Selene: Pioneering Automated Proof in Software Verification

Lichen Zhang[*]
Peking University
lczhang9653@gmail.com

Shuai Lu[†] and **Nan Duan**
Microsoft Research Asia
{shuailu,nanduan}@microsoft.com

Abstract

Ensuring correctness is a pivotal aspect of software engineering. Among various strategies available, software verification offers a definitive assurance of correctness. Nevertheless, writing verification proofs is resource-intensive and manpower-consuming, and there is a great need to automate this process. We introduce Selene in this paper, which is the first project-level automated proof benchmark constructed based on the real-world industrial-level operating system microkernel, seL4. Selene provides a comprehensive framework for end-to-end proof generation and a lightweight verification environment. Our experimental results with advanced large language models (LLMs), such as GPT-3.5-turbo and GPT-4, highlight the capabilities of LLMs in the domain of automated proof generation. Additionally, our further proposed augmentations indicate that the challenges presented by Selene can be mitigated in future research endeavors.

"Program testing can be used to show the presence of bugs, but never to show their absence."
– Dahl et al.'s (1972)

1 Introduction

Confirming the correctness of the software, *i.e.*, checking whether it adheres to the properties specified in the requirements, is advantageous for software engineering (SE). In contrast to testing, which is incomplete, verification provides rigorous guarantee of software correctness or incorrectness (D'Silva et al., 2008). Specifically, during testing, an adequate number of test cases are created and tested against the subject program. If the program violates a testing oracle or encounters other errors (*e.g.*, runtime error), a bug is found. However, the opposite conclusion cannot be guaranteed otherwise. Verification often involves the usage of a formal language and the corresponding prover. [1] This process requires formal proofs to rigorously demonstrate that the program satisfies the required properties, which can be verified by the prover.

In general, software verification involves two stages. ❶ The prerequisite specification stage translates the required properties and the subject program into the formal language, creating a to-be-proved proposition stating that "the program meets the properties", *a.k.a.*, the specification. ❷ The proof stage is supposed to generate proofs that prove the above specification and can be formally checked by the prover. Both stages consume significant resources and manpower, with the second stage being particularly demanding. *E.g.*, the seL4 operating system microkernel [2], which has been formally verified against strong functionality and security properties, requires 7 person-months dedicated to the specification stage and 11 person-years to the proof stage for correctness verification, and the amount of proof code in seL4 is even ten times more than that of the microkernel implementation itself (Klein et al., 2014). Therefore, in order to promote provable software, automated software verification, particularly automated proof, is highly desirable. As an early exploratory effort, in this paper, we explore to automate the major overhead.

Typically, automated proof in software verification is a conditional generation task from the specification to the proof, involving reasoning capabilities. Fortunately, large language models (LLMs) offer an opportunity, as they have demonstrated significant capacity in logic and reasoning at mathematical theorem proving (Jiang et al., 2022; First et al., 2023; Jiang et al., 2023). Only limited re-

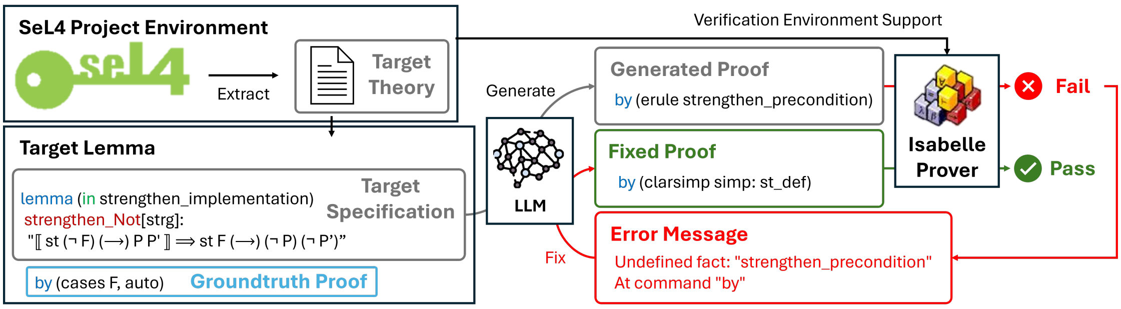
[*] Work done while Lichen Zhang serves as an intern at Microsoft Research Asia, Beijing, China.

[†] Correspondence to Shuai Lu

[1] Please note that there are other verification techniques such as model checking. We refer to it as methods involving interactive proof assistants in this paper.

[2] https://sel4.systems/

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1776–1789
August 11-16, 2024 ©2024 Association for Computational Linguistics



```
lemma (in strengthen_implementation)
 strengthen_Not[strg]:
 "⟦ st (¬ F) (⟶) P P' ⟧ ⟹ st F (⟶) (¬ P) (¬ P')"
by (cases F, auto)
```

```
by (erule strengthen_precondition)
```

```
by (clarsimp simp: st_def)
```

```
Undefined fact: "strengthen_precondition"
At command "by"
```

Figure 1: A demonstration of the Selene pipeline for automated proof generation (best viewed in color). Selene facilitates both the construction of proofs from scratch (indicated by the gray "generate" path) and the refinement of existing proofs augmented by error messages (highlighted by the red "fixing" path). To validate the correctness of the generated proofs, they are subjected to verification by the Isabelle prover within the authentic seL4 environment.

search has explored how to leverage LLM for code verification (Sun et al., 2023; Yao et al., 2023). And they only focus on function-level code verification, rather than a complete industrial-level software. A distinctive feature of industrial-level projects is the complex dependencies among lemmas and files, which makes automated proof even harder. In order to promote software verification, we propose a real-world industrial-level benchmark based on seL4 for automated proof, namely Selene. SeL4 is a high-assurance operating system microkernel, and it is comprehensively formally verified. The verification of seL4 is mainly based on the formal language of Isabelle (Isabelle, 2023), containing over 100k lines of code in Isabelle and thousands of lemmas (specification + proof), among which we randomly extract 360 for benchmarking. In the major pipeline (as presented in Figure 1), Selene inputs the specification of the target lemma, extracted from seL4, into the subject LLM, and checks the generated proof via the prover within the seL4 environment. As seL4 is a complicated system, Selene provides the complete dependency graph of lemmas, definitions and functions, along with a lightweight verification environment for each lemma to be evaluated. Due to the dependencies of the lemmas, almost the entire verification project needs to be rebuilt in order to check the generated proof, which can lead to a huge evaluation overhead (tens of minutes per lemma). Thence, Selene creates an isolated verification environment for each lemma to avoid duplicate construction and verification of the dependent lemmas, which enables efficient evaluation (it usually takes only a few minutes or even seconds to verify a generation).

We evaluate GPT-3.5-turbo (OpenAI, 2023a) and GPT-4 (OpenAI, 2023b) in the Selene pipeline. The experimental results demonstrate the feasibility of LLMs for automated proof in software verification. Still, we have identified some further challenges in Selene. ❶ The dependency graph of seL4 is complicated, and extracting facts to be applied from it can be hard for LLMs. ❷ The logic and reasoning process of a rather large proof may be beyond the capability of the subject LLMs. Even GPT-4 has difficulty in solving the rather difficult categories in Selene. Therefore, to address the challenges, we introduce three distinct augmentations, *i.e.*, similar lemma augmentation, dependency augmentation and fixing augmentation. These augmentations yield varying improvements across the Selene's different categories. Despite the inherent difficulties, our experimental results with these augmentations offer promising indications that the challenges posed by Selene are surmountable.

The main contributions of this paper can be summarized as below.

- We introduce the Selene benchmark, tailored for project-level automated proof in software verification, grounded in the real-world industrial-level project of the seL4 operating system microkernel.
- We introduce the technique of lemma isolation, which facilitates a lightweight verification environment capable of handling the complexities inherent in systems such as seL4.
- Our experiments with GPT-3.5-turbo and GPT-4 demonstrate the potential of LLMs in automated proof generation in software verification.
- We incorporate augmentations into the framework, which mitigate some of the challenges encountered within Selene and suggest promising avenues for future studies.

2 Related Work

2.1 Automated Theorem Proving by LLM

Automated theorem proving, especially mathematical theorem proving, has garnered significant attention in the field of artificial intelligence. LLMs have shown promising performance in proving formal theorems using proof assistants, such as Isabelle (Isabelle, 2023), Coq (Coq, 2023), and Lean (Lean, 2023). Thor (Jiang et al., 2022) integrates LLMs and hammer-based (Blanchette et al., 2016) provers in Isabelle. DSP (Jiang et al., 2023) leverages LLMs to produce structured formal sketches for auotomated proving. Besides, ProofNet (Azerbayev et al., 2023) and Baldur (First et al., 2023) both train or finetune LLMs on formal language corpora. When facing errors, Baldur (First et al., 2023) and Lyra (Zheng et al., 2023) refine the incorrect proofs with error messages.

In addition to the automatic approaches, there are existing benchmarks in the field of formal theorem proving. MiniF2F (Zheng et al., 2022) consists of mathematical problems from Olympiads competitions covering multiple formal languages. PISA (Jiang et al., 2021) includes the Archive of Formal Proofs in Isabelle. ProofNet (Azerbayev et al., 2023) contains mathematical problems in Lean along with parallel natural language descriptions. LeanDojo (Yang et al., 2023) builds a large benchmark in Lean with complete dependencies and the running environment.

2.2 Automated Software Verification

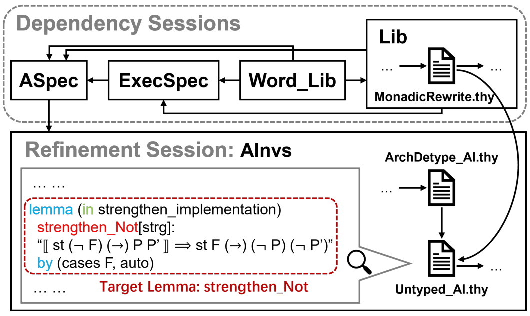
Software verification involves checking whether the software meets the requirements. In this paper, we leave alone the dynamic techniques (such as testing) that need to run the software, and only discuss the static formal verification techniques.

We briefly introduce four main techniques of automated software verification. Please refer to the survey for more details (D'Silva et al., 2008). ❶ Static analysis contains a collection of technologies (*e.g.*, pointer analysis, value range analysis) that analyze the behavior of the software without actual execution. By abstract interpretation (Cousot and Cousot, 1977), which approximately determines the undecidable software behaviors, one may check the correctness. ❷ Model checking traverses all plausible states of the software to determine whether a property holds (Emerson and Clarke, 1980; Queille and Sifakis, 1982). If the property is violated, the algorithm produces a reproducible trace, *i.e.*, a counterexample. Due to the large state space, algorithms for model checking are often abstracted or depth-bounded (Biere et al., 1999). ❸ Verification-aware programming languages, such as Dafny (Dafny, 2023) and Verus (Verus, 2023), supports formal specification through preconditions, postconditions, and loop invariants, *etc*., and employs first-order logic solvers (*e.g.*, Z3 (de Moura and Bjørner, 2008)) to automatically prove the specifications. They encourage the programmers to write correct specifications while writing the program, leaving the correctness verification burden to automatic solvers. ❹ Interactive verification relies on the interactive proof assistants. Both specifications and proofs during formal verification require substantial manual effort, and they are challenging to be fully automated. Hammers are still the major solutions to automating interactive verification. In the era of LLMs, it is highly feasible to explore automated proof in interactive verification.

Currently, there is limited research specifically addressing the problem of automated software verification with language models. Clover (Sun et al., 2023) introduces a benchmark for consistency checking among code, specification, and docstring, building on the verification-aware language of Dafny. Yao et al. (2023) proposes to use GPT-4 to write invariants, assertions, and other proof structures for Rust-based formal verification, in the short function-level code snippets.

3 Selene

We present the Selene benchmark in this paper (Figure 1), and evaluate LLMs' capabilities upon automated proof generation. The subject LLM's goal is to write proofs for the given specifications from seL4 and pass the verification. However, the verification process in the original seL4 environment is time-consuming. Given the impracticality of waiting for dozens of minutes to verify a single proof generated by the LLM, we construct Selene to align with the objective of lightweight evaluation. Drawing on the session design in Isabelle and seL4 (Section 3.1), we introduce lemma isolation (Section 3.2), which enables rapid verification of the target lemma (usually a few seconds). Due to the complexity of seL4, we further delve into some specific implementation details of Selene in Section 3.3.



```
lemma (in strengthen_implementation)
  strengthen_Not[strg]:
  "⟦ st (¬ F) (→) P P' ⟧ ⟹ st F (→) (¬ P) (¬ P')"
  by (cases F, auto)
```

Figure 2: An illustrative example of the seL4 verification structure. The arrows pointing from A to B indicate that B is dependent upon A, where A and B can be lemmas, theory files, or sessions, *etc*.

3.1 Preliminary of SeL4

SeL4 is a comprehensively formally verified operating system microkernel (Klein et al., 2014), providing an excellent example for software verification. Most of the verification work on seL4's functional correctness is based upon Isabelle (Isabelle, 2023), which is the basis of Selene.

Isabelle sessions. In the context of large verification projects, Isabelle employs sessions to effectively and efficiently organize the environment (Wenzel, 2023). The concept bears resemblance to the "package-class-function" structure in programming languages, with the design of "session-theory-lemma" in Isabelle. A session serves as a container for verification results typically centered around a specific topic, and maintains them in a persistent form. It enables easy accessibility without the need for repeated rebuilding lemmas within the session. Such design facilitates incremental development during software verification, allowing modifications to be made without necessitating a complete rebuild, as results in the unchanged and independent sessions remain persistent. Isabelle organizes the sessions using a series of ROOT files (please refer to Appendix B), which contain meta information such as the dependencies and the entry theory files for the sessions.

SeL4 verification structure. The verification of seL4 consists of multiple layers of refinement (De Roever and Engelhardt, 1998), progressing from high-level conceptual ideas to the concrete C implementation of the operating system [3]. Thence, there are many sessions involved in seL4 as shown in Figure 2, with some directly completing a refinement layer (*e.g.*, AInvs) while others providing dependencies (*e.g.*, ASpec and Lib) such as definitions and property specifications.

In our early studies about the verification process of seL4, we have identified some possible challenges. ❶ The dependencies in seL4 are highly complicated. A refinement session is typically dependent on multiple other sessions, creating a huge and complex dependency graph that makes it hard to identify the prerequisite components for proving a certain lemma in the refinement sessions. For instance, the session AInvs in Figure 2 is dependent on four sessions (Word_Lib, ExecSpec, ASpec, and Lib), and theories in AInvs depend not only on theories within AInvs (*e.g.*, Untyped_AI directly depends on ArchDetype_AI, and both of them are from AInvs), but also on lots of theories from the four dependency sessions (*e.g.*, Untyped_AI is also dependent upon MonadicRewrite from Lib). Such a large dependency graph usually contains hundreds or thousands of definitions, functions, and lemmas. Identifying prerequisite components from this dependency graph to prove lemmas in AInvs can be a great challenge. ❷ SeL4 is a systematic project that requires a lot of expert knowledge of operating system, *i.e.*, seL4 is sorely domain-specific. LLMs may not be quite familiar these fields, and therefore the quality of generated proofs may not be satisfying. ❸ Proofs in seL4 are often in the procedural style, *i.e.*, they specify a series of tactics to apply without describing the intermediate results. In contrast, proofs for general mathematical problems are often in the declarative style (Zheng et al., 2022), *i.e.*, they specify both the proving goals and the proving operations explicitly [4] (see Appendix A). Although previous work have demonstrated that LLMs can deal with declarative proofs (Jiang et al., 2022; First et al., 2023), the procedural style in seL4 may become a challenge.

3.2 Lemma Isolation in Selene

As outlined in Section 3.1, for large projects like seL4, Isabelle constructs the overall verification at the session granularity. However, it can lead to significant overhead during our evaluation – after generating a proof for the given lemma, one may have to wait for multiple minutes to build the corresponding session from scratch. To address

[3] A refinement formally proves that a concrete system corresponds to the abstract model and that all properties of the abstract model also hold for the concrete system.

[4] The variation in problem domains may account for such differences. Unlike pure and abstract mathematical problems, which are well-suited for the declarative style, software verification usually involves large, concrete, and complex systems, which may benefit from the procedural style (Harrison, 1996).

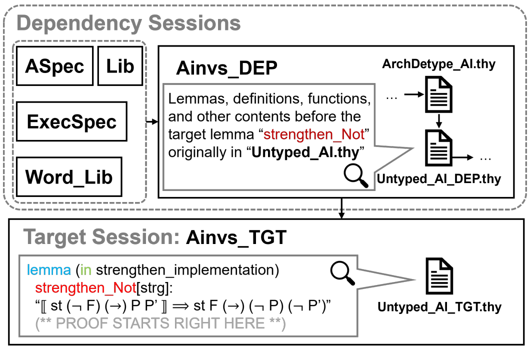

Dependency Sessions
ASpec
Lib
ExecSpec
Word_Lib
Ainvs_DEP
ArchDetype_AI.thy
Lemmas, definitions, functions, and other contents before the target lemma “strengthen_Not” originally in “Untyped_AI.thy”
...
...
Untyped_AI_DEP.thy
Target Session: Ainvs_TGT
lemma (in strengthen_implementation)
strengthen_Not[strg]:
“⟦ st (¬ F) (→) P P’ ⟧ ⟹ st F (→) (¬ P) (¬ P’)”
(** PROOF STARTS RIGHT HERE **)
Untyped_AI_TGT.thy

Figure 3: A working example of lemma isolation in Selene. Based on the original seL4 structure in Figure 2, we construct an isolated session (AInvs_TGT) along with a dependency session (AInvs_DEP) to facilitate efficient verification of the target lemma (strengthen_Not).

	P1	P2	P3	D
Extracted	1,995	2,496	928	45
Sampled	160	120	80	45
Correctly verified	144	109	64	43
Demonstration	5	5	5	5
Evaluation	139	104	59	38

Table 1: Statistics of Selene. P1, P2 and P3 denote the three difficulty levels for lemmas in procedural style, while D represents lemmas in declarative style.

this issue, we propose lemma isolation, wherein the target lemma is isolated from its dependencies, thereby avoiding repeated verification of the dependencies and creating a lightweight environment for Selene evaluation.

Following the working example presented in Figure 2, we isolate the target lemma strengthen_Not from the original session AInvs, depicted in Figure 3. The isolation process yields a minimal target session AInvs_TGT, which exclusively contains only the target lemma strengthen_Not. To verify AInvs_TGT, a dependency session AInvs_DEP is required. AInvs_DEP consists of theory files originally found in the dependency tree of Untyped_AI along with a new theory file (Untyped_AI_DEP) containing the contents preceding strength_Not in Untyped_AI. The theories in AInvs_DEP reconstruct the dependencies of the target lemma strength_Not in the original AInvs session.

AInvs_DEP, as well as other dependency sessions (ASpec, Lib, *etc.*), are verified once and remain fixed during evaluation. Accessing the persistent verification results in these sessions to verify AInvs_TGT takes little time. Lemma isolation can reduce the verification time to about $\frac{1}{3}$ of rebuilding from scratch (see Appendix C), creating a lightweight verification environment.

3.3 Key Know-how about Selene

In addition to the isolation design, the implementation of Selene involves many details, which can be attributed to the complexity of the seL4 system.

Lemma extraction. We gather theory files from the refinement sessions in seL4, and extract lemmas through a rough parser (*e.g.*, lemmas always begin with the token “lemma” or “throrem” and end with the token “qed”, “done” or a “by ...” statement). Lemmas within contexts or locales [5] are excluded from the process, because we find them incompatible with our design of lemma isolation. If the proof for a lemma exceeds 20 lines, we exclude it from Selene, as it may be too long and too challenging for LLMs. Finally, we collect 5,464 lemmas across 11 sessions from seL4.

Dependency session construction. We construct the dependency session by replacing only the target theory file in the directory. Taking Figure 2 and 3 for instance, we replace the theory file Untyped_AI in the session AInvs with the new theory Untyped_AI_DEP to build the dependency session AInvs_DEP. In the ROOT file, we set the entry to Untyped_AI_DEP and copy other meta information of AInvs to complete the construction of AInvs_DEP (please refer Appendix B). Even if there are additional theories in Untyped_AI_DEP, this setup will not include them into the dependency graph, providing correct dependencies to AInvs_TGT.

Lemma category. As mentioned earlier, we observed that the majority of proofs in seL4 are in procedural style (5,419 out of 5,464 lemmas collected), while only a small number are in declarative style (45). Procedural proofs typically applies a sequence of tactics to achieve the proving goal, and the length usually reflects the level of difficulty. For procedural style, we categorize lemmas into three difficulty levels according to the proof length: P1 (one single line), P2 (two to six lines), and P3 (seven to twenty lines). Lemmas from each difficulty level are randomly sampled to create the benchmark. As for lemmas in declarative style, all of them are included in the benchmark.

Correctness checking. It is important to check the correctness of the isolated sessions, as the imple-

[5] Contexts and locales in Isabelle are designed to deal with parametric theorems. Please refer to the documentation for more details (Ballarin, 2023).

mentation may not be guaranteed to be accurate. There are three potential causes of incorrect isolation: ❶ the extracted lemmas may be incomplete due to the limitation of keyword matching; ❷ copying meta information in ROOT files may result in configuration errors; ❸ the complex system setup of seL4 may lead to errors during lemma isolation. In addition, prior to evaluation, the dependency sessions should also be verified once to produce the necessary persistent results. We exclude those incorrect lemmas from Selene, leaving the remaining lemmas ready for evaluation. Table 1 lists the statistics of each step in Selene construction.

4 Evaluation

4.1 Evaluation Pipeline

Pipeline. The evaluation pipeline of Selene is presented in Figure 1. The subject LLM takes the specification, extracted from the isolated target session, as input, and generates a potential proof for it. The isolated target session is updated by appending the generated proof to the specification, and subsequently verified by the Isabelle prover. As designed in Section 3.2, since the dependency sessions have been already built once, the verification results are persistently available to the target session, thus the verification of the target session does not consume significant amount of time.

Metrics. We employ accuracy at k trials as the performance indicator, denoted as ACC#k. Specifically, the subject LLM independently generates k proofs using temperature sampling (Ficler and Goldberg, 2017; Fan et al., 2018; Caccia et al., 2020) and nucleus (top-p) sampling (Holtzman et al., 2020). If at least one of the k trials is successfully verified, ACC#k for the corresponding lemma is 1; otherwise, it is 0.

Prompt. The prompt includes an instruction, which specifies the task of automated proof, along with several demonstrations for in-context learning (Brown et al., 2020). Each demonstration consists of a specification and its corresponding groundtruth proof (please refer to Appendix D).

4.2 Evaluation Setup

We evaluate GPT-3.5-turbo (OpenAI, 2023a) and GPT-4 (OpenAI, 2023b) upon Selene. Within each set (P1, P2, P3, and D), we randomly select five lemmas as demonstrations, which remain fixed during our evaluation, and evaluate the remaining lemmas against the subject LLMs, as listed in Table 1. The subject LLMs take in the concatenation of the instruction, five demonstrations, and the target lemma specification, without additional augmentations, and generate proof trials.

	ACC	P1	P2	P3	D
GPT-3.5 -turbo	#1	28.1	2.9	0	0
	#5	35.3	5.8	0	5.3
GPT-4	#1	41.7	7.7	0	10.5
	#5	51.8	12.5	1.7	15.8

Table 2: Performance of GPT-3.5-turbo and GPT-4 against Selene (values in percentage).

ACC#1 and ACC#5 are assessed in our evaluation. The probability threshold (top-p) is set to 0.95, and the temperature is set to 0 for ACC#1 and 0.5 for ACC#5. Generation trials that exceed the token length of 2,048, contain the token"sorry" or "oops" (which can bypass the verification process, leading to false positive results), or take more than 10 minutes during verification (timeout) are all considered as failures.

4.3 Evaluation Result

The results are listed in Table 2. The results suggests that LLMs have the capacity to automate proof generation in Selene, with GPT-4 notably achieving 51.8% ACC#5 upon P1. Nevertheless, as the complexity of the proofs for procedural lemmas increases (P1→P3), the task becomes increasingly challenging for both GPT-3.5-turbo and GPT-4 models. In fact, both models struggle significantly when attempting to prove lemmas within the P3 category, which require comprehending an extensive dependency graph and employing more sophisticated reasoning capabilities. Interestingly, both the subject models perform better when addressing declarative lemmas (D) within Selene, as opposed to those categorized under P3, despite the proofs for most D category lemmas being of comparable length to those in P3, typically ranging from 7 to 20 lines. A plausible explanation could be that the inclusion of intermediate goals within declarative proofs mitigates the difficulty in logic and reasoning. In addition, we find that in many cases, GPT-4 adopts different proving strategies than the groundtruth (see cases in Appendix E), suggesting that the LLM is not simply memorizing.

Failure type. We analyze and categorize the errors made by GPT-4 during the evaluation process to better understand the challenges posed by Selene. The errors are classified into three distinct cate-

Error	P1	P2	P3	D
Total	81	96	59	34
Undefine	38(47%)	37(39%)	21(36%)	12(35%)
Logic	41(51%)	55(57%)	31(52%)	20(59%)
Other	2(2%)	4(4%)	7(12%)	2(6%)

Table 3: The composition of different types of errors made by GPT-4. The errors are collected in the ACC#1 setting evaluation. Outside the brackets are the absolute number of errors, inside the brackets are the percentages.

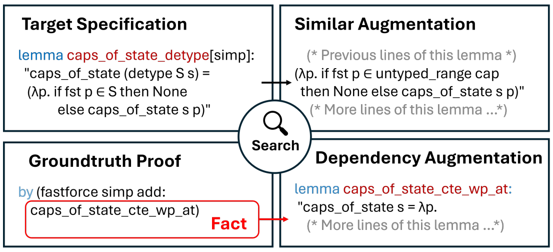
gories based on the nature of the error encountered: ❶ "undefined errors", where tactics not defined in seL4 are applied in the proofs, ❷ "logic errors", where the proof cannot be finished (*e.g.*, application of inappropriate tactics, presence of incomplete proving goals), and ❸ "other errors", including syntax errors, runtime errors, and other issues. The error composition is presented in Table 3. The majority of the errors (over a half) committed by GPT-4 can be attributed to its inadequate reasoning capability, which leads to unfinished proof goals (logic errors). A smaller, yet still significant, proportion of errors (undefined errors) stem from a lack of comprehensive knowledge of the dependencies within the entire seL4 project. Additionally, it is notable that GPT-4 barely makes syntax error, as most cases in other errors are refusal to generate proof [6], timeouts, and empty outputs (*e.g.*, exceeding the generation length), *etc.*

5 Augmentation

As previously discussed, LLMs exhibit significant potential for automated proof when evaluated against Selene, however, it is also evident that the task presents substantial challenges. We propose some augmentation techniques and evaluate them in our evaluation pipeline, with the aspiration that they may serve as a catalyst for further exploration in future studies.

5.1 Augmentation to Evaluation Pipeline

Similar lemma augmentation. SeL4 is an intricate piece of software, and as a consequence, its formal verification process is even more complex, involving a multitude of lemmas that can be similar (or even identical). The presence of these similar lemmas naturally offers an opportunity to augment the automated proof pipeline, and similar augmentation has been proven beneficial in tasks such as question-answering (Lewis et al., 2020) and code completion (Lu et al., 2022). Specifically, we build a retrieval library by segmenting theory files from seL4 into discrete chunks. The segmentation is guided by the blank lines in the text. Retrieval is performed through the BM25 algorithm (Robertson and Zaragoza, 2009) (the upper part of Figure 4), which involves querying the target specifications against the retrieval library to identify analogous text segments (*i.e.*, similar lemmas). To ensure the integrity of the experiment, the groundtruth proof is deliberately omitted from the retrieval process to prevent biases in the search results. During our evaluation, we select the initial ten lines from the chunk most closely resembling the target specification as the augmentation.



```
lemma caps_of_state_detype[simp]:
 "caps_of_state (detype S s) =
  (λp. if fst p ∈ S then None
        else caps_of_state s p)"
```

```
  (* Previous lines of this lemma *)
(λp. if fst p ∈ untyped_range cap
 then None else caps_of_state s p)"
  (* More lines of this lemma ...*)
```

```
by (fastforce simp add:
  caps_of_state_cte_wp_at)
```

```
lemma caps_of_state_cte_wp_at:
 "caps_of_state s = λp.
  (* More lines of this lemma ...*)
```

Figure 4: Demonstrative examples of similar lemma augmentation and dependency augmentation.

Dependency augmentation. The complex dependencies inherent in the seL4 project pose significant obstacles to LLMs when evaluated against Selene, as evidenced in Table 3. To mitigate this challenge, we introduce the dependency augmentation. Particularly, we extract the applied facts from the ground truth proof, and identify their origin by searching in the chunk library (those chunks not in the dependency sessions are omitted during this process), as shown in the lower part in Figure 4. The pinpointed definitions, functions, and lemmas are clearly integral to the proof of the target specification. And these elements are then provided to the subject LLM as augmentations, with the intention of simplifying the task by providing correct information for the model to apply. Ideally, this augmentation should alleviate the obstacles posed by dependencies, allowing the subject LLM to focus on applying the accurate information provided. However, during our practice, the absence of sophisticated tools means we cannot pinpoint every fact and its origin with complete precision. Consequently, the results of the dependency aug-

[6] For instance, GPT-4 may refuse our request by generating texts like "I cannot assist with this request". This situation does not happen much, but it is difficult to prevent it, even if we order it in the prompt to always generate a response. In addition, GPT-3.5-turbo produces much more refusal issues.

Augmentation	P1	P2	P3	D
GPT-4	41.7	7.7	0	10.5
+Similar	47.5	14.4	1.7	10.5
+Dependency	52.5	14.4	1.7	–
+Fixing	53.2	9.6	0	18.4

Table 4: ACC#1 of GPT-4 with augmentations evaluated against Selene (values in percentage). For the D category, we skip the dependency augmentation, due to the complexity of fact extraction in this category.

Aug.	Error			
	Total	Undef.	Logic	Other
GPT-4	81	38(47%)	41(51%)	2(2%)
+Similar	73	29(40%)	42(57%)	2(3%)
+Dependency	66	16(24%)	45(68%)	5(8%)
+Fixing	65	30(46%)	32(49%)	3(5%)

Table 5: The composition of errors made by GPT-4 with augmentations evaluated against Selene-P1.

mentation should be viewed as a potential upper limit of the subject LLM's capability in this context. We use the first five lines from the origin of each identified fact as the augmentation.

Fixing augmentation. When a proof attempt does not succeed, it is almost a standard procedure to examine the error message in order to fix the flawed proof (refer to Figure 1). The error message typically provides comprehensive feedback, such as the error type and the state of the proof at the moment of failure. There are existing studies that support the capability of LLMs to fix previously incorrect logic by incorporating error messages (First et al., 2023; Zheng et al., 2023; Chen et al., 2023), which make this augmentation even feasible when dealing with Selene. The evaluation is conducted as a two-round dialogue – if the subject LLM does not succeed in the first round, we feed the error message into the model and ask it to try again; if the subject LLM succeeds in the first trial, we do not carry out the second round of fixing.

We evaluate GPT-4 with the three augmentations, with the performance indicator of ACC#1. All other settings remain the same as in Section 4.2.

5.2 Augmentation Result

The results listed in Table 4 indicate the three augmentations lead to improvements across different categories. We also examine the error composition of GPT-4 with augmentations evaluated against P1, as listed in Table 5. In the below, we analyze the effect of each augmentation strategy and carry out some ablation studies.

Augmentation	P1	P2	P3	D
+TryAgain	49.6	7.7	0	10.5
+Similar & Fixing	61.9	20.2	1.7	7.9

Table 6: Ablation of augmentations (ACC#1 of GPT-4).

Similar augmentation. The similar augmentation is found to enhance performance upon procedural categories (P1-P3), indicating the utility in the augmented contexts; but it does not yield a significant effect upon the D category, suggesting a potential area for further investigation. According to Table 5, the similar augmentation marginally ameliorates the incidence of undefined errors without showing notable impact on logic errors. This improvement could be attributed to the facts introduced from the inclusion of similar lemmas.

Dependency augmentation. The dependency augmentation significantly improves GPT-4 on P1 (41.5%→52.5% in Table 4). As for the errors in Table 5, it is notable that the dependency augmentation results in a substantial diminution of undefined errors, corroborating our intended purpose.

Fixing augmentation. In Table 5, as the complexity of the proof increases (*i.e.*, P2 and P3), the fixing augmentation is less effective. This trend is expected since simple proofs (as in P1) typically contain straightforward errors that can be corrected in a single fixing attempt, whereas longer and more complex proofs may require multiple rounds of corrections. Also, as demonstrated in Table 5, there is a noticeable reduction in logical errors, which can be attributed to the integration of error messages. We further ablate by not providing the error message to GPT-4, only asking it to try again if the first attempt fails. The results are listed in the "TryAgain" row of Table 6. TryAgain brings limited improvement compared to fixing, suggesting that error messages are important.

Similar + dependency. We carry another ablation by combining similar and fixing augmentations together ("Similar&Fixing" in Table 6). Based on Table 5, the similar and the fixing augmentations improve the undefined fact and the logic error issues, respectively. Results show that combining both augmentations significantly improves GPT-4's performance upon P1 and P2. On D category, these two augmentations may have opposite effects, causing unexpected performance degradation (even worse than raw GPT-4). This phenomenon may be worthy of future exploration.

6 Conclusion

In this paper, we study the domain of automated proof within the context of software verification. We introduce Selene, which is a real-world industrial-level automated proof benchmark derived from the seL4 project. Selene provides a lightweight verification environment facilitated by lemma isolation with Isabelle sessions. The current framework supports end-to-end proof generation and evaluation, bolstered by supplementary augomentation. By evaluating against advanced LLMs such as GPT-3.5-turbo and GPT-4, we demonstrate the potential of LLMs in automated proof generation for software verification. Nevertheless, Selene poses formidable challenges that LLMs have yet to overcome fully. It is our hope that Selene will catalyze further research in this area, promoting advancements in software verification.

7 Limitation

We present some discussions on the limitations of Selene. As an early step of software verification, we consider addressing these limitations and challenges as our future work. Hopefully, we could offer insights that may serve as a catalyst for future studies in this field.

Dependency extraction. SeL4 contains a huge and complex dependency graph, posing a significant challenge in the accurate extraction of dependencies, *i.e.*, facts. Our analysis has revealed that undefined errors (*e.g.*, applying nonexistent facts) account for nearly half of GPT-4's failures in Selene. The dependency augmentation experiment has further proven the effectiveness and necessity of dependency in addressing this issue. One promising research direction may be to transition from providing LLMs with groundtruth facts as done in this paper, to employing advanced techniques (such as RAG (Lewis et al., 2020; Asai et al., 2023)) to automatically extract candidate facts directly from the codebase. We leave this as our future work.

Specification generation. There are two stages in software verification – the prerequisite specification stage and the proof stage. In this paper, we primarily concentrate on the automation of the proof stage, which constitutes the main bulk of the verification workload. However, it is important to acknowledge that the specification stage, which involves translation of properties and programs into formal languages, is not without its own set of challenges. This stage is not only time-consuming and resource-intensive but also necessitates substantial advancements in automation to enhance efficiency.

Proof state. The current pipeline of Selene only supports end-to-end proof generation, *i.e.*, the subject LLM generates the entire proof. Our experimental results indicate that LLMs possess the ability to prove lemmas within the less challenging P1 category. However, the effectiveness significantly diminishes when addressing lemmas from the more complex P3 category. This observation aligns with the experiences of human practitioners, who typically cannot construct proofs for P3 lemmas in a single attempt but instead progress incrementally, selecting suitable operations at each step based on the evolving proof state. To enhance the capability of LLMs in addressing P3 lemmas, it may be necessary to introduce the interactive proof state into the Selene pipeline in the future, thereby mimicking the human practitioners during proof construction.

To observe the intermediate state, PISA (Jiang et al., 2021) provides a practical solution. PISA implements a scala-Isabelle framework to glance at the proof state of the Isabelle verification system. With the proof state available, LLMs are allowed to generate proofs in an interactive manner. We believe that techniques in PISA are orthogonal with lemma isolation – lemma isolation allows rather quick session loading, and PISA provides and interactive environment. The incorporation of PISA and Selene is another future work.

References

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *CoRR*, abs/2310.11511.

Zhangir Azerbayev, Bartosz Piotrowski, Hailey Schoelkopf, Edward W. Ayers, Dragomir Radev, and Jeremy Avigad. 2023. Proofnet: Autoformalizing and formally proving undergraduate-level mathematics. *CoRR*, abs/2302.12433.

Clemens Ballarin. 2023. Tutorial on locales and locale interpretation.

Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer.

Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 2016. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Massimo Caccia, Lucas Caccia, William Fedus, Hugo Larochelle, Joelle Pineau, and Laurent Charlin. 2020. Language gans falling short. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *CoRR*, abs/2304.05128.

Coq. 2023. The coq proof assistant. https://coq.inria.fr/.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM.

Dafny. 2023. The dafny programming and verification language. https://dafny.org/.

Ole-Johan Dahl, Edsger W. Dijkstra, and Charles Antony Richard Hoare. 1972. *Structured programming*, volume 8 of *A.P.I.C. Studies in data processing*. Academic Press.

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer.

W-P De Roever and Kai Engelhardt. 1998. *Data refinement: model-oriented proof methods and their comparison*. 47. Cambridge University Press.

Vijay Victor D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 27(7):1165–1178.

E. Allen Emerson and Edmund M. Clarke. 1980. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer.

Angela Fan, Mike Lewis, and Yann N. Dauphin. 2018. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 889–898. Association for Computational Linguistics.

Jessica Ficler and Yoav Goldberg. 2017. Controlling linguistic style aspects in neural language generation. *CoRR*, abs/1707.02633.

Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1229–1241. ACM.

John Harrison. 1996. Proof style. In *Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 154–172. Springer.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Isabelle. 2023. The isabelle proof assistant. https://isabelle.in.tum.de/.

Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. 2021. Lisa: Language models of isabelle proofs. In *6th Conference on Artificial Intelligence and Theorem Proving*, pages 378–392.

Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygóźdź, Piotr Mił oś, Yuhuai Wu, and Mateja Jamnik. 2022. Thor: Wielding hammers to integrate language models and automated theorem provers. In *Advances in Neural Information Processing Systems*, volume 35, pages 8360–8373. Curran Associates, Inc.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2023. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70.

Lean. 2023. The lean project. https://lean-lang.org/.

Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 6227–6240. Association for Computational Linguistics.

OpenAI. 2023a. Gpt-3.5 turbo fine-tuning and api updates. https://openai.com/blog/gpt-3-5-turbo-fine-tuning-and-api-updates.

OpenAI. 2023b. GPT-4 technical report. *CoRR*, abs/2303.08774.

Jean-Pierre Queille and Joseph Sifakis. 1982. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer.

Stephen E. Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark W. Barrett. 2023. Clover: Closed-loop verifiable code generation. *CoRR*, abs/2310.17807.

Verus. 2023. The verus project. https://github.com/verus-lang/verus.

Makarius Wenzel. 2023. The isabelle system manual.

Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models. *CoRR*, abs/2306.15626.

Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. 2023. Leveraging large language models for automated proof synthesis in rust. *arXiv preprint arXiv:2311.03739*.

Chuanyang Zheng, Haiming Wang, Enze Xie, Zhengying Liu, Jiankai Sun, Huajian Xin, Jianhao Shen, Zhenguo Li, and Yu Li. 2023. Lyra: Orchestrating dual correction in automated theorem proving. *CoRR*, abs/2309.15806.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 2022. minif2f: a cross-system benchmark for formal olympiad-level mathematics. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

A Procedural *Vs*. Declarative Style

The procedural style proofs specify a series of tactics to apply, without describing the intermediate results. A demonstrative lemma from seL4 is shown below.

```
1  lemma unbind_notification_valid_sched[wp]:
2  "{valid_sched} unbind_notification ntfnptr
3    {λrv. valid_sched}"
4  apply (simp add: unbind_notification_def)
5  apply (rule hoare_seq_ext[OF _ gbn_sp])
6  apply (case_tac ntfnptra, simp, wp, simp)
7  apply (clarsimp)
8  apply (rule hoare_seq_ext[OF _ get_simple_ko_sp])
9  apply (wp set_bound_notification_valid_sched, clarsimp)
10 done
```

In the example, line 4-10 apply a sequence of tactics to achieve the proving goal. Declarative style proofs, on the other hand, explicitly write both the intermediate proving goals and the proving operations. A typical example from seL4 is shown below.

```
1  lemma thread_set_as_user:
2  "thread_set (λtcb. tcb ( tcb_arch := arch_tcb_context_set
3    (f $ arch_tcb_context_get (tcb_arch tcb)) (tcb_arch tcb) )) t
4    = as_user t (modify f)"
5  proof -
6    have P: "∧f. det (modify f)"
7    by (simp add: modify_def)
8    thus ?thesis
9    apply (simp add: as_user_def P thread_set_def)
10   apply (clarsimp simp add: select_f_def simpler_modify_def
11     bind_def image_def)
12   done
13 qed
```

Line 6 in this lemma specifies the intermediate proving goal, and the following lines performs a series of tactics.

In general, mathematical problems are usually pure and abstract, and therefore they are well-suited for the declarative style; while software verification usually deals with large, concrete and complex systems like seL4, and it benefits from the procedural style (Harrison, 1996). In Selene, we notice that most proofs in seL4 are in the procedural style.

	P1	P2	P3	D
Checking	148.9	145.8	217.3	178.7
GPT-3.5-turbo	40.2	43.7	42.5	50.6
GPT-4	35.6	43.5	43.9	43.3

Table 7: Average elapsed time of verification of correctness checking before evaluation, and ACC#1 evaluation of GPT-3.5-turbo and GPT-4 without augmentations (values in seconds).

B ROOT File

As previously introduced, Isabelle organizes the environment with sessions, which can be regarded as a container for verification results of certain topics. The structure of a session, including its dependent sessions and its entries, *etc*., is defined and described in the ROOT file. An example from seL4 is presented below, which is a part of a large ROOT file.

```
1 session BaseRefine in "refine/base" = AInvs +
2   description \<open>Background theory and
3     libraries for refinement proof.\<close>
4   sessions
5     Lib
6     CorresK
7   theories
8     "Include"
```

This partial ROOT file defines a session named "BaseRefine", which locates at the directory "refine/base" (Line 1). BaseRefine is directly dependent on another session "AInvs" (Line 1), and it imports two more sessions, "Lib" and "CorresK" (Line 4-6, these two sessions are also dependency to BaseRefine). BaseRefine has only one entry theory file, "Include.thy" (Line 7-8). The theory Include is dependent on other theories in BaseRefine, and the prover verifies the whole session in a bottom-up manner (it first checks all dependencies of Include, and then verifies lemmas within Include).

As in Figure 3, during lemma isolation, Selene sets the theory "Untyped_AI_DEP.thy" as the entry of the dependency session (Ainvs_DEP in the figure), and sets the dependency of the target session (Ainvs_TGT) to the dependency session (Ainvs_DEP).

C Verification Time

The time cost of the verification process is listed in Table 7. Correctness checking bears resemblance of building from scratch, and it takes on average about three times longer than verifying only the isolated target session. Note that we even include the ten minutes of timeout during evaluation in Table 7. Since we only perform correctness checking once before evaluation, lemma isolation can greatly improve the verification efficiency during evaluation of Selene.

D Prompt

Instruction. The basic instruction is shown below.

> You are an experienced formal language programmer. You not only know the Isabelle formal language very well, but also are very familiar with the seL4 project. As a reminder, seL4 is an almost fully formally verified operating system microkernel. Your mission is to write formal proofs in Isabelle for the given specifications, which formally describe properties of seL4 in Isabelle. You are not supposed to write anything other than formal proofs in Isabelle. *E.g.*, You should not write comments or explanations in natural language. In addition, the formal proofs you write will be automatically checked, therefore, you need to do your best to make it correct.

For each augmentation, there is an augmented instruction listed below. we concatenate the basic instruction and the corresponding augmented instruction, forming the final instruction.

> **Similar:** Some chunks of seL4 with similar specifications are provided before the target specification. Each chunk is provided between the tags of "<sim>" and "</sim>". You can use these chunks to assist the proof of the target specification.
> **Dependency:** Some previous chunks of seL4 are provided before the target specification as plausible dependencies. Each chunk is provided between the tags of "<dep>" and "</dep>". You can use these chunks to assist the proof of the target specification.
> **Fixing:** If the previous proof is not correct, the error message may be provided inside curly brackets {just like this}. If the error message is provided, you are supposed to make the previous proof correct at your best.

Demonstration. In general, a demonstration for the subject LLM (*e.g.*, GPT-4) is an input-output pair. In the most simple evaluation setting of Selene (without any augmentation), the input in the demonstration is the specification of the demonstrative lemma and the output is the corresponding proof. When augmented by similar chunks, the demonstration output remains the same, while the input is a concatenation of the similar chunk and the specification. The similar chunk is tagged by "<sim>" and "</sim>". The dependency augmentation is quite similar, with different tags of "<dep>" and "</dep>".

As for fixing augmentation, we need to collect erroneous proofs and corresponding error messages upon the demonstrative lemmas. For each lemmas within the five demonstrations, we invoke the subject LLM in the same manner as in Section 4, using the other four lemmas as demonstrations. In this way, we may collect some errors produced by the subject LLM. The demonstrations for fixing augmentation are much more complicated. We carry out this augmentation in a two-round dialogue way. *I.e.*, each demonstration may contain two input-output pairs (if the subject LLM fails). The first pair is the specification and the erroneous proof generated by the subject LLM; the second pair is the error message and the groundtruth proof. Note that the error message is inside curly brackets (*e.g.*, {Undefined fact: “st_def” At command “by”}). If the subject LLM successfully generates a proof for the demonstrative lemma, *i.e.*, it does not make any error, the demonstration only contains one pair, *i.e.*, the specification along with the groundtruth proof.

E Generated Cases

Some cases generated by GPT-3.5-turbo and GPT-4 upon the P1 category of Selene are presented in Table 8. These cases are randomly selected from the union of correct generation from GPT-3.5-turbo and GPT-4 under the setting of ACC#1, without cherry-picking. In the table, we can observe that GPT-4 almost do not generate a proof identical to the groundtruth (except for tbc_context_merge). This shows that to a certain extent, GPT-4 can understand and generate proofs for seL4.

Target Lemma	Generated Proof	
(Specification + Groundtruth Proof)	GPT-3.5-turbo	GPT-4
lemma (in strengthen_implementation) strengthen_Not[strg]: "[| st (¬ F) (⟶) P P' |] ⟶ st F (⟹) (¬ P) (¬ P')" (* PROOF STARTS RIGHT HERE *) by (cases F, auto)	by (erule strengthen_precondition) [×] Undefined fact	by (clarsimp simp: st_def) [√]
lemma gen_invocation_typeI: "invocation_type l = GenInvocationLabel x ⟹ gen_invocation_type l = x" (* PROOF STARTS RIGHT HERE *) by (simp add: gen_invocation_type_def)	by (simp add: gen_invocation_type_def) [√]	by (simp add: gen_invocation_type_def split: invocation_label.splits) [√]
lemma ts_Restart_case_helper: "(case ts of Structures_A.Restart ⇒ A | _ ⇒ B) = (if ts = Structures_A.Restart then A else B)" (* PROOF STARTS RIGHT HERE *) by (case_tac ts, simp_all)	by (case_tac ts, simp_all) [√]	by (cases ts; simp) [√]
lemma irq_state_back: "P (irq_state_of_state (internal_state_if ((a,b),c))) (irq_masks_of_state (internal_state_if ((a,b),c))) ⟹ P (irq_state_of_state b) (irq_masks_of_state b)" (* PROOF STARTS RIGHT HERE *) by simp	by simp [√]	by (clarsimp simp: internal_state_if_def) [×] Undefined fact
lemma cte_wp_at_conj: "cte_wp_at (λc. P c ∧ Q c) p s = (cte_wp_at P p s ∧ cte_wp_at Q p s)" (* PROOF STARTS RIGHT HERE *) by (fastforce simp: cte_wp_at_def)	by (auto simp: cte_wp_at_caps_of_state) [√]	by (auto simp: cte_wp_at_caps_of_state) [√]
lemma smrs_cte_at[wp]: "{cte_at p} set_mrs thread buf msgs {λ_. cte_at p}" (* PROOF STARTS RIGHT HERE *) by (wp valid_cte_at_typ)	Sorry, I cannot assist with this request. [×] Outer syntax error	by (wp | simp add: set_mrs_def)+ [√]
lemma master_eqE: "[| capMasterCap cap = capMasterCap cap'; ⋀cap. F (capMasterCap cap) = F cap |] ⟹ F cap = F cap'" (* PROOF STARTS RIGHT HERE *) by (rule master_eqI, assumption, simp)	by (metis) [√]	by (drule meta_mp) auto [×] Failed to apply proof method
lemma tcb_context_merge[simp]: "arch_tcb_context_get (tcb_arch (tcb_registers_caps_merge tcb tcb')) = arch_tcb_context_get (tcb_arch tcb)" (* PROOF STARTS RIGHT HERE *) by (simp add: tcb_registers_caps_merge_def)	by (simp add: tcb_registers_caps_merge_def) [√]	by (simp add: tcb_registers_caps_merge_def) [√]
lemma cte_map_tcb_2: "cte_map (t, tcb_cnode_index 2) = t + 2*2^cte_level_bits" (* PROOF STARTS RIGHT HERE *) by (simp add: cte_map_def tcb_cnode_index_def to_bl_1)	by (simp add: cte_map_def tcb_cnode_index_def) [√]	by (simp add: cte_map_def tcb_cnode_index_def cte_level_bits_def) [√]
lemma object_type_asid_reset [simp]: "object_type (asid_reset obj) = object_type obj" (* PROOF STARTS RIGHT HERE *) by (clarsimp simp: asid_reset_def)	by (cases obj, auto) [×] Failed to finish proof	by (cases obj; simp add: asid_reset_def) [√]

Table 8: Cases generated by the subject LLMs upon P1 of Selene. These cases are collected under the ACC#1 setting. The verification results are marked under the generated proofs. Please note that we have added some line breaks into the original lemmas and the generated proofs for the convenience of display in the table.