VerusSeek: Retrieval-Augmented LLM-Based Proof Synthesis for Rust Programs

Yuchen Zhang[1],Cheng Wen[1], Jialun Cao[2], Dugang Liu[3], Zhiwu Xu[3], Yuwei Liu[4], Shengchao Qin[1]

[1] Guangzhou Institute of Technology, Xidian University, Guangzhou, China
[2] The Hong Kong University of Science and Technology, Kowloon, Hong Kong
[3] College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
[4] Institute of Software at Chinese Academy of Sciences, Beijing, China

1. Research Background

Rust has been widely used to build highly reliable system software, but memory safety alone is insufficient to guarantee the functional correctness of programs. Verus introduces formal specifications and SMT-based verification mechanisms into Rust, enabling developers to prove that programs satisfy formal contracts. However, in practical use, Verus still requires a substantial amount of manual effort. Given a program implementation and its functional contract, developers typically need to manually construct auxiliary proof content, such as loop invariants, assertions, lemmas, and proof code blocks, before the program can be successfully verified.

3. Motivating Example

```
1  spec fn rotation_split(len: usize, n: usize) -> int { 1. Contract: helper func
2      len - (n % len)
3  }
4
5  fn rotate_right(list: &Vec<u32>, n: usize) -> (new_list: Vec<u32>)
6      requires                                          2. Contract: pre/post condition
7          list.len() > 0,
8      ensures
9          new_list.len() == list.len(),
10         new_list@ == list@.subrange(rotation_split(list.len(), n)
11         as int, list@.len() as int).add(
12             list@.subrange(0, rotation_split(list.len(), n) as int),
13             ),
14 {
15     let rotation = n % list.len();
16     let split_index = list.len() - rotation;
17     let mut new_list = Vec::with_capacity(list.len());
18     let mut index = split_index;
19     while index < list.len()
20         invariant                                     3. LoopInvariant
21             list.len() > 0,                                   3.1 Bounds
22             list.len() == list.len(),
23             split_index <= index <= list.len(),
24             new_list.len() == index - split_index,            3.2 Frame
25             new_list@ == list@.subrange(split_index as int, index as int),
26             forall k: int 0 <= k < list.len() ==> list@[k] == list@[k],
27         decreases list.len() - index                  3.3 Termination
28     {
29         new_list.push(list[index]);
30         index += 1;
31     }
32     index = 0;
33     while index < split_index
34         invariant                                     4. LoopInvariant
35             list.len() > 0,                                   4.1 Bounds
36             list.len() == list.len(),
37             0 <= split_index <= list.len(),
38             0 <= index <= split_index,
39             new_list.len() == list.len() - split_index + index,    4.2 Frame
40             new_list@ == list@.subrange(split_index as int, list.len() as
       int).add(
41                 list@.subrange(0, index as int)),
42             forall k: int 0 <= k < list.len() ==> list@[k] == list@[k],
43         decreases split_index - index                 4.3 Termination
44     {
45         new_list.push(list[index]);
46         index += 1;
47     }
48     new_list
49 }
```

Figure 1. Verification Example from MBPP

The verification of rotate_right requires two relational loop invariants and involves proof information such as bounds, frame relations, and termination measures. The goal of VerusSeek is to retrieve and reuse such proof patterns from historically verified code.

4. Core Design

Semantic Proof-Construct Chunking: Organize verified code into reusable fine-grained proof constructs rather than complete functions or files, including function contracts, loop invariants, lemmas, proof code blocks, and assertions. Type-Aware Retrieval: Retrieve relevant proof constructs according to the type of the missing proof annotation. Hierarchical Context Expansion: Preserve the necessary proof context while avoiding interference from irrelevant information. Loop-Invariant Hint Synthesis: Use structural information such as bounds, frame facts, and termination measures to assist loop-invariant generation.

2. Overview of the VerusSeek

VerusSeek is a retrieval-augmented LLM-based proof synthesis approach for Rust/Verus programs. Given an incomplete or verification-failing Verus program and a knowledge base constructed from historically verified code, VerusSeek first retrieves evidence relevant to the current proof task, then prompts a large language model to generate the missing proof annotations, and uses the Verus verifier to check the generated patch. If verification fails, the system continues to repair the program based on the verification results; if verification succeeds, it outputs a revised program that passes Verus verification.

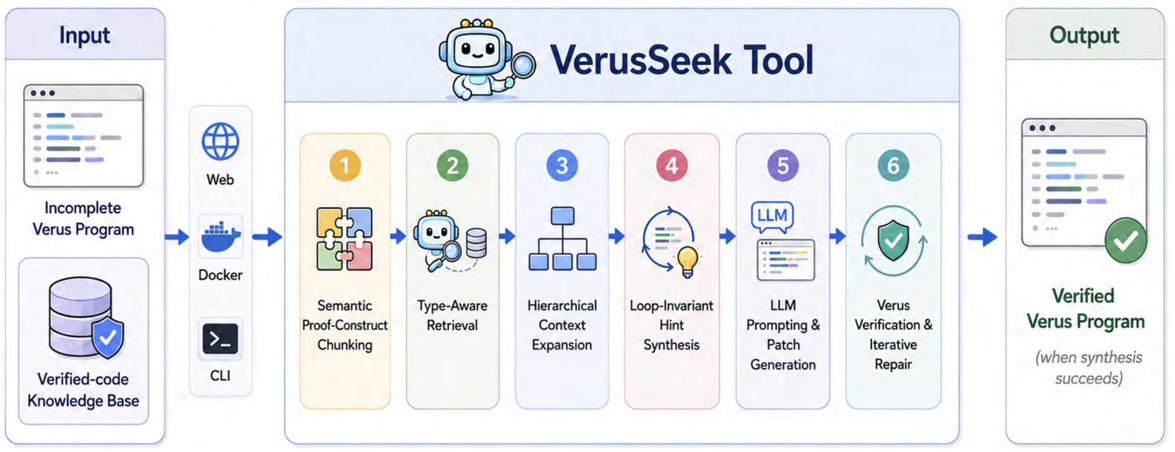


Figure 2. VerusSeek Workflow

① Semantic Proof-Construct Chunking ② Type-Aware Retrieval
③ Hierarchical Context Expansion ④ Loop-Invariant Hint Synthesis
⑤ LLM Prompting & Patch Generation ⑥ Verus Verification & Repair

5. Experimental Results

Table 1. VerusBench Evaluation

Suite	Step	AutoVerus [5]	RagVerus [6]	Basic RAG	Chunked RAG	VERUSSEEK
CloverBench (11)	Step 1	7	3	9	7	8
	Step 2	0	1	0	2	2
	Step 3	0	0	0	0	0
	Total	7/11	4/11	9/11	9/11	10/11
Diffy (38)	Step 1	1	12	2	27	26
	Step 2	2	1	0	4	5
	Step 3	1	4	3	4	1
	Total	4/38	17/38	5/38	35/38	32/38
Misc (23)	Step 1	11	11	12	11	15
	Step 2	0	0	2	3	1
	Step 3	0	0	1	0	1
	Total	11/23	11/23	15/23	14/23	17/23
MBPP (78)	Step 1	34	34	38	27	55
	Step 2	6	12	10	6	5
	Step 3	7	7	1	4	3
	Total	47/78	53/78	49/78	37/78	63/78
All (150)	Step 1	53	60	61	72	104
	Step 2	8	14	12	15	13
	Step 3	8	11	5	8	5
	Total	69/150	85/150	78/150	95/150	122/150

6. Conclusion

VerusSeek targets LLM-assisted Verus proof development and addresses the mismatch in retrieval granularity found in traditional retrieval methods. It organizes historically verified code into fine-grained proof constructs, supplements retrieved results with the minimal yet necessary hierarchical context, and combines loop-invariant hints with Verus-guided iterative repair. Experimental results on 150 VerusBench tasks show that VerusSeek significantly improves the verification success rate compared with existing LLM methods and retrieval-augmented baselines, demonstrating that semantic chunking and hierarchical context expansion can effectively support LLM-assisted formal verification of real-world Rust/Verus programs.

Web Docker CLI VerusSeek