

西安电子科技大学
XIDIAN UNIVERSITY

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

Yuchen Zhang, Cheng Wen, Jialun Cao, Dugang Liu, Zhiwu Xu,
Yuwei Liu and Shengchao Qin

Xidian University / HKUST / Shenzhen University / ISCAS

Motivation: LLMs Still Struggle with Non-trivial Verus Proofs

- Verus enables functional verification for Rust, but proof completion still requires substantial manual effort.
- Developers often need to construct **loop invariants, assertions, lemmas, and proof blocks**.
- Existing LLM-based approaches can automate part of this process, but difficult tasks often require specific proof patterns that the model does not know.

```
1  spec fn is_divisible(n: int, divisor: int) -> bool {          1. Contract: helper function
2      (n % divisor) == 0
3  }
4
5  fn prime_num(n: u64) -> (result: bool)
6      requires                                                  2. Contract: pre/post condition
7          n >= 2,
8      ensures
9          result == (forallk: int 2 <= k < n ==> !is_divisible(n as int, k)),
10 {
11     if n <= 1 {
12         return false;
13     }
14     let mut index = 2;
15     while index < n
16         invariant                                             3. LoopInvariant
17             n >= 2,                                                   3.1 Bounds
18             2 <= index <= n,
19             forallk: int 2 <= k < index ==> !is_divisible(n as int, k),   3.2 Relation
20             decreases n - index,                                      3.3 Termination
21     {
22         if ((n % index) == 0) {
23             proof {                                           4. ProofBlock
24                 assert(is_divisible(n as int, index as int));
25                 assert(existsk: int 2 <= k < n && is_divisible(n as int, k));
26             }
27             return false;
28         }
29         index += 1;
30     }
31     true
32 }
```

Figure 1: A motivating example for Verus proof synthesis.

The challenge is not generating more code, but finding the right proof knowledge.

Motivating Example: What Proof Is Missing?

- What is missing?

Bounds: 2 ≤ index ≤ n

Semantic invariant:

all previously checked numbers do not divide n

Termination: decreases n - index

Proof assertions: connect a discovered divisor to the postcondition

```
1  spec fn is_divisible(n: int, divisor: int) -> bool {          1. Contract: helper function
2      (n % divisor) == 0
3  }
4
5  fn prime_num(n: u64) -> (result: bool)
6      requires                                                  2. Contract: pre/post condition
7          n >= 2,
8      ensures
9          result == (forallk: int 2 <= k < n ==> !is_divisible(n as int, k)),
10 {
11     if n <= 1 {
12         return false;
13     }
14     let mut index = 2;
15     while index < n
16         invariant                                             3. LoopInvariant
17             n >= 2,                                                          3.1 Bounds
18             2 <= index <= n,
19             forallk: int 2 <= k < index ==> !is_divisible(n as int, k),      3.2 Relation
20             decreases n - index,                                             3.3 Termination
21     {
22         if ((n % index) == 0) {
23             proof {                                           4. ProofBlock
24                 assert(is_divisible(n as int, index as int));
25                 assert(existsk: int 2 <= k < n && is_divisible(n as int, k));
26             }
27             return false;
28         }
29         index += 1;
30     }
31     true
32 }
```

Figure 1: A motivating example for Verus proof synthesis.

The executable code is simple; the difficult part is constructing the proof structure required by Verus.

Key Idea: Retrieve Proof Constructs, Not Entire Files

Traditional RAG

Verified file / function

↓

Large code context

↓

Many unrelated statements

↓

LLM must locate and adapt the proof pattern

Problem: semantic similarity in code does not always expose the proof fragment that actually matters.

VerusSeek

Index verified Verus code as semantic proof constructs:

Contracts | **Loop Invariants** | **Assertions** | **Lemmas** | **Proof Blocks**

Verifier feedback tells VerusSeek what proof type is missing.

failed loop obligation → **LoopInvariant + contract context**

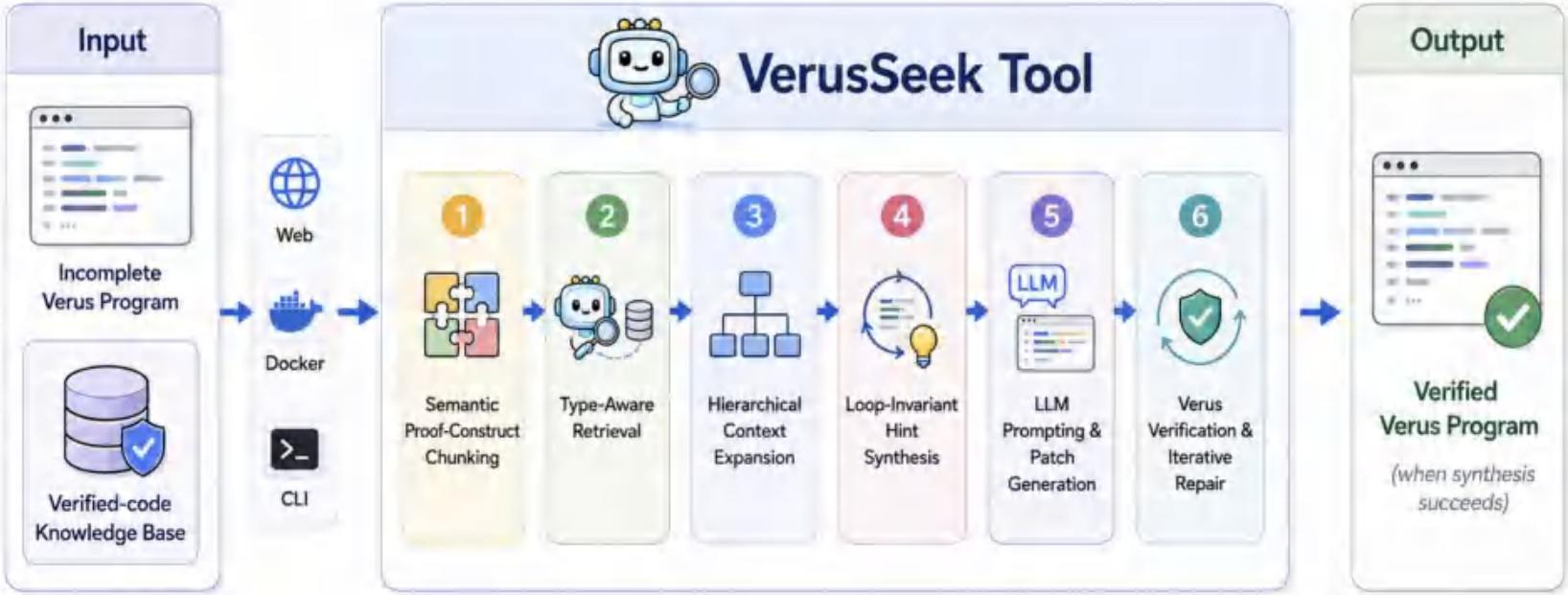
failed assertion → **Assertion / Proof Block evidence**

auxiliary proof need → **Lemma and related proof blocks**

Core idea: retrieve the proof structure that matches the current proof obligation, not just similar code.

Identify what type of proof evidence is missing and retrieve the most relevant constructs.

VerusSeek Workflow



The loop stops when the program is verified or the synthesis budget is exhausted.

1. Build proof knowledge: Chunk verified programs into semantic proof constructs.

2. Retrieve and complete proof context: Use verifier feedback to retrieve relevant constructs and restore necessary local context.

3. Generate, verify and Repair: The LLM generates proof annotations, and Verus checks every candidate. Verification errors are fed back into the next synthesis round.

Design Details: Proof Knowledge Construction and Retrieval

➢ Semantic Proof-Construct Chunking

VerusSeek decomposes previously verified Verus programs into reusable proof constructs, including contracts, loop invariants, assertions, lemmas, and proof blocks. Each construct is indexed separately with lightweight metadata for retrieval

➢ Type-Aware Retrieval

Given a failing program and verifier feedback, VerusSeek identifies what kind of proof evidence is missing and prioritizes constructs of the corresponding type. For example, loop failures retrieve loop invariants and related contracts, while assertion failures prioritize assertions and proof blocks.

Design Details: Context Restoration and Structural Hints

➢ Hierarchical context expansion

A retrieved proof construct is rarely useful in isolation. VerusSeek restores a bounded amount of surrounding context, such as enclosing function contracts, neighboring invariants, and related assertions. This keeps dependencies needed for reuse while avoiding an entire file or function.

➢ Loop-invariant hint synthesis

For loop-related tasks, retrieval may not provide all structural facts required by Verus. VerusSeek derives lightweight hints from program structure, including bounds from loop guards, frame conditions for unchanged inputs, and decreases terms from progress variables.

西安电子科技大学
XIDIAN UNIVERSITY

Evaluation: Experimental Setup

1 Benchmark: VerusBench

150 non-trivial verification tasks across four suites.

Suite	# Tasks
CloverBench	11
Diffy	38
MBPP	78
Misc	23
Total	150

Table 4 totals:

- Executable LoC: 2849
- Spec LoC: 1252

2 Compared Baselines

- **AutoVerus:** LLM generate-verify-repair (no retrieval).
- **RagVerus:** Repository-level retrieval-augmented baseline.
- **Basic RAG:** Whole-file retrieval.
- **Chunked RAG:** Semantic chunking + type-aware retrieval + hierarchical context expansion (without loop invariant synthesis).
- **VerusSeek:** Full system (our approach).

3 Metrics

Verification Success Rate
Percentage of tasks for which the final solution is verified by Verus.

First-Shot Success Rate
Percentage of tasks solved in the first generation attempt (no verification failures).

Number of Generation Iterations
Average number of generation iterations / attempts before success.

4 LLM Configuration

Models: `gpt-4o, deepseek-v3.2`

Max tokens: `4096`

Temperature: `1.0`

Embeddings: `text-embedding-3-small` (1536 dimensions)

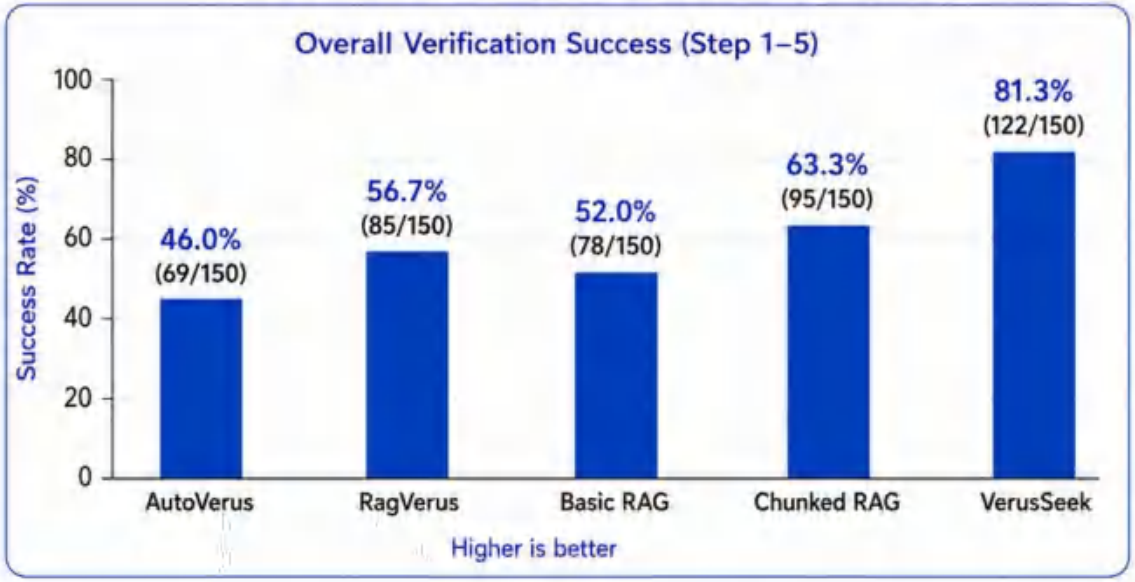
Per-task characteristics (Averages across all tasks)

Total LoC	Spec LoC	Proof LoC	Loop-invariant proof	Non-loop-inv. proof	Number of loops	Number of helper lemmas
32	8	10	8	2	1.6	0.07

- We evaluate VerusSeek on 150 non-trivial tasks from four VerusBench suites.
- We compare it with AutoVerus, RagVerus, Basic RAG, and Chunked RAG under the same experimental budget.
- We mainly measure verification success, first-shot success, and the number of generation attempts, using both GPT-4o and DeepSeek-v3.2.

Evaluation: Overall Effectiveness

Evaluation on VerusBench under GPT-4o (150 Tasks)





Takeaways

- **Construct-level retrieval** yields a major improvement over coarse retrieval.
- **VerusSeek** achieves the best overall and first-shot performance.
- Compared with AutoVerus, VerusSeek improves overall success from **46.0%** to **81.3%**.

- VerusSeek achieves the best overall verification success, proving 122/150 tasks.
- It also solves 104 tasks in the first attempt, clearly outperforming all baselines.
- The results show that better proof knowledge improves both overall effectiveness and first-shot proof quality.

Takeaways

- **Proof-aware Retrieval**

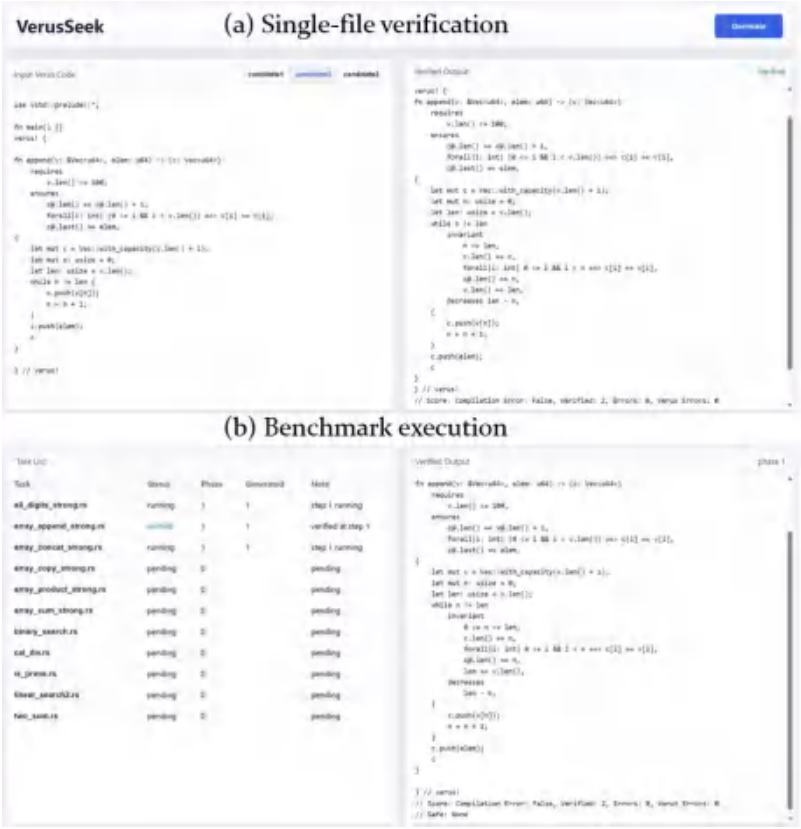
  Retrieve reusable proof constructs instead of large code regions.

- **Verifier-guided Synthesis**

  Combine retrieved evidence, structural hints, and Verus feedback.

- **Practical Tool Support**

  Web interface, Docker deployment, and command-line execution.



Web Docker CLI VerusSeek

Artifact & Demo Available

VerusSeek makes LLM-based proof synthesis more effective by grounding generation in reusable proof knowledge and formal verification.