_tail` requires 34 lines of auxiliary specification code compared to 16 lines of program code[2].

In SL verifiers, *predicates* enable the expression of heap properties for recursive heap data structures, such as `lseg` and `llist` on lines 4-9 in Fig. 1 for lists. But, SL verifiers cannot reliably automate unrolling recursive predicates; so instead, *folds and unfolds* (*opens and closes* in VeriFast) must be specified to control the availability of predicate information during verification, including for heap access checks[3]. For example, `open llist(head)` on line 28 in Fig. 1 tells VeriFast to consume the `list(head)` predicate from `append_tail`'s precondition (line 26) and produce the predicate's body `lseg(head,0)` for use in the following open on line 29. The second open provides the `head->next |-> ?next` points-to assertion, which justifies `curr->next` in the while loop's condition (line 42) on entry. Conversely, the `close llist(head)` on line 58 tells VeriFast to consume `list(head)`'s body `lseg(head,0)` and produce `list(head)` itself, which is used to prove `append_tail`'s postcondition (line 27) after returning `head` (line 59).

Loop invariants, which specify properties preserved across all iterations of a loop, enable static verifiers to verify all execution paths of a loop. They are 1) checked to hold before the loop, and 2) used to verify the loop body similar to a function body given the loop invariants as both the pre- and postconditions. In SL verifiers, loop invariants must additionally specify ownership of heap locations accessed by the loop to ensure safe access. Following these constraints, the loop invariant for the while loop in Fig. 1 on lines 43-45 specifies that the current node is always non-null (line 43) and the footprint of the loop in three parts: the list segmented from its head to the current node (line 43), the points-to assertions for the current node (line 44), and the list segmented from the node after the current one to the end (line 45). The points-to assertions and list segment after the current node provide access to heap locations used in the loop (*e.g.* `curr->next` on lines 42 and 48). The list segment from the head to current node retains these heap locations for use after the loop.

Finally, SL verifiers need *inductive lemmas* to break down or connect properties about the heap. For example, at the end of `append_tail`, we need to leverage `lseg(curr,0)` (produced on line 55) and `lseg(head,curr)` (provided by loop invariant) to obtain `lseg(head,0)` for the postcondition (line 27). SL verifiers cannot automatically prove transitivity of list segments, so the inductive lemma `lseg_merge` specified on lines 11-22 provides this proof. Then, `lseg_merge` is called on line 56 to help prove the postcondition.

[2] Empty lines and trivial lines (*e.g.* with only {, }, /*@ or @*/) are excluded.

[3] Although VeriFast can infer some heap chunks (*e.g.*, `llist(head)` on line 58) automatically, fold/unfold cannot be totally eliminated in SL verifiers.

C. Large Language Models (LLMs) & Prompt Engineering

LLMs are transformer-based models pretrained on large-scale natural language and code corpora to learn general linguistic and structural patterns [44]–[48]. Rather than task-specific fine-tuning, LLMs can be adapted at inference time via in-context learning [49]–[52] with prompt engineering. Chain-of-Thought (CoT) [53] and Retrieval-Augmented Generation (RAG) [54] are widely used techniques for improving LLM performance on tasks requiring structured reasoning and domain knowledge. CoT encourages explicit intermediate reasoning, which has been shown to improve accuracy on complex reasoning tasks [55], [56]. RAG augments the model input with task-relevant contextual information retrieved from an external corpus, allowing the model to leverage additional background knowledge beyond what is implicitly captured during pretraining [57]–[59]. These prompting techniques enable LLMs to better leverage domain-specific knowledge and reasoning patterns, leading to improved performance on specialized tasks requiring structured reasoning and expertise, such as test generation [9], [60]–[62], code generation [63]–[65], and proof assistants [16], [20], [66]–[68].

In RAG, the retrieved context is usually retrieval using similarity between the input query and candidate contexts. Sparse retrieval (RAG-sparse) relies on lexical similarity, such as term overlap measured by traditional information retrieval methods (e.g., TF-IDF and BM25 [69]), while dense retrieval (RAG-dense) [70] selects context based on the similarity of neural embeddings, enabling retrieval of semantically related content even when surface forms differ.

D. VeriFast

VeriFast [42], [71] is an SL verifier being actively developed by the imec-DistriNet research group at KU Leuven in Leuven, Belgium. VeriFast supports the modular verification of single and multi-threaded C, Java, and unsafe Rust code for functional and memory safety properties. These properties may be expressed with inductive datatypes, primitive recursive pure functions over these datatypes, first-order logic over data values, abstract predicates, SL points-to assertions and separating conjunctions. To aid in the verification of such rich specifications, users may write loop invariants, opens and closes of predicates, and lemmas [42], [71] or rely on limited symbolic inference features where available [72]. VeriFast verifies a program using a forward symbolic execution algorithm, which sends assertions over data values to an SMT solver to be checked against the current path condition [42]. We use the default SMT solver supported by VeriFast, which is Z3 [73].

VeriFast was chosen due to its maturity and active development since 2011 (version 25.11 released on Nov. 27, 2025). It is supported on Windows, Linux, and macOS with easy to install binaries and has a VS Code extension. It also has detailed tutorials for C and Rust, detailed documentation, and Zulip chatroom support [71]. Finally, VeriFast's GitHub repository [71] contains 770 test C programs derived from real-world code to be considered for our test dataset (§III-A).

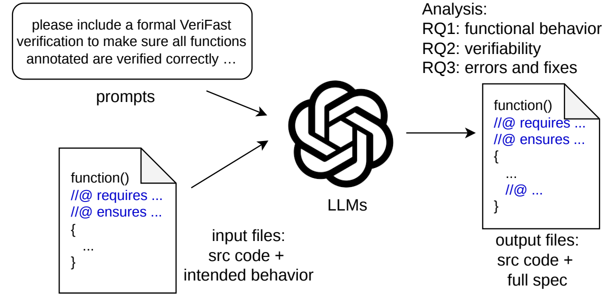

Fig. 2: The overview of the study

III. Study Design

To evaluate LLMs on generating VeriFast specifications, we aim to answer the following research questions:

RQ1 How well do LLMs preserve *functional behavior* when generating specifications for proofs of correctness in VeriFast across different prompts, input types, and LLMs?

RQ2 How successful are LLMs at generating specifications that result in *verified* code with VeriFast across different prompts, input types, and LLMs?

RQ3 When LLMs generate specifications that fail to verify with VeriFast, what *errors* are produced and how can they be fixed?

This section describes the empirical studies we perform to answer RQ1-3. First, we conduct two small-scale studies to pick the best prompting method out of 8 candidates and the best 3 LLMs out of 10 candidates for further study. These studies follow the same procedure as our larger-scale study just over the subset of our dataset (§III-A) not affected by data leakage. In particular, for each chosen LLM, input program, and prompt, we feed the input program and prompt to the LLM and record the output program produced. Input programs contain functions specified with their intended behavior in 1 of 3 forms, and prompts contain instructions and details (depending on prompt type) that instruct the model to produce a verified version of the input with VeriFast. Then, we evaluate the quality of the output program quantitatively and qualitatively for its preservation of functional behavior, verifiability, and errors. This workflow is illustrated in Fig. 2. The larger-scale study follows this workflow for the best prompt approach, best 3 LLMs, and all input types and programs in our dataset.

A. Test Dataset

Our dataset includes four different versions of 303 functions from 60 C programs (*i.e.*, ≈ 5 functions per program), with 50 from the VeriFast GitHub repository (§III-A1) and 10 developed by (§III-A2). Three of the variants are only specified with functional behavior in different formats (§III-A3) and serve as input files to the LLMs for verification. The fourth variant is fully specified and verified with VeriFast and serves as a ground truth for our analyses. The programs in this fourth form have an average of 43 lines of source code (ranging from 3 to 143) and 81 lines of specifications (ranging from 5 to

600). Of the 303 functions, 45 contain concurrency, 39 loops, 134 need recursive predicates for verification, and 85 are other non-trivial heap manipulating functions [4].

1) 50 Programs Derived from the VeriFast GitHub repository: VeriFast's GitHub repository contains 770 C programs, from example, tutorial, test, and library folders, that are specified with VeriFast's specification language and represent real code. From the 770 programs, we started with the 160 single-file ones from the example and tutorial folders. Of the 160 files, 46 do not have code or SL specifications, 49 are duplicates or in the tutorial used in prompts, 10 had verification errors with VeriFast 24.8.30 that we were unable to fix, and 5 could not have inputs prepared. After dropping these files, we are left with 50 verified programs–with 258 functions, an ave. of 44 lines of code (rge: 3-143), and an ave. of 91 lines of specifications (rge: 5-600)–making up the ground truth files in our dataset. Their input variants are described in §III-A3.

2) 10 Author Developed Programs: To test the LLMs on functions where data leakage is impossible, three authors manually crafted 10 new non-trivial single-file programs–writing both the C code and the VeriFast specifications–to be similar to programs in the 50-program dataset while not overlapping them. The resulting ground truth programs contain 45 functions, an ave. of 34 lines of code (rge: 17-55) and 32 lines of specifications (rge: 18-45). Six of the functions are concurrent, 3 contain loops, 20 require recursive predicates during verification, and 16 are other non-trivial SL dependent functions. Their input variants follow §III-A3.

3) Three Input Types: Inspired by Rego et al. [35], for each of the 60 ground truth programs (303 functions), we developed three different, partially specified versions of them, which we refer to as Natural Language (NL), Functional Behavior (FB), and Functional Behavior Plus (FBP) versions:

NL The program contains only the source code and natural language description of its functional behavior, which is defined as the state of memory in logic before and after the function. For example, Table I shows the NL version of the `append_tail` function on the left, which contains an informal description of the state of memory when entering and exiting the function in code comments (*e.g.* "the returned value is the head of an acyclic list").

FB The program contains source code and formal pre- and postconditions in VeriFast's specification language describing functional behavior. For example, Table I shows the FB format of `append_tail` in the middle, where predicates `llist` and `lseg` capture the shape of an acyclic list for use in the pre- and postcondition.

FBP The program contains source code and formal pre- and postconditions that describe the program's functional behavior along with other specification constructs required to pass verification (*e.g.* the allocation status of memory locations and constraints on integer bounds). For example, Table I shows the FBP format of `append_tail` on the right, which additionally contains `malloc_block_node(from)` to aid in proving the absence of memory leaks in `append_tail` and its callers. FBP is a baseline to compare NL and FB with.

[4]Note that only one of these tags is assigned to a function. Concurrency outranks loops, which outranks recursive predicates.

Note, we ensure all input files for a function specify the same functional behavior as each other and the ground truth.

B. Prompt Design and Selection with Pilot Study 1

We explore various prompting methods that vary in content and input granularity similarly to Misu et al. [26] and Du et al. [74]. We consider a Basic (baseline), Chain-of-thought (CoT) [53], RAG-sparse [69], and RAG-dense [70]. Our Basic prompt asks LLMs to generate verifiable specifications for a function given the input program and dependent standard VeriFast library files. The Basic prompt provides minimal guidance beyond requesting a single complete output code block. The CoT prompt further instructs an LLM step by step through the specification process. Step 1 directs the LLM to write preconditions and postconditions for each function with sub-steps for functional behavior, memory safety, and integer bounds. It also enforces correct placement between the function declaration and body. Step 2 instructs the LLM to generate loop invariants where applicable, ensuring they hold at each iteration and entail the post-loop conditions. Finally, Step 3 asks the LLM to insert auxiliary inline statements, such as `open`, `close`, and lemma applications. Building on Basic, RAG-sparse and RAG-dense additionally provide the model with VeriFast tutorial text and programs chosen with sparse and dense retrieval accordingly. Note, we do not include few-shot prompts, since RAG provides similar specified examples. We also consider 2 types of input granularity for each prompt type: input of the whole program (*w/o splitting*), or one function from the program (*w/ splitting*). All prompts used are in the replication package described in §VIII.

1) Pilot Study 1 for Prompt Selection: In total, we have 4 $\times$ 2 = 8 prompting methods. Since our analyses for RQ1 and RQ3 are primarily qualitative, it is infeasible to consider all 8 prompting methods across our entire dataset, inputs, and LLMs. Therefore, we conducted a pilot study with the 10 author-written programs' (45 functions') FBP inputs, GPT-4o, and our 8 prompts. We choose FBP since it is closest to a complete specification and GPT-4o due to its successes in related work [23], [26], [35]. The outputs are assessed for verification success rate, number of verification errors and their severity, and number of functional behavior changes in specifications and source code (defined in Section §III-D). The results show RAG-sparse *w/ splitting* has the highest verification success rate (64% compared to at most 56% for the others) and the fewest errors (63 compared to more than 67 for others). Moreover, it makes only 3 logical errors and 2 changes on functional behavior, meaning that LLMs are much less likely to make severe mistakes in the verification process or prove trivial results when using this prompting. Therefore, we selected RAG-sparse *w/ splitting* to use in our full study.

<table>
<tr><th>NL</th><th>FB</th><th>FBP</th></tr>
<tr>
<td>

```
/*** Description:
 * The 'append_tail' function.
 *
 * @param head: the head to a
     given list
 *
 * It requires that parameter head
      is the head of an acyclic
     list.
 * It makes sure that the returned
      value is the head of an
     acyclic list. */
 node append_tail(node head)
```

</td>
<td>

```
/*@predicate lseg(node from, node to)
     =
   from == to ? true : from != 0 &*&
       from->next |-> ?next &*& lseg(
       next, to);

predicate llist(node head) = lseg(
    head, 0); @*/

node append_tail(node head)
 //@requires llist(head);
 //@ensures llist(result);
```

</td>
<td>

```
/*@predicate lseg(node from, node to)
     = from == to ? true :
from != 0 &*& from->next |-> ?next
    &*& lseg(next, to) &*&
     malloc_block_node(from) ;

predicate llist(node head) = lseg(
    head, 0); @*/

node append_tail(node head)
 //@requires llist(head);
 //@ensures llist(result);
```

</td>
</tr>
</table>

Table I: Example of 3 input types for `append_tail` from Fig. 1, where the feature of each input type is highlighted in yellow.

C. Pilot Study 2 for LLM Selection

As with Pilot Study 1 (§III-B), it is infeasible to consider all 10 LLMs in our full study. As such, we conducted a similar pilot study using the selected RAG-sparse *w/ splitting* prompt to choose the best three performing LLMs out of Claude 3.7 Sonnet (Claude-3-7), Deepseek-chat, Deepseek-reasoner, Gemini 2.5 Pro (Gemini-2.5), Llama 3.3 70B (Llama-3.3), Llama 4 Scout (Llama-4), Qwen3-32B (Qwen3), GPT-4o, GPT-4o-mini, and o3. We first checked the verification success rate. If the success rate is acceptable, then we examined the number of verification errors, their severity and the number of functional behavior changes on specifications and code. On this basis, we excluded Llama-3.3, Llama-4, Qwen3, and GPT-4o-mini due to a low success rate of $\leq 24\%$ (others are $\geq 40\%$). Meanwhile, Claude-3-7, Gemini-2.5, and GPT-4o show the highest verification success rates at $\geq 60\%$ (the rest are $\leq 49\%$). Our qualitative analysis indicated they also have the fewest number of verification errors (≤ 73; others are ≥ 76) and a small number of functional behavior changes (≤ 2), so these three LLMs are selected.

There is a potential bias in fixing GPT-4o for prompt selection and then using this selected prompt to select LLMs; so, we conducted an extra study to mitigate this issue. We explored the success rate of all 10 LLMs for each of the 8 prompts (80 combinations) across the 45 newly created functions' FBP inputs (since Claude-3-7 deprecated by the time of this study, we used Claude Haiku 4.5 as a close alternative). The chosen RAG-sparse w/ splitting prompt and RAG-sparse w/o splitting prompt are tied for the largest median number of verified functions (16; others are ≤ 14.5) across all LLMs. Further, the 3 chosen LLMs result in the largest median number of verified functions (23, 22.5, and 20.5; others are ≤ 15) across all prompts. Thus, the selection of our chosen prompt and LLMs are justified.

D. Quantitative & Qualitative Analyses

This section describes the quantitative and qualitative analyses we performed on the LLMs' output files to answer RQ1-3. These analyses are performed per **function** rather than per **program** as in Rego et al. [35] to produce more precise results.

1) RQ1: How well do LLMs preserve functional behavior when generating specifications for proofs of correctness in VeriFast across different prompts, input types, and LLMs?: To answer RQ1, we qualitatively analyzed whether a function's source code and pre- and postconditions from the input file are changed by the LLM in violation of the function's intended behavior. This ensures the LLM is not trying to prove a different and potentially trivial property compared to the input. For our analysis, at least two authors independently examined the corresponding input and output files and assigned an appropriate code. They reviewed each other's result and resolved any disagreement. Our methodology here, including codes, is inspired by Rego et al. [35]. For the output pre- and postconditions, we coded their functional behavior as "equivalent", "strengthened", "weakened", or "other" when compared to the input pre- and postconditions. Here, "equivalent" means the pre- and postconditions have the same semantics, and "strengthened" ("weakened") means the output pre- and postconditions are semantically stronger (weaker) than the input ones. The rest of the cases are coded with "other", meaning the semantics of the pre- and postconditions are changed.

For source code, we coded the outputs' functional behavior as "unchanged" or "changed". A function is marked as "changed" when it contains a compiler error or its source code semantics is changed from the input, *e.g.* the output and input functions do not return the same values.

2) RQ2: How successful are LLMs at generating specifications that result in verified code with VeriFast across different prompts, input types, and LLMs?: To answer RQ2, we ran VeriFast with default options (*e.g.* checking integer overflow and heap chunk leakage) on the output functions with specifications and calculated the number and percentage of functions passing verification (*i.e.* the *verification success rate*). This metric provides a succinct overview of specification quality and is widely used in prior works [25]–[27], [34].

3) RQ3: When LLMs generate specifications that fail to verify with VeriFast, what errors are produced and how can they be fixed?: To analyze errors produced by LLMs and the fixes, we sampled 50 error-prone functions out of 303, and examined all failing variants out of the 9 variants (across 3 LLMs and 3 input types) for each sampled function, totaling 354 functions. Our sampling process proceeded as follows. First, we calculated the verification success rates for each of the 303 functions across their 9 variants. Then, we grouped the

<table>
<tr><th>Output file</th><th>Root cause & Fix</th><th>Severity & Component</th></tr>
<tr><td>/*@ requires queue(q, ?count) &*& count > 0;
ensures queue(q, count - 1) &*& ... @*/
int dequeue(struct queue *q)</td><td>Out-of-place precondition and postcondition.

So move them to after the function signature.</td><td>compile

PPP</td></tr>
<tr><td>/*@ predicate heap(struct heap_node *h, int minValue)
    = h == 0 ? minValue == INT_MAX : ... @*/

struct heap_node *heap_merge(struct heap_node *h1,
    struct heap_node *h2)
//@ requires heap(h1, ?min1) &*& heap(h2, ?min2);
/*@ ensures heap(result, ?minResult) &*&
    minResult == (min1 < min2 ? min1 : min2) ; @*/ {
   if (h1 == 0 && h2 == 0) return h2; ... }</td><td>At the first return, VeriFast needs to know min1 = min2 = INT_MAX, but only with heap(0, min1) &*& heap(0, min2), although they mean the same in logic by the definition of predicate heap.

So add //@ open heap(h1, min1); and //@ open heap(h2, min2); at the start of the function to expose the constraints on min1 and min2.</td><td>practical

OCAL</td></tr>
<tr><td>while (curr->next != 0)
/*@ invariant curr != 0 &*& malloc_block_node(curr)
    &*& curr->next |-> ?next &*& lseg(head, curr)
    &*& lseg(next, 0); @*/ {
   struct node *tmp = curr; curr = curr->next;
   //@ close lseg(curr, curr);
   //@ close lseg(tmp, curr);
}</td><td>After an iteration, the loop invariant requires lseg(head, curr), but only with lseg(head, tmp) &*& lseg(tmp, curr), where tmp is the value of curr when iteration begins.

So add a lemma to concatenate lseg(head, tmp) and lseg(tmp, curr) into the big one lseg(head, curr).</td><td>practical

LEM</td></tr>
<tr><td>/*@ predicate array(struct int_array *arr, list<int>
    cs) = arr->values[..10] |-> cs; @*/

for (int i = 0; i < 10; i++)
/*@ invariant ints_(arr->values, 10, ?vs) &*& 0 <= i
    &*& i <= 10; @*/
{ ... }
//@close array(arr, zeros(10));</td><td>After the loop, the loop invariant only shows “there exist 10 elements in the array”, not enough to show that “there exist 10 zeros in the array” required by the close.

So strengthen ints_(arr->values, 10, ?vs) to ints(values, i, zeros(i)) &*& ints_(values + i, 10 - i, _) in loop invariant to show the initialized zeros.</td><td>theoretical

LI</td></tr>
</table>

Table II: Examples of errors and analysis, where the errors in the output file are highlighted in red.

functions by their language feature (concurrency, loops, recursive predicate, or other), decided on a success rate threshold for each feature category (33%, 33%, 44%, 56% respectively), and set a quota for each category (7, 7, 22, and 14) to match the dataset distribution. Finally, we select functions below the thresholds, first picking all the functions from the author dataset; and then, randomly sampling additional ones up to the quota for each category.

For the analysis itself, we manually fixed failing output functions while recording errors, their type, location, and severity, and the corresponding fixes. VeriFast was used throughout this process to help locate and record errors and determine if the function is verified. This analysis is inspired by Rego et al. [35], but we additionally track error severity. Two authors independently analyzed each output function and afterwards discussed and resolved conflicts in their coding. If the two authors fixed the function in similar ways but coded differently, then they adjusted the coding through compromise. But, if the two authors fixed the function in different ways, then we adopted the fix that has fewer modifications. Each error is tagged by severity represented by these codes:

- Compile — This means VeriFast fails due to parsing or type check errors. For example, the first error in Table II is caused by an incorrectly placed pre- and postcondition that will not parse correctly.
- Practical — This error is during the verification process. It is applied when a specification is insufficient for automated reasoning, but not logically incorrect in general nor logically insufficient for proof obligations. For example, the second error in Table II is “practical” since `open heap(0, min1)` is missing and VeriFast treats `heap(0, min1)` and `min1 == INT_MAX` differently without this open even though they mean the same thing.
- Theoretical — This error occurs when verification fails due to logically incorrect or insufficient specifications, and is the most severe. For the last example in Table II, the invariant `ints_(arr->values, 10, ?vs)` is too weak to imply `array(arr, zeros(10))`’s body required by the `close` after the loop.

We also assigned each error a code corresponding to the specification component it pertains to:

- PPP — for predicate/fixpoint function/precondition/postcondition, specifying behavior and key proof obligations.
- OCAL — for open/close/assert/leak statements, which are simple auxiliary specifications.
- LEM — for lemma or built-in call, which convert verifier information to a new form.
- LI — for loop invariant.
- SRC — for source code.
- OTHER — for the rest of the cases (*e.g.* ghost variables).

As examples, Table II additionally contains the modified component tag and descriptions of the fix for each error listed.

	Claude-3-7	Gemini-2.5	GPT-4o	**Total**
NL	21	24	19	64 (7.0%)
FB	29	29	24	82 (9.0%)
FBP	36	45	19	100 (11.0%)
Total	86 (9.5%)	98 (10.8%)	62 (6.8%)	246 (9.0%)

Table III: Number and percentage of changed functional behavior of source code in output files

	eq	str	wk	oth
Total	2375 (87%)	146 (5%)	116 (4%)	90 (3%)

	eq	str	wk	oth	%eq+str
NL	649	103	114	43	83%
FB	859	25	2	23	97%
FBP	867	18	0	24	97%

	eq	str	wk	oth	%eq+str
Claude-3-7	779	30	43	57	89%
Gemini-2.5	802	72	17	18	96%
GPT-4o	794	44	56	15	92%

Table IV: Distribution of functional behavior in precondition and postcondition in output files

E. Experimental Setup

To carry out the experiment on inputs, LLMs and prompts, we implemented a configurable script to interact with LLMs using their APIs, where we set the temperature as 0 to promote reproducibility. Our environment is a Yoga 14sACH 2021 machine with 16 CPUs of AMD Ryzen 7 5800HS Creator Edition, 16 GiB memory and 1 TB SSD, which runs Ubuntu 24.04 LTS and Python 3.12. For prompting with RAG, we used BM25 for sparse encoding and TextEmbedding for dense encoding (both from Python library fastembed), and stored the vectors on a Qdrant node with 0.5 vCPU, 1GiB memory and 4GiB disk. We used VeriFast 24-08-30 [5] to verify output files.

IV. Results

For the full study, we fed all 3 input variants for each of our 303 functions and the RAG sparse *w/ splitting* prompt into 3 LLMs resulting in 2,727 output functions. We analyzed the outputs according to §III-D to answer RQ1-3 (stated in §III).

A. RQ1 - Functional Behavior

1) Source code: Table III presents the number and percent of output functions that have their source code behavior changed for each LLM, input type, and overall. Only 246/2,727 (9.0% of) output functions have their source code behavior changed, a promising result. Among input types, NL shows the least amount of change with 64/909 functions (7.0%) changed, while FBP shows the most with 100/909 (11.0%) changed. For models, GPT-4o performs the best with 62/909 (6.8%) changed; the other two models result in 86 and 98 functions (around 10%) changed. The Cohen Kappa value [75] of our coding here is 0.73, which is substantial agreement.

We further investigated the source code changes and present the results. Out of the 246 changed functions, 102 (42%) are marked as such due to small semantic edits, such as the addition/deletion of an if statement, change in variable usage, or giving library function declarations function bodies. We also observe edits leading to compiler errors, such as adding hallucinated header files or generating entire duplicate source code blocks, affecting 143/246 functions (58%).

[5] https://github.com/verifast/verifast/releases/tag/24.08.30

2) Preconditions and Postconditions: LLMs do well at preserving specified functional behavior, as 2,375/2,727 (87% of) output functions have pre- and postconditions (contracts) with equivalent behavior to the input specifications and an additional 146 (5%) are strengthened in comparison (Table IV). The NL input type performs noticeably worse than the FB and FBP types, but still preserves behavior in most cases. That is, for NL, 752/909 (83% of) output contracts are equivalent or strengthened compared to the input vs. 884/909 (97%) for FB and 885/909 (97%) for FBP. Despite this difference, all 3 LLMs preserve specified behavior similarly well (Claude-3-7: 809/909 (89%), Gemini-2.5: 874/909 (96%), GPT-4o: 838/909 (92%)). For this analysis, the Cohen Kappa is 0.38, *i.e.* fair agreement. There were many conflicts (275+) when coding the NL outputs due to ambiguity in natural language.

Upon further investigation, we see the NL input results in all but 2 weakened output contracts and nearly half (48%) of all other (changed) contracts. This is not surprising as NL provides models with the least amount of formal specification guidance and leaves room for ambiguity. Notably, Gemini-2.5 contributes the least to these numbers, often choosing to strengthen output contracts instead when given more freedom. Lastly, 66/90 (73% of) functions marked as having their specified behavior changed (*i.e.* marked as "other") are marked as such due to containing duplicate or missing specifications.

3) Discussion: Despite allowing LLMs to change source code and pre- and postconditions for verification, LLMs with prompting generally preserve the intended functional behavior of input source code and specifications. Users may also provide the input behavior of a function as formal specifications rather than natural language comments, and use Gemini-2.5 over other models to improve performance further. Gemini-2.5 is also notably robust to the ambiguity introduced by informal specifications, and is a good choice when formal specifications cannot be provided. However, LLMs cannot guarantee the absence of behavioral changes, and we witnessed a small percentage of such changes in our study. Some automated solutions exist to this problem, but come with trade-offs. First, LLMs can be instructed and/or forced to not modify the input code and specifications as with Banerjee et al. [36] and Chen et al.'s [33] work. While sound, this approach severely limits the applicability of LLMs for verification (because strengthening specifications and modifying code are required to verify some programs). A few recent work [33], [76] use test cases to detect incorrect or weak specifications in LLM outputs, but their completeness depends on the availability and coverage of test cases. Therefore, mitigating, detecting, and addressing behavioral changes in code and specifications by LLMs is important future work for tool builders.

	Claude-3-7	Gemini-2.5	GPT-4o	**Average**
NL	26.1%	36.0%	11.9%	24.6%
FB	24.1%	38.9%	25.4%	29.5%
FBP	39.3%	44.2%	36.3%	39.9%
Average	29.8%	39.7%	24.5%	31.4%

Table V: Verification success rate by input type and LLM

> **Summary of RQ1:** LLMs with prompting generally preserve the input behavior of source code and specifications, even when specifications are provided informally. But, using formal input specifications and/or Gemini-2.5 (over other models) can boost performance. While behavioral changes are rare (9.0% for source code and 7% for pre- and postconditions), they do happen and often result from adding/deleting if statements, changing variable usages, giving definitions to library function declarations, duplicating code and specifications, and hallucinating header files. Thus, achieving sound and complete guarantees of functional behavior preservation is an important open challenge.

B. RQ2 - Verifiability

1) Overall Verifiability of LLM-Generated Specifications: Across 2,727 generated output functions (303 functions $\times$ 3 input types $\times$ 3 LLMs), 855 verify successfully, yielding a 31.4% verification success rate (about 1 in 3 functions). We explore this rate further with respect to input type and LLM (§IV-B2) and functions' main language feature (§IV-B3).

2) Verifiability by Input Type and LLM: As shown in Table V, FBP has the highest verification rate at 39.9% followed by FB at 29.5% and NL at 24.6%. Moreover, across all three LLMs, verification rates are highest with FBP inputs and generally lowest with NL inputs (with the exception of Claude-3-7). Therefore, stronger formal guidance improves outcomes, *i.e.* supplying more complete input specifications leads to higher verification success. Similarly, verification performance varies based on LLM choice. Overall, Gemini-2.5 outperforms Claude-3-7 and GPT-4o (with ave. rates 39.7%, 29.8%, and 24.5% respectively). For each input type, Gemini-2.5 also achieves the highest success rates compared to the others.

3) Verifiability by Language Feature: Verification success rate varies substantially depending on a function's language features. Table VI shows that Normal SL functions have an 51.2% ave. success rate (much higher than the overall success rate of 31.4%). This suggests current LLMs can handle basic predicates and simple heap reasoning reasonably well. However, performance drops when recursive predicates are utilized (30.9%), and falls even further for concurrent (12.3%) functions and functions with loops (11.4%). These types of functions require more complex reasoning (especially heap reasoning) with predicates, loop invariants, and lemmas.

We also find that Gemini-2.5 performs significantly better than Claude-3-7 and GPT-4o on concurrent functions and functions with loops (Gemini-2.5: 21.5% and 21.4%, Claude-3-7: 10.4% and 9.4%, and GPT-4o: 5.2% and 3.4%). Note, Claude-3-7 also outperforms GPT-4o by roughly 5-6% on these functions. This suggests LLM choice matters when verification requires non-trivial auxiliary proof structure.

	Claude-3-7	Gemini-2.5	GPT-4o	**Average**
Normal SL	51.0%	54.1%	48.6%	51.2%
Recursive Predicate	28.9%	42.0%	21.9%	30.9%
Concurrency	10.4%	21.5%	5.2%	12.3%
Loop Invariant	9.4%	21.4%	3.4%	11.4%
Model Gap (High-Low)	41.6	32.7	45.2	39.8

Table VI: Verification success rate by language feature and LLM

4) Discussion: Prompting LLMs to generate verified functions with VeriFast achieves a modest 31.4% verification success rate (about 1 in 3 functions). However, upon further investigation, we can suggest strategies to improve performance. First, users should provide intended behavior in formal specification form to reduce ambiguity and provide more guidance to the models. Second, the cost of running our full study for each LLM is: Gemini-2.5 at $233, Claude-3-7 at $232, and GPT-4o at $171. While Gemini-2.5 is similarly priced to Claude-3-7, it significantly outperforms Claude-3-7, especially on harder functions containing concurrency and loops. GPT-4o has the lowest cost but also the worst performance. Considering, Gemini-2.5 also rarely changes the functional behavior of input programs, spending more on Gemini-2.5 for better overall performance may be worth it for verification tasks. Finally, success is higher on the simpler end of our dataset (Normal SL functions verify at 51.2%), but verification remains challenging for concurrent and loop containing programs (both near 12%). Thus, tackling failures in cases that require substantial auxiliary proof structure will significantly improve performance. We investigate these failure modes and provide actionable insights from them in §IV-C.

> **Summary of RQ2:** The selected prompt and LLMs show modest success at producing verified C code, and perform very poorly on tasks requiring more complex reasoning. Providing intended behavior as formal specifications can boost performance, as well as using Gemini-2.5 over other models. But, Gemini-2.5 has a higher cost, highlighting a trade-off between cost and performance.

C. RQ3 - Verification Errors

From our deep error analysis, we find the total number of verification errors across all LLMs and input types is 1,652 (Table VII) from the 354 selected failing functions. We present the results of our investigation into their severity, location, and fix/root cause. We also contextualize and provide actionable insights from the results.

1) Error Severity: Regarding error severity (Table VII), there are more compile–syntax and type–(383, 23%) and practical errors (1,167, 71%) than theoretical errors (102, 6%). Thus, prompt guided LLMs largely struggle to generate specifications that are compilable and sufficient for automated reasoning with VeriFast. Fortunately, they generate only a small fraction of specifications that are logically inconsistent. Upon further inspection of practical errors, 858 (51.9% of

all errors) arise from not satisfying a required heap chunk (*e.g.*, predicate or points-to assertion) during verification. For example, LLMs sometimes fail to write opens, closes, or lemmas that capture how the heap changes throughout a function for the verifier. This leads to failed heap chunk checks due to a lack of ownership. These errors would arise regardless of SL verifier, since these specification patterns are inherent to this broader class of verifiers. There are also VeriFast unique practical errors: 58 (3.5% of all) errors related to missing or incorrectly written malloc blocks for leak checking.

2) Error Location and Fix/Root Cause: Returning to Table VII, about half of all errors (767, 46%) can be fixed by modifying simple auxiliary specifications open/close/assert/leak (OCAL). They are either missing (58%), redundant (31%), misplaced (7%), or wrong (4%) in the output files. Errors on predicate/fixpoint function/precondition/postcondition (PPP) specifications are also significant (381, 23%), with 44% caused by compiler violations (syntax, 22% and type, 13%), 56% caused by verification failures, such as missing or incorrectly writing a malloc block (14%) or bounds check (14%). For lemma errors (LEM) (331, 20%), 62% of them are due to missing a lemma call and 23% of them are due to redundant lemma calls or definitions. For errors on loop invariants (LI) (61, 3.6%), 20% of them are caused by missing loop invariants, 25% are caused by them being too weak, and 20% are caused by them being too strong.

3) Errors by Input Type and LLM: We show the number of errors broken down by input type and severity on the left of Table VIII. NL results in the highest number of errors (688), followed by FB (518) and FBP (446). NL also leads to the majority (62%) of all compiler errors and nearly half of all theoretical errors (47%). This is caused by the lack of guidance natural language comments provide to LLMs on writing compilable specifications and the ambiguity in comments' intention. All input types lead to a large number of practical errors, but FBP results in the fewest (339 compared to 404 for NL and 424 for FB). This is unsurprising as FBP contains specifications that are closest to a verified solution.

Table VIII also breaks down the number of errors by LLM and severity. Gemini-2.5 has the fewest overall errors (424 compared to 546 and 682), practical errors (290 compared to 369 and 508), and theoretical errors (9 compared to 37 and 56). It also matches the other models on compiler errors. A closer inspection reveals that the lower volume of practical errors is primarily driven by the OCAL category (129 compared to 205 and 322). The discrepancy in theoretical errors stems from the categories of PPP (0 compared to 11 and 11) and loop invariants (0 compared to 7 and 18).

4) Discussion: LLMs with prompting primarily struggle to generate specifications that are compilable and sufficient for automated reasoning, such as OCAL specifications. As before, a straightforward way to reduce these errors is to provide formal input specifications and use Gemini-2.5. To reduce compiler errors, a repair loop with VeriFast's error messages can be considered because such messages contain information useful for locating and fixing compiler errors. A simpler solution would filter out failing cases and re-prompt until compilation succeeds, which is reasonable since compiler errors have a lower occurrence.

	PPP	OCAL	LEM	LI	SRC	OTHER	**Total**
Compile error	167	73	57	14	31	41	383
Practical error	192	656	270	22	18	9	1167
Theoretical error	22	38	4	25	6	7	102
Total	381	767	331	61	55	57	1652

Table VII: Errors by severity and fix location in deep analysis

	NL	FB	FBP	Claude-3-7	Gemini-2.5	GPT-4o
Compile	236	66	81	140	125	118
Practical	404	424	339	369	290	508
Theoretical	48	28	26	37	9	56
Total	688	518	446	546	424	682

Table VIII: Error severity by input type and LLM in deep analysis

For automated reasoning errors, particularly OCAL errors, we investigated VeriFast's auto-feature that can automatically generate open and close specifications in simple cases. We find that 39.4% open and close errors can be fixed with this feature. So, a mixture of LLM-based and symbolic techniques for specification/proof generation can provide better results. Additionally, since the vast majority of automate reasoning errors are related to insufficient reasoning about heap changes throughout a function, having VeriFast expose symbolic heap state information to LLMs in repair loops may also reduce such errors. Related, better debugging techniques and error messaging when verification failure occurs should be explored for SL verifiers to aid LLMs in generation/repair. Current error messaging for heap-related failures does not often correspond to or properly pin-point the root cause of failures.

Summary of RQ3: LLMs' errors are mainly on reasoning about SL verifier-specific knowledge (*e.g.* syntax and heap reasoning), rather than on general logic. Using techniques on the LLM side (*e.g.*, feedback loop with heap information) or in symbolic execution (*e.g.*, heap chunk searching) may be helpful to reduce such errors.

V. Threats to Validity

a) Internal: A key internal threat stems from benchmark construction: inputs are derived from verified VeriFast programs, ensuring a well-scoped task but limiting scenarios where LLMs must also refactor and verify arbitrary source code. The pilot studies to select the best prompt approach and best 3 LLMs may introduce selection bias. We show with a wider study without this bias that our selections are still the best choices (III-C). Reproducibility is another concern, as we analyzed only one output per function per configuration to control qualitative analysis cost. We set the temperature of the LLMs to 0 to reduce randomness. Our qualitative analyses have the authors apply codes and fixes, which is a subjective process open to interpretation. Thus, we made sure at least two authors code independently and discuss until an agreement is reached. For RQ1, inter-rater agreement is

moderate to substantial (Cohen Kappa is 0.73 and 0.38). For RQ3, we haven't found a proper metric due to its open-ended coding. Finally, since the majority of our test dataset comes from a public repository, data leakage is possible. Our 10 new programs used for LLM and prompt selection and the full study have not been leaked. Plus, the relatively low verifiability on programs (31.4%) suggests limited leakage effects.

b) External: External validity is limited by our choice of benchmarks, language, and verifier. Our dataset contains 60 programs and 303 functions, which may not capture the diversity of codebases in terms of size, libraries, and specification conventions. However, our dataset is representative of real-world code and is feature diverse (*e.g.* concurrency). We evaluated on C programs only, although VeriFast also supports Java and Rust. This is standard in related work, so we leave studying other programming languages to future work. Finally, we only study proof generation for VeriFast and not any other SL verifiers (*e.g.* Viper or Gillian). While similar, they still have different specification languages and degrees of automation from each other, which may affect results. However, errors related to heap reasoning are likely to generalize across SL verifiers, since reasoning about heap ownership and separation is fundamental to all SL systems.

VI. Related Work

Early work in specification generation for static verifiers is primarily symbolic [72], [77]–[79] or heuristic based [80], [81]. With improvements in transformer-based language models, there has been a surge in recent work exploring their synthesis capabilities. The vast majority of this recent work is for non-SL verifiers, such as Frama-C [82], OpenJML [83], Dafny [3], and Verus [4], which have different specification patterns (proofs) compared to SL verifiers.

For Frama-C, AutoSpec [22] few-shot prompts an LLM to fill in specifications one-by-one on a given C program for verification with Frama-C; while VeCoGen [21] prompts an LLM to generate both C programs and specifications. Janßen et al. [23] ask ChatGPT and Kamath et al. [24] ask multiple LLMs to generate inductive loop invariants validated with Frama-C. SpecGen [25] uses OpenJML to iteratively guide an LLM to generate specifications for verifying Java programs. For Dafny, Laurel [27] generates helper assertions using LLMs with novel prompting techniques; Silva et al. [37] further refines fault localization and specification inference to generate multiple assertions; Banerjee et al. [36] adds predefined proof strategies to LLMs and post-processes the generated Dafny annotations; Pascoal et al. [28] combines GPT-4o and Claude 3.5 Sonnet to generate loop invariants; and Poesia et al. [31] combines an LLM and search to add annotations iteratively to a Dafny method until it verifies. Misu et al. [26] empirically explore how well GPT-4 and PaLM-2 synthesize verified Dafny methods with prompting. Our empirical evaluations are similar except for our deeper error analysis which checks severity and fixes. Both Yao et al. [32] and Chen et al. [33] use GPT models to generate specifications backed by Verus. Yao et al. [32] breaks the verification tasks into smaller ones for GPT-4 to complete and combines the results via static analysis. Chen et al. [33] uses GPT-4o for self-evolving specification generation and fine-tunes a new LLM to take over this process. AutoVerus [34] uses multiple LLM agents to generate and refine different types of specifications. In contrast, we do not use LLM feedback loops, since we focus on the capabilities of off-the-shelf LLMs with prompt engineering.

Rego et al. [35] is the only work other than ours, which explores the efficacy of LLMs to generate specifications for SL verification. Our work is heavily inspired by theirs. We extend and refine their preliminary work to explore more LLMs, prompting approaches, and benchmarks. Further, all of our analyses operate at the more precise granularity of functions rather than whole programs, and our error analysis additionally records the severity of errors. We also offer actionable insights from our results, while Rego et al. [35] do not.

VII. Conclusion

This paper provides a comprehensive insight into how LLMs perform when asked to generate specifications for C programs that can be verified by VeriFast. After narrowing down the best performing mainstream LLMs to three and the prompt approach out of eight to one, we analyzed the output of these three LLMs across 303 functions using RAG-sparse (*w/ splitting*) with three different input types. Our results show that while LLMs do preserve functional behavior, their overall verification success is modest. Their performance improves when input specifications are stronger; and among the three models, Gemini 2.5 Pro consistently has the highest verification rates. Further qualitative analysis shows recurring error patterns related to heap reasoning. This highlights the current limitation of LLMs in producing precise specifications required by separation logic tools, rather than misunderstanding the program. Future work can leverage this knowledge to improve AI techniques and verifiers to support generating auxiliary specifications with LLMs.

VIII. Data Availability

The artifacts of this work, including the dataset, prompts, scripts for quantitative analysis and our qualitative analysis, are provided at https://zenodo.org/records/19570523.

References

[1] M. W. Moskewicz *et al.*, "Chaff: Engineering an efficient sat solver," in *Proc. of the 38th annual Design Automation Conf.*, 2001, pp. 530–535.

[2] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[3] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International conference on logic for programming artificial intelligence and reasoning*. Springer, 2010, pp. 348–370.

[4] A. Lattuada *et al.*, "Verus: Verifying rust progs. using linear ghost types," *Proc. of the ACM on PLs.*, vol. 7, no. OOPSLA1, pp. 286–315, 2023.

[5] A. Chakarov *et al.*, "Formally verified cloud-scale authorization," in *2025 IEEE/ACM 47th Int'l. Conf. on Softw. Eng. (ICSE)*. IEEE Computer Society, 2025, pp. 703–703.

[6] P. Philippaerts *et al.*, "Software verification with verifast: Industrial case studies," *Science of Computer Programming*, vol. 82, pp. 77–97, 2014.

[7] B. Chen *et al.*, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[8] S. Sarsa *et al.*, "Automatic generation of programming exercises and code explanations using large language models," in *Proc. of the 2022 ACM Conf. on Int'l. Computing Educ. Research-Vol. 1*, 2022, pp. 27–43.
[9] J. Wang *et al.*, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, 2024.
[10] C. S. Xia *et al.*, "Fuzz4all: Universal fuzzing with llms," in *Proc. of the IEEE/ACM 46th Int'l Conf. on Softw. Eng.*, 2024, pp. 1–13.
[11] M. D. Ernst *et al.*, "The daikon system for dynamic detection of likely invariants," *Science of computer prog.*, vol. 69, no. 1-3, pp. 35–45, 2007.
[12] K. Pei *et al.*, "Can large language models reason about program invariants?" in *Int'l Conf. on ML*. PMLR, 2023, pp. 27 496–27 520.
[13] D. Xie *et al.*, "Impact of large language models on generating software specifications," *arXiv preprint arXiv:2306.03324*, 2023.
[14] C. Sun *et al.*, "Classinvgen: Class invariant synthesis using large language models," in *Int'l Symp on AI Verif.* Springer, 2025, pp. 64–96.
[15] W. Cao *et al.*, "Clause2inv: A generate-combine-check framework for loop invariant inference," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 1009–1030, 2025.
[16] C. Zheng *et al.*, "Lyra: Orchestrating dual correction in automated theorem proving," *arXiv preprint arXiv:2309.15806*, 2023.
[17] K. Yang *et al.*, "Leandojo: Theorem proving with retrieval-augmented language models," *Adv. in Neural Info. Processing Syst.*, vol. 36, 2024.
[18] A. Q. Jiang *et al.*, "Lisa: Language models of isabelle proofs," in *6th Conf. on AI and Theorem Proving*, 2021, pp. 378–392.
[19] S. Welleck *et al.*, "Llmstep: Llm proofstep suggestions in lean," *arXiv preprint arXiv:2310.18457*, 2023.
[20] E. First *et al.*, "Baldur: Whole-proof generation and repair with large language models," in *Proc. of the 31st ACM Joint European Softw. Eng. Conf. and Symp. on the Found. of Softw. Eng.*, 2023, pp. 1229–1241.
[21] M. Sevenhuijsen *et al.*, "Vecogen: Automating generation of formally verified c code with large language models," in *2025 IEEE/ACM 13th Int'l Conf. on FM in SE (FormaliSE)*. IEEE, 2025, pp. 101–112.
[22] C. Wen *et al.*, "Enchanting program specification synthesis by large language models using static analysis and program verification," in *Int'l Conf. on Computer Aided Verification*. Springer, 2024, pp. 302–328.
[23] C. Janßen *et al.*, "Can chatgpt support software verification?" in *Int'l Conf. on Fundam. Approaches to SE*. Springer, 2024, pp. 266–279.
[24] A. Kamath *et al.*, "Leveraging llms for program verification," in *2024 FM in Computer-Aided Design (FMCAD)*. IEEE, 2024, pp. 107–118.
[25] L. Ma *et al.*, *SpecGen: Automated Generation of Formal Program Specifications via Large Language Models*. IEEE Press, 2025, p. 16–28. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00129
[26] M. R. H. Misu *et al.*, "Towards ai-assisted synthesis of verified dafny methods," *Proc. of the ACM on SE*, vol. 1, no. FSE, pp. 812–835, 2024.
[27] E. Mugnier *et al.*, "Laurel: Unblocking automated verification with large language models," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA1, Apr. 2025. [Online]. Available: https://doi.org/10.1145/3720499
[28] J. Pascoal Faria *et al.*, "Automatic generation of loop invariants in dafny with large language models," in *International Conference on Fundamentals of Software Engineering*. Springer, 2025, pp. 138–154.
[29] Á. F. Silva *et al.*, "Leveraging large language models to boost dafny's developers productivity," in *Proc. of the 2024 IEEE/ACM 12th Int'l Conf. on FM in SE (FormaliSE)*, 2024, pp. 138–142.
[30] S. K. Lahiri, "Evaluating llm-driven user-intent formalization for verification-aware languages," in *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2024*, 2024, p. 142.
[31] G. Poesia *et al.*, "dafny-annotator: Ai-assisted verification of dafny programs," *arXiv preprint arXiv:2411.15143*, 2024.
[32] J. Yao *et al.*, "Leveraging large language models for automated proof synthesis in rust," *arXiv preprint arXiv:2311.03739*, 2023.
[33] T. Chen *et al.*, "Automated proof generation for rust code via self-evolution," in *The 13th Int'l Conf. on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=2NqssmiXLu
[34] C. Yang *et al.*, "Autoverus: Automated proof generation for rust code," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, Oct. 2025. [Online]. Available: https://doi.org/10.1145/3763174
[35] M. Rego *et al.*, "Evaluating the ability of gpt-4o to generate verifiable specifications in verifast," in *2025 IEEE/ACM 2nd Int'l Conf on AI Foundation Models and Softw. Eng. (Forge)*, 2025, pp. 246–251.
[36] D. Banerjee *et al.*, "Dafnypro: Llm-assisted automated verification for dafny programs," *arXiv preprint arXiv:2601.05385*, 2026.
[37] Á. Silva *et al.*, "Inferring multiple helper dafny assertions with llms," *arXiv preprint arXiv:2511.00125*, 2025.
[38] J. C. Reynolds, "Sep. logic: A logic for shared mutable data structs." in *Proc. 17th Ann. IEEE Symp. on Logic in CS*. IEEE, 2002, pp. 55–74.
[39] M. Parkinson *et al.*, "Sep. logic and abstraction," in *Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Princ. of PLs.*, 2005, pp. 247–258.
[40] P. Müller *et al.*, "Viper: A verification infrastructure for permission-based reasoning," in *Verification, Model Checking, and Abstract Interpretation: 17th Int'l Conf., VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*. Springer, 2016, pp. 41–62.
[41] J. Fragoso Santos *et al.*, "Gillian, part i: a multi-language platform for symbolic execution," in *Proc. of the 41st ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, 2020, pp. 927–942.
[42] B. Jacobs *et al.*, "Verifast: A powerful, sound, predictable, fast verifier for c and java," in *NASA FM Symp.* Springer, 2011, pp. 41–55.
[43] J. Smans *et al.*, "Implicit dynamic frames," *ACM Transactions on Prog. Langs. and Syst. (TOPLAS)*, vol. 34, no. 1, pp. 1–58, 2012.
[44] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
[45] H. Touvron *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.
[46] A. Liu *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.
[47] E. Nijkamp *et al.*, "Codegen: An open large language model for code with multi-turn program synth." *arXiv preprint arXiv:2203.13474*, 2022.
[48] A. Lozhkov *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.
[49] T. Brown *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
[50] A. Chowdhery *et al.*, "Palm: Scaling language modeling with pathways," *J. of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.
[51] A. Grattafiori *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.
[52] S. M. Xie *et al.*, "An explanation of in-context learning as implicit bayesian inference," *arXiv preprint arXiv:2111.02080*, 2021.
[53] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in llms," *Adv. in neural info. proc. syst.*, vol. 35, pp. 24 824–24 837, 2022.
[54] P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Adv. in neural info. proc. syst.*, vol. 33, pp. 9459–9474, 2020.
[55] A. Lewkowycz *et al.*, "Solving quantitative reasoning probls. with lang. models," *Adv. in neural info. proc. syst.*, vol. 35, pp. 3843–3857, 2022.
[56] E. Zelikman *et al.*, "Star: Bootstrapping reasoning with reasoning," *Adv. in Neural Info. Proc. Syst.*, vol. 35, pp. 15 476–15 488, 2022.
[57] J. Yang *et al.*, "Swe-agent: Agent-comptr. interfaces enable autom. softw. eng." *Adv. in Neural Info. Proc. Syst.*, vol. 37, pp. 50 528–50 652, 2024.
[58] F. Zhang *et al.*, "Repocoder: Repository-level code completion through iterative retrieval and gen." *arXiv preprint arXiv:2303.12570*, 2023.
[59] Q. Luo *et al.*, "Repoagent: An llm-powered open-source framework for repository-level code documentation generation," *arXiv preprint arXiv:2402.16667*, 2024.
[60] M. Zheng *et al.*, "Llms for validating network protocol parsers," in *2025 IEEE Secur. and Priv. Wkshps. (SPW)*. IEEE, 2025, pp. 56–64.
[61] Y. Deng *et al.*, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proc. of the 32nd ACM SIGSOFT int'l symp. on softw. test. and anal.*, 2023, pp. 423–435.
[62] C. Lemieux *et al.*, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th Int'l Conf. on Softw. Eng. (ICSE)*. IEEE, 2023, pp. 919–931.
[63] Y. Zhang *et al.*, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.
[64] C. E. Jimenez *et al.*, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.
[65] Y. Ding *et al.*, "Cocomic: Code completion by jointly modeling in-file and cross-file context," in *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, 2024, pp. 3433–3445.
[66] K. Thompson *et al.*, "Rango: Adaptive retrieval-augmented proving for automated software verification," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ser. ICSE '25. IEEE Press, 2025, p. 347–359. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00161
[67] M. Lu *et al.*, "Adaptive proof refinement with llm-guided strategy selection," *arXiv preprint arXiv:2510.25103*, 2025.

[68] M. Mikuła *et al.*, "Magnushammer: A transformer-based approach to premise selection," in *The 12th Int'l Conf. on Learning Rep.*, 2024. [Online]. Available: https://openreview.net/forum?id=oYjPk8mqAV
[69] S. Robertson *et al.*, *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009, vol. 4.
[70] V. Karpukhin *et al.*, "Dense passage retrieval for open-domain question answering." in *EMNLP (1)*, 2020, pp. 6769–6781.
[71] B. Jacobs *et al.*, "Verifast 25.11," https://github.com/verifast/verifast, 2025.
[72] F. Vogels *et al.*, "Annotation inference for separation logic based verifiers," in *International Conference on Formal Methods for Open Object-Based Distributed Systems*. Springer, 2011, pp. 319–333.
[73] L. De Moura *et al.*, "Z3: An efficient smt solver," in *Int'l Conf. on Tools & Algo. for the Const. and Anal. of Syst.* Springer, 2008, pp. 337–340.
[74] X. Du *et al.*, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639219
[75] A. J. Viera *et al.*, "Understanding interobserver agreement: the kappa statistic," *Fam med*, vol. 37, no. 5, pp. 360–363, 2005.
[76] M. Endres *et al.*, "Can large language models transform natural language intent into formal method postconditions?" *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1889–1912, 2024.
[77] Q.-T. Ta *et al.*, "Automated lemma synth. in symbolic-heap sep. logic," *Proc. of the ACM on Prog. Langs.*, vol. 2, no. POPL, pp. 1–29, 2017.
[78] C. Calcagno *et al.*, "Compositional shape analysis by means of bi-abduction," in *Proc. of the 36th annual ACM SIGPLAN-SIGACT symp. on Princ. of prog. langs.*, 2009, pp. 289–300.
[79] ——, "Compositional shape analysis by means of bi-abduction," *Journal of the ACM (JACM)*, vol. 58, no. 6, pp. 1–66, 2011.
[80] C. A. Furia *et al.*, "Inferring loop invariants using postconditions." *Fields of logic and computation*, vol. 6300, pp. 277–300, 2010.
[81] C. Flanagan *et al.*, "Houdini, an annotation assistant for esc/java," in *Int'l Symp. of Formal Methods Europe*. Springer, 2001, pp. 500–517.
[82] F. Kirchner *et al.*, "Frama-c: A software analysis perspective," *Formal aspects of computing*, vol. 27, no. 3, pp. 573–609, 2015.
[83] D. R. Cok, "Openjml: Jml for java 7 by extending openjdk," in *NASA formal methods symposium*. Springer, 2011, pp. 472–479.