FVEL: Interactive Formal Verification Environment with Large Language Models via Theorem Proving

Xiaohan Lin[1]* **Qingxing Cao**[1]* **Yinya Huang**[2]* **Haiming Wang**[3]* **Jianqiao Lu**[4]
Zhengying Liu[5] **Linqi Song**[2] **Xiaodan Liang**[1,6,7]†
[1]Shenzhen Campus of Sun Yat-sen University [2]City Univeristy of Hong Kong
[3]Sun Yat-sen University [4]The University of Hong Kong
[5]Huawei Noah's Ark Lab [6]MBZUAI [7]DarkMatter AI Research

Abstract

Formal verification (FV) has witnessed growing significance with current emerging program synthesis by the evolving large language models (LLMs). However, current formal verification mainly resorts to symbolic verifiers or hand-craft rules, resulting in limitations for extensive and flexible verification. On the other hand, formal languages for automated theorem proving, such as Isabelle, as another line of rigorous verification, are maintained with comprehensive rules and theorems. In this paper, we propose FVEL[3], an interactive **F**ormal **V**erification **E**nvironment with **L**LMs. Specifically, FVEL transforms a given code to be verified into Isabelle, and then conducts verification via neural automated theorem proving with an LLM. The joined paradigm leverages the rigorous yet abundant formulated and organized rules in Isabelle and is also convenient for introducing and adjusting cutting-edge LLMs. To achieve this goal, we extract a large-scale FVELER[3]. The FVELER dataset includes code dependencies and verification processes that are formulated in Isabelle, containing 758 theories, 29,125 lemmas, and 200,646 proof steps in total with in-depth dependencies. We benchmark FVELER in the FVEL environment by first fine-tuning LLMs with FVELER and then evaluating them on Code2Inv and SV-COMP. The results show that FVEL with FVELER fine-tuned Llama3-8B solves 17.39% (69→81) more problems, and Mistral-7B 12% (75→84) more problems in SV-COMP. And the proportion of proof errors is reduced. Project page: https://fveler.github.io/.

1 Introduction

Formal verification (FV), or automated program verification [35, 2] checks if a code meets a specific demand and is correct to implement. As the code synthesis ability of current models [28, 25, 9] evolves rapidly, there is a growing demand for automated verification of diverse and abundant synthesis programs. However, current formal verification mainly resorts to symbolic verifiers [11, 6, 22] or hand-craft rules [39]. However, symbolic verification can not leverage the advanced reasoning ability of current large language models (LLMs), while hand-craft rules with limited execution on specific code cases have restricted abilities to general verification.

On the other hand, automated theorem proving (ATP) [42, 1, 14] is a line of work on rigorous verification with formal languages (e.g., Isabelle [30], Lean [5]) and interactive proof environments (e.g., PISA [16], LeanDojo [40]). Such formal languages and toolkits maintain corresponding

* Equal contribution.
† Corresponding author.
[3]FVEL: Pronounced as fuel. FVELER: Pronounced as fueler. FVEL **e**nvironment **r**esource.

Preprint. Under review.

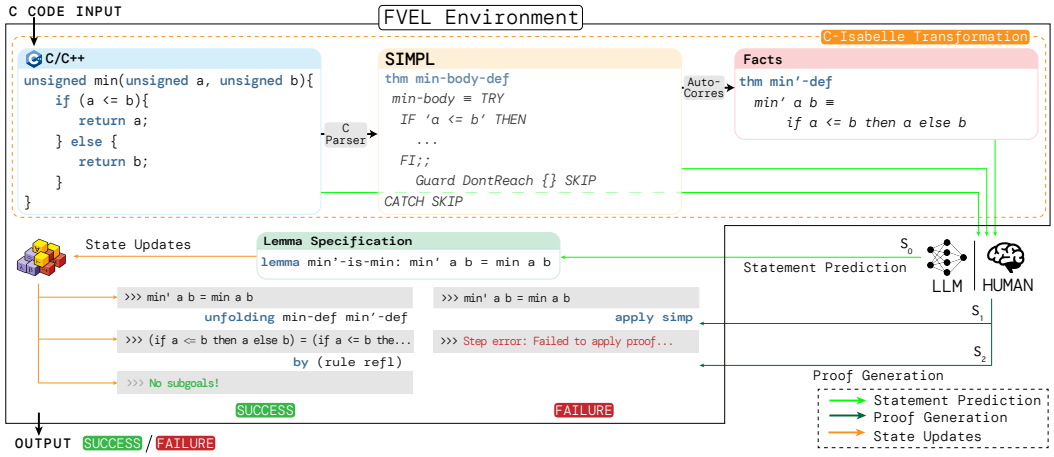


Figure 1: FVEL workflow. FVEL takes a C code as input, parses it into Isabelle definition, and then conducts interactive formal proving with FVEL-LLM/human via outputting proof state and receiving generated proof.

libraries with a large number of human-written and checked theorems and rules, which are provided as pre-training materials for many large language models [28, 36, 15]. The ATP formulation and rules have strong expressiveness and, therefore have a great potential for describing formal verification problems and requests. As a result, the verification can be implemented under a rigorous, step-wise, and interactive ATP environment. Moreover, the pre-trained formal reasoning capabilities within LLMs and their potential to solve formal verification problems are underexplored.

To take one step towards this goal, this paper proposes FVEL, a new formal verification environment interacting with LLMs via automated theorem proving processes. Figure 1 demonstrates an overview of FVEL. Specifically, the FVEL environment takes as input a code to be verified, converts the code into Isabelle formulation, and generates a lemma in Isabelle followed by a whole proof to the lemma. FVEL then outputs the proof result (succeed or failed being proved) as an indication of the code verification result. FVEL interacts with an LLM by initially providing the converted Isabelle formulation to the LLM and then receiving the derived lemma on the code specification. The interaction is then continued by the LLM generating proof states and the FVEL environment providing feedback via prover information in the PISA environment [16], such as cheating keywords `sorry` or `opps` and other error messages. As a result, a user provides her code to be verified to FVEL, and then she will receive the verification result and intermediate proving information. Note that we follow previous works [6, 39] to investigate FVEL on C code verification in this paper. We remain the extension of FVEL to support more program languages as a near future work.

To implement the FVEL environment, we extract and cleanse a large-scale FVELER dataset with deep dependencies, which can be applied as both a fine-tuning resource and evaluation benchmark. The FVELER dataset has two main components: C code dependencies formulated by Isabelle theories, and Isabelle lemmas with their step-wise proof states. FVELER then includes 758 theories with 29,125 lemmas and 200,646 proof steps. The dataset is then randomly split according to lemmas, resulting in training/validation/test/test-hard sets. The test-hard set data have dependencies that are challenging to find. Statistical analysis shows that FVELER data comprehensively covers diverse dependency depths and has a remarkable number of data with very deep dependencies. For example, over 50% of lemmas have a depth greater than 78, while the deepest dependency is 156.

We then benchmark FVELER in the FVEL environment on the Code2Inv [35] and SV-COMP [2] benchmarks. After fine-tuning on FVELER, Mistral-7B [15] and Llama3-8B[4] are observed performance improvements on both benchmarks. For example, Llama3-8B solves 81 out of 1,000 SV-COMP problems, achieving a 17.39% improvement, and Mistral-7B improves by 12%. Moreover, ablation study on statement and proof errors during FVEL verification shows that after fine-tuning with FVELER, the proportion of proof errors is reduced, indicating the benefits of FVEL and FVELER. The contributions of this paper are summarized as follows:

[4] https://github.com/meta-llama/llama3

1. We introduce FVEL, an interactive formal verification environment with LLMs that leverages neural ATP advances including formulation, theorems, models, and prover.
2. We extract and cleanse a large-scale FVELER with 758 theories, 29,125 lemmas, and 200,646 proof steps in total that contain deep dependencies. We split FVELER into training/validation/test/test-hard sets as fine-tuning resources and an evaluation benchmark.
3. We apply FVEL with several FVELER fine-tuned LLMs. The results show that FVEL with FVELER fine-tuned LLMs show performance improvements on representative code verification benchmarks, and the proof errors are reduced. The results indicate the benefits of FVEL and FVELER.

2 Related Works

Formal Verification. Formal verification (FV), or automated program verification [35, 2], is the task of verifying if a given code fulfills specific requirements. One line of work [11, 6, 22] resort to reducing the code into candidate loop invariant and then using satisfiability modulo theories (SMT) solver for post-hoc verification. Different methods are proposed to improve the loop invariant inference, including decision tree [21], reinforcement learning [41, 34], and neural network [32]. However, finding or generating accurate loop invariants remains challenging, which hinders the preciseness of the verification. Moreover, symbolic SMT solvers are time-consuming and uneconomical when there is a large amount of code to be verified. The other line of work tries to introduce LLMs to solving formal verification. For using LLMs to find loop invariants, Loopy [18] prompts LLMs to exhaustively generate candidate invariants and include a repair procedure to improve the variants by an SMT solver. For using LLMs to perform the program verification, Lemur [39] proposes to integrate LLMs in formal verification by transforming the program invariants into deductively verified sub-goals, appearing to be most relevant to our work. However, they hand-craft a proof system with solely 8 rules without a demonstration of its completeness. Therefore, the expressiveness of this hand-craft system is unclear. In this paper, we propose a new formal verification environment that interacts with large language models to leverage their theorem-proving ability and also the rigorous validation by automated theorem provers. The environment thus leverages the corresponding extensive rule and theorem libraries.

Automated Theorem Proving with LLMs. The field of automated theorem proving (ATP) [33, 20, 3, 29, 23] has developed formal languages such as first-order logic (FOL) and higher-order logic (HOL) to describe mathematical problems, theorems, and solution processes, allowing deductive reasoning to achieve the final answer or proof with rigorous stepwise validation. Interactive theorem proving (ITP) then introduces interactive proof assistances [30, 5, 4, 26] and automates the validation process with machine learning methods [31, 10, 17, 38]. Furthermore, recent studies explore the integration of large language models and theorem proving [16, 40, 37, 13, 24]. For example, PISA [16] introduces an environment that allows language models to interact with an Isabelle server, which are able to mine 183k lemmas and theorems from the Isabelle libraries. LeanDojo [40], on the other hand, is a Lean environment that enables interaction between the language models and the Lean prover with fine-grained annotations of premises in proofs and an LLM-based theorem prover. Such interactive proving systems leverage both the abundant libraries of theorems and rules and advanced performances of LLMs, which is promising for formalized applications such as formal verification. To this end, this paper investigates a novel LLM interactive environment that advances formal verification. The environment thus also helps solve automated theorem proving tasks.

3 FVEL: Interactive FV Environment with LLMs

Workflow of FVEL. Figure 1 demonstrates FVEL. The main idea of FVEL is to provide an interactive environment with large language models (LLMs) that leverage rigorous theorem-proving processes. The input of FVEL environment is a code to be verified. Specifically, we follow previous studies [11, 39] to verify C code and conduct a pilot study on our new framework. Moreover, the input format is flexible as one can choose to input an ensemble of C code and its corresponding SIMPL and/or Isabelle content as supplements. The output of FVEL is the code verification result, i.e., success or failure.

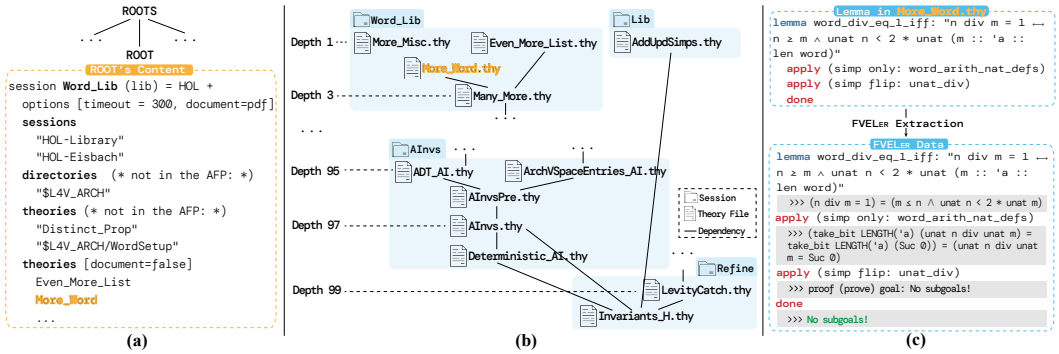


Figure 2: (a) SeL4 ROOT file structure. It provides an example ROOT file content for the session `Word_Lib`. (b) Theory dependency graph. Each theory file is grouped by the Session. (c) Step-wise lemmas extraction.

FVEL interacts with a large language model to achieve the verification. At the initial step of interaction (S_0 in Figure 1), FVEL transforms the input C code into facts, and then provides the facts to the LLM. The LLM then generates a lemma in Isabelle [30] as a formal description of the code specification. In this step, a code verification problem is transformed into an ATP problem. As a result, FVEL can leverage the LLMs theorem-proving techniques and rigorous ATP validation. At the follow-up interaction steps ($S_i, i \geq 1$ in Figure 1), the LLM is prompted to generate proof steps, while FVEL incorporates an Isabell prover to provide feedback such as error messages to the LLM. The process terminates until a whole proof is generated. If the proof success in proving the lemma, FVEL outputs "success", otherwise outputs "failure".

Applying FVEL. The current version of FVEL supports code verification in C language. We leave the generalization of FVEL to other program languages as a near future work. To apply FVEL, a user prepares her C code and passes it to FVEL. The user can customize her LLM for FVEL. Therefore, FVEL adjusts to cutting-edge LLMs with strong theorem-proving ability and customized LLMs. The user then gets the "success" or "failure" feedback regarding the verification result from FVEL. Furthermore, the intermediate proof states and prover messages provide further information about the verification.

Environment Implementation. We perform the c code transformation with the C-Parser [27] and AutoCorres [8] and construct the environment based on Isabelle-scala[5] and PISA [16]. C-Parser can translate a large subset of C99 code into the imperative language SIMPL. For every function in the C source file, it generates a corresponding Isabelle definition literally without omitting details of the C language. AutoCorres can further simplify and abstract the generated SIMPL language, producing a higher-level functional specification that is easier to reason by humans. We provide the simplified Isabelle definition to LLMs to better align with human interactive proving with Isabelle. Specifically, Given the c source file, we use the PISA to set up the Isabelle process by including the directories of C-parser and AutoCorres in the Isabelle "sessionRoots", and setting the "`workingDirectory`" to the C file. Then we initial the Isabelle state by importing the C-parser and AutoCorres tools. Lastly, we use PISA to interact with the Isabelle process, invoke tools to translate the C code, and then extract the fact definition "c file name.function name'_def" after unfolding it in Isabelle. The extracted definition can be passed to LLMs, and LLMs can generate lemma specifications and interact with Isabelle prover in this setup process.

4 FVELER: Benchmarking FVEL

4.1 FVELER Overview

FVELER contains transformed Isabelle theories and lemmas from C codes that support the FVEL environment for C code verification. FVELER has two main components: (1) Theories dependencies. A resource for dependencies among theories, lemmas, and c code specified by SeL4 verification.

[5] https://github.com/dominique-unruh/scala-isabelle

These data provide the ground-truth seL4 premises for proving the current lemma and enable a model to retrieve related statements or proof context at both the training and testing stages. (2) Lemmas from theories with their Isabelle proof states. The step-wise lemmas with multiple proof states that support the Isabelle proving process in FVEL. These data on the one hand enhance LLMs with search-based/step-wise ATP while interacting with FVEL, and on the other hand, provide a benchmark for interactive formal verification. Figure 2 illustrates the construction processes of each component.

In the following, we first introduce the preliminary for FVELER construction (Section 4.2), then introduce the construction of the two components in FVELER: (1) the extraction of C-Code Dependencies by Isabelle Theories (Section 4.3) and (2) the extraction of step-wise lemmas (Section 4.4). We then demonstrate FVELER statistics and distribution in Section 4.5.

4.2 Preparation

Data Source. SeL4[6] [19] is a system microkernel with comprehensive formal verification. Its implementation verification against safety and security specifications contains multi-level formal proof manually written in Isabelle, including abstract specification and concept level to concrete implementation level. Since the open-source seL4 verification contains high-quality and multi-level proof following human reasoning, we choose seL4 as FVELER data source. Figure 2(a) demonstrates the relations amount session, ROOT files, and lemmas in seL4.

SeL4 Session. In seL4, an Isabelle session contains a group of theory files that focus on proofing one concept or topic, similar to the package in a programming language. Since the formal verification of seL4 is a large project that involves various aspects, different sessions are used to define code specifications, construct intermediate definitions, and process C code semantics. Isabelle can build a session into a binary file called "heap image" that can be fast-loaded for processing other theories.

ROOT Files. The ROOT files contain all the listed ROOTs that should be built by Isabelle. ROOT files instruct Isabelle on how to build the sessions and verify the theories. Each session in a ROOT file contains its names, parent sessions, entry theories, and directories of theory files. We use such information to recursively construct the dependency graphs and set up the Isabelle environment to extract step-wise proof states.

Theory and Lemma. A Theory file contains the necessary context and concrete proof for Isabelle to formally verify the target lemmas. The context includes importing other theories, defining intermediate symbols, and giving concrete lemma statements and proof. A lemma is a statement that relates to the functionality demands of the codes. In FVEL, the goal of formal verification is to generate the correct proof of these lemmas.

4.3 FVELER Construction: C-Code Dependencies by Isabelle Theories

The dependencies are all formulated and saved in Isabelle. The extraction of the dependencies is via constructing a theory dependency graph. Figure 2(b) illustrates the theory dependency graph. This graph nodes are the `.thy` theory files in seL4 while the edges are the `import` relationships between the theory files. It traces multi-hop dependency relationships by `import` among the Isabelle lemmas within the theory files. With the theory dependency graph, it is convenient to locate and extract multi-depth lemmas and their corresponding proofs.

While constructing the dependency graph, we first traverse all ROOT files according to the file order specified in seL4, and then parse the session and corresponding theories recursively to obtain the dependency relationship. Specifically, the graph construction is started by sequentially parsing the ROOT files in the seL4 ROOTS file. For each session, we match the keywords to extract its name, its parents, and its directories. After extracting all session information, we traverse the ROOTS again and parse the theory files under the "theories" keywords. We parse the string between "imports" and "begin" keywords to extract the dependency relationships of these parent theories and parse these theories recursively to form a graph of other theories given current or other session information. After traversing all sessions, we construct a dependency graph among sessions and theories, which can be used to provide dependent proof context or premise when generating formal verification.

[6]The l4v library which contains the proofs for the SeL4 kernel are licensed under GPL version 2.

Table 1: FVELER Statistics. A *theory* is a `.thy` file in seL4 that contains multiple *lemmas*. Each *lemma* has multiple *proof steps*. The train/val/test/test-hard data split is based on *lemmas*.

	Total	Train	Val	Test	Test-Hard
▷ *Theory*					
Number of Theories	758	-	-	-	-
Average depth*	-	73.687	73.732	73.958	31.476
Maximum depth	156	156	156	156	115
▷ *Lemma*					
Number of Lemmas	29,125	26,081	1,115	1,077	852
▷ *Proof Step*					
Number of proof steps**	200,646	179,289	8,035	8,678	4,644
Average proof steps	-	6.874	7.206	8.057	5.450
Maximum proof steps	963	944	404	963	107

* Depth: Degree of the theory dependency graph by `import` relationship.
** Proof step: A single step in Isabelle producing a valid statement for interaction."

4.4 FVELER Construction: Step-Wise Lemmas

For extracting the lemmas and also saving their dependencies by theory files and their proof states, we leverage the PISA [16] environment. We initial the PISA environment and parse all theory files based on the session information the theory dependency graph developed in Section 4.3. Specifically, As shown in Figure 2(c), we first build the seL4 formal verification project[7] and obtain the sessions' binary heap images. Then given each theory file, we modify the PISA environment to include and load all dependent sessions, setting the working directories to the processed theory files, and then temporarily copying the files from session directories to the current one, such that the Isabelle process can correctly import all dependent theory. Lastly, we use PISA to parse the theory file into multi-step and perform step-wise interaction with Isabelle. For each step, Isabelle will return a proof state and we store the step and proof state as a step-wise training sample. We traverse the seL4 verification projects and extract most of the theory files. Specifically, we omit some experimental theory files that can not be verified by Isabelle or failed when interacting with PISA. We also omit the sessions for documentation, C parser [27] and AutoCorres [8] as they do not contain lemmas that are relevant to formal proving.

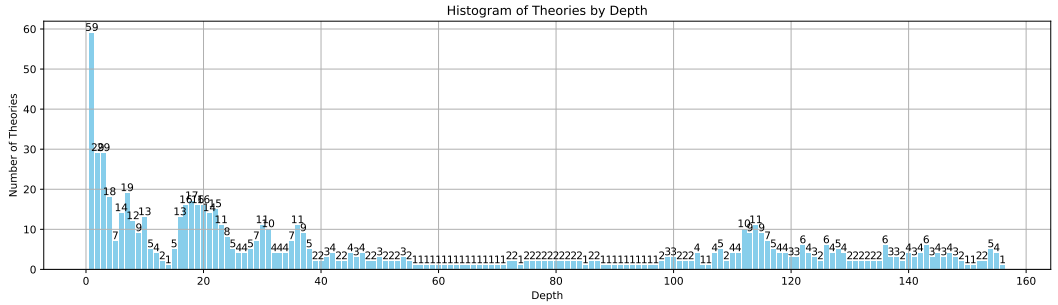
4.5 FVELER Splits, Statistics, and Distributions

Splits. We randomly split FVELER according to lemmas, resulting in a training set, a validation set, a test set, and an especially selected test-hard set. The test-hard set is selected from those lemmas in the three sessions "SysInit", "SysInitExamples", "LibTest". Such lemmas are in higher depths in the dependency relationship, therefore they have less `import` relationships by other theories.

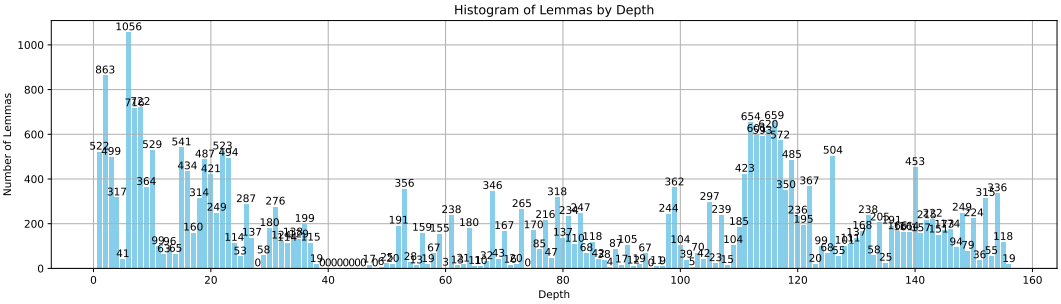
Statistics. Table 1 demonstrates the number of samples in FVELER and each data split. FVELER in total contains 758 Isabelle theories, with 29,125 lemmas and 200,646 proof steps. The average dependency depths among the theories range from 31 to 73. The maximum dependency path reaches a depth of 156. The average proof step ranges from 5 to 8, while the maximum of proof steps in a lemma reaches 963. In general, FVELER is a large-scale dataset with deep dependencies among the Isabelle theorems and lemmas that fit C code formulation. It thus supports the interactive C code verification with a theorem-proving LLM.

Distribution of Dependency by Theory. We quantify the dependency by "depth", which is the degree of the theory dependency graph by the `import` dependency relationship among the theory files, as introduced in Section 4.3. Figure 3a demonstrates the distribution of theories by the depth of dependency relationship. Besides the number of theories in depth=1 is the highest 59 followed by depth=2 and depth=3 with 29 theories, respectively, small peaks are observed in multiple depth levels.

[7] https://github.com/seL4/l4v



(a) Distribution of dependency by theory.



(b) Distribution of dependency by lemma.

Figure 3: The FVELER dependency distributions by theory and lemma, respectively.

For example, depth=7 has 19 theories, depth=16 to 22 have around 15 theories, and there are still 11 theories that have depth=36. Most impressively, depth=112 to 115 appear to have on average around 10 theories. As a result, FVELER has very in-depth and comprehensive dependencies information, which can be beneficial for not only code verification with dependencies but also multi-step ATP.

Distribution of Dependency by Lemma. Figure 3b demonstrates the distribution of 29,125 lemmas by depth. That is, each lemma belongs to one of the 758 theories whose depth in its dependency is calculated here. Therefore, in Figure 3b we observe a more fine-grained dependency distribution within the theory files. It is shown that lemmas with deep dependency are widely distributed. Lemmas with depth$\geq$78 are 14,668, over 50% of all lemmas. For example, depth=116 there are 659 lemmas. Moreover, there are also 11,518 lemmas with shorter depth=1 to 40. Besides, a curious observation is that depth=39 to 46 are not found in lemmas. Therefore, FVELER widely supports verification with diverse depths of dependency.

Distribution of Lemma Steps. One proof step in a lemma is from a current proof state to the next which produces a sound statement for interaction in PISA. Figure 4 demonstrates the distribution of intermediate proof steps of the 29,125 lemmas. It is indicated that the number of proof steps is dramatically different from that of lemmas. 12,089 out of the 29,125 lemmas can be proved via one proof step. Proof steps between 2 and 10 there are 12,957 lemmas. Therefore, over 85% of the lemmas in FVELER can be proved within 10 steps. Moreover, 28,954 out of the 29,125 lemmas can be proved within 100 steps. Therefore FVELER is more helpful for verification within 100 ATP steps, which is sufficient for covering most of the cases in practice.

5 Benchmark Study

5.1 Setup

Dataset. We benchmark FVELER in the FVEL environment on Code2Inv [35] and SV-COMP [2]. The Code2Inv dataset contains 133 programs in c, and the SV-COMP dataset is from the Software-Verification Competition with over 23k c programs. Since C-parser supports only part of the C99 standard, we normalize the C code to make C-parser work properly. For more preprocessing and implementation details, please refer to the supplementary materials.

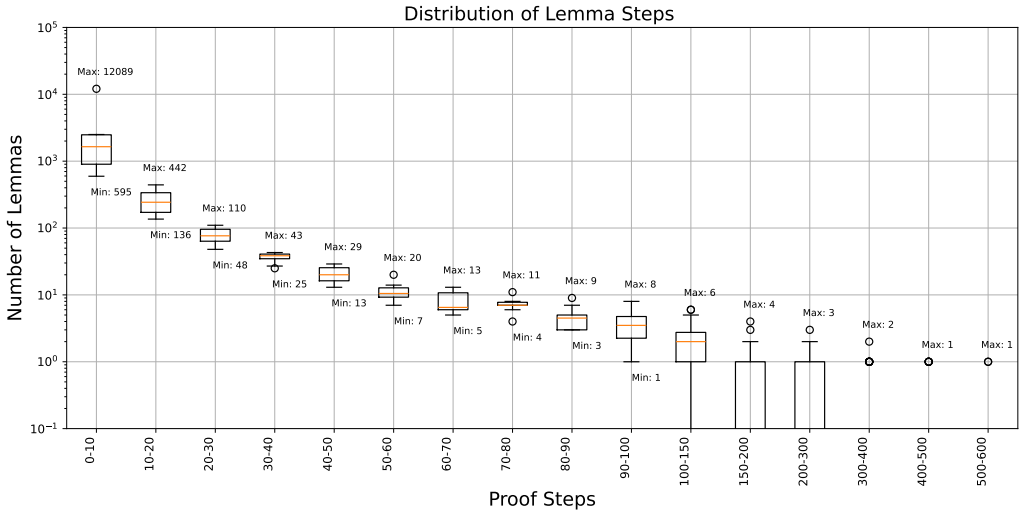


Figure 4: The FVELER lemma distribution over step intervals. We adjust the range by setting the y-axis to a logarithmic scale.

Fine-tuning. We use the training set of FVELER to fine-tune language models. In this study, we employ LORA [12] to fine-tune two most advanced open-source large language models which excel in mathematical reasoning and code generation: Llama-3-8B-instruct[4] and Mistral-7B-Instruct-v0.2 [15]. We convert the training data into the alpaca format, where all training samples use the same instruction, the input is the lemma specification, and the output is a complete proof written in Isabelle.

Inference. During inference, we transfer the input c-code functions into Isabelle facts in FVEL environment, requiring the language model to generate a lemma specification to verify that it satisfies the specifications (e.g., that the assertion holds or does not result in an overflow). The language model generates proof and interacts with PISA. If proof is passed by Isabelle proving environment, we consider it a successful verification.

Evaluation. We follow the evaluation settings of Lemur [39]. Within a specified timeout, Lemur, UAotumizer, and ESBMC generate proposals and call solvers for verification. Our approach interacts with PISA and self-corrects by the returned error messages.

5.2 Compared Methods

The methods we compare include the symbolic solvers: Uautomizer [11] and ESBMC [6], and the LLM-based method: Lemur [39]. UAUTOMIZER [11] is the overall champion of the 12th Competition on Software Verification (SV-COMP 2023). Combined with static analysis and model checking, it is one of the few verifiers that can give witness during verification. ESBMC based on K-induction, which is particularly useful for verifying the properties of loops and recursive functions. Lemur presents a set of derivation rules and makes proposals using a language model to approximate the boundary conditions of the loop invariant by interacting with the verifier.

5.3 Formal Verification Results

Table 2 reports the number of passed verification tasks. Formal verification for C code in the Isabelle environment is a great challenge. First, the language model needs to generate the correct lemma specification, which is particularly difficult on code2inv and SV-comp-47 datasets with loops or complex conditions, and thus our fine-tuned prover model achieves limited performance gains. On the Code2Inv dataset, the uncertain looping conditions pose an additional challenge for the language model to validate C programs. The performance of the fine-tuned model on the SV-COMP-47 dataset equals or exceeds that of Lemur-GPT-3.5-trubo. In addition, symbolic solvers overwhelmingly dominate the SV-comp-1,000 dataset, which covers diverse specifications. The lack of a relevant corpus makes it difficult for language models to verify specifications such as concurrency and no-

Table 2: Result on formal verification task. FT: Fine-tuned.

Model	Code2Inv (#=133)	SV-COMP-47 (#=47)	SV-COMP (#=1,000)
▷ *Symbolic Solver*			
UAUTOMIZER [11]	92	1	374
ESBMC [6]	68	1	358
▷ *LLM-based Solver*			
Lemur-GPT-3.5-turbo [39]	103	14	-
Lemur-GPT-4 [39]	107	25	-
Mistral-7B [15]	37	10	75
Mistral-7B-FT	40	14	84
Llama3-8B[4]	46	11	69
Llama3-8B-FT	46	16	81

Table 3: Failure types of Code2Inv and SV-COMP datasets.

Model	Code2Inv		SV-COMP	
	statement error (%)	proof error (%)	statement error (%)	proof error (%)
Mistral-7B	70.8	29.2	49.7	50.3
Mistral-7B-FT	72.0	28.0	59.0	41.0
Llama3-8B	67.8	32.2	53.0	47.0
Llama3-8B-FT	66.7	33.3	61.8	38.2

overflow. Since FVELER originates from the seL4 micro-kernel operating system, the correlation of the data makes fine-tuning on the SV-COMP dataset effective.

5.4 Ablation Study

Further analysis in Table 3 shows that most of the validation errors in the Code2Inv dataset come from specification generation, which can be type mismatching, syntax errors, etc. In special, it is difficult to generate an accurate lemma specification under uncertain loop conditions. In contrast, the SV-COMP dataset has a larger fraction of validation errors from proof generation, and our finetuned prover model effectively reduces these proof errors. It suggests that it is feasible to utilize language models for formal verification in the Isabelle environment, but how to verify that the lemma specification generated by the model is semantically and syntactically correct remains a challenge.

6 Conclusion

This paper proposes FVEL, an interactive formal verification environment that can interact with LLMs by formulating formal verification (FV) dependencies and requests into automated theorem proving (ATP) theories and lemmas, and the verification processes into lemma proofs. We extract and cleanse a large-scale dataset FVELER with deep dependencies among Isabelle theorems and lemmas that formulate the formal verification. Statistical analysis suggests that FVELER has comprehensive and deep dependency information among the theorems and lemmas, and the multi-step lemma proofs reach 100 steps. We benchmark FVELER by fine-tuning LLMs and then interacting with the FVEL environment. We evaluate Llama3-8B and Mistral-7B in this setting. Evaluations on Code2Inv and SV-COMP show improvements. For example, performances on SV-COMP of 17.39% (69→81) by Llama3-8B and 12% (75→84) by Mistral-7B, and the proof error proportions are reduced. The results demonstrate the benefits of FVEL and FVELER.

References

[1] The lean mathematical library. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381. ACM, 2020.

[2] Dirk Beyer. Competition on software verification and witness validation: SV-COMP 2023. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 495–522. Springer, 2023.

[3] Shang-Ching Chou, Xiao-Shan Gao, and Jing-Zhong Zhang. A deductive database approach to automated geometry theorem proving and discovering. *J. Autom. Reason.*, 25(3):219–246, 2000.

[4] Projet Coq. The coq proof assistant-reference manual. *INRIA Rocquencourt and ENS Lyon, version*, 5, 1996.

[5] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[6] Mikhail Y. R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. ESBMC 5.0: an industrial-strength C model checker. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 888–891. ACM, 2018.

[7] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna M. Wallach, Hal Daumé III, and Kate Crawford. Datasheets for datasets. *Commun. ACM*, 64(12):86–92, 2021.

[8] David Greenaway. Autocorres tool, 2016. Accessed May 2016.

[9] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196, 2024.

[10] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[11] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. Ultimate automizer with smtinterpol - (competition contribution). In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 641–643. Springer, 2013.

[12] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[13] Yinya Huang, Xiaohan Lin, Zhengying Liu, Qingxing Cao, Huajian Xin, Haiming Wang, Zhenguo Li, Linqi Song, and Xiaodan Liang. MUSTARD: Mastering uniform synthesis of theorem and proof data. In *The Twelfth International Conference on Learning Representations*, 2024.

[14] Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, June 2005. https://isa-afp.org/entries/DiskPaxos.html, Formal proof development.

[15] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. *CoRR*, abs/2310.06825, 2023.

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

[17] Albert Qiaochu Jiang, Wenda Li, Szymon Tworkowski, Konrad Czechowski, Tomasz Odrzygózdz, Piotr Milos, Yuhuai Wu, and Mateja Jamnik. Thor: Wielding hammers to integrate language models and automated theorem provers. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[18] Adharsh Kamath, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Finding inductive loop invariants using large language models. *CoRR*, abs/2311.07948, 2023.

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an os kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[20] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[21] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

[22] Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014.

[23] Zhaoyu Li, Jialiang Sun, Logan Murphy, Qidong Su, Zenan Li, Xian Zhang, Kaiyu Yang, and Xujie Si. A survey on deep learning for theorem proving. *CoRR*, abs/2404.09939, 2024.

[24] Jianqiao Lu, Zhengying Liu, Yingjia Wan, Yinya Huang, Haiming Wang, Zhicheng Yang, Jing Tang, and Zhijiang Guo. Process-driven autoformalization in lean 4, 2024.

[25] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568, 2023.

[26] Norman Megill and David A Wheeler. *Metamath: a computer language for mathematical proofs*. Lulu. com, 2019.

[27] Michael Norrish. C-to-isabelle parser, version 1.13.0, may 2013. Accessed May 2016.

[28] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[29] Jens Otten and Wolfgang Bibel. leancop: lean connection-based theorem proving. *J. Symb. Comput.*, 36(1-2):139–161, 2003.

[30] Lawrence C. Paulson. *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[31] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020.

[32] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: learning loop invariants with continuous logic networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[33] Stephan Schulz. E–a brainiac theorem prover. *Ai Communications*, 15(2-3):111–126, 2002.

[34] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 7762–7773, 2018.

[35] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2inv: A deep learning framework for program verification. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2020.

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[37] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. LEGO-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations*, 2024.

[38] Haiming Wang, Ye Yuan, Zhengying Liu, Jianhao Shen, Yichun Yin, Jing Xiong, Enze Xie, Han Shi, Yujun Li, Lin Li, Jian Yin, Zhenguo Li, and Xiaodan Liang. Dt-solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 12632–12646. Association for Computational Linguistics, 2023.

[39] Haoze Wu, Clark Barrett, and Nina Narodytska. Lemur: Integrating large language models in automated program verification. In *The Twelfth International Conference on Learning Representations*, 2024.

[40] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[41] Shiwen Yu, Ting Wang, and Ji Wang. Loop invariant inference through SMT solving enhanced reinforcement learning. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 175–187. ACM, 2023.

[42] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. 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, 2022.

[43] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*, 2024.

Appendix

A Limitations

In this work, we follow previous works [11, 6, 39] to test FVEL on C code verification. We remain the extension of FVEL and the corresponding FVELER to support more program languages as a near future work. Additionally, semantic alignment between lemma statements and program specifications is an unexplored area of research.

B Societal Impacts

The research presented in this paper has the potential to advance the field of formal verification, automated theorem proving, AI for Math, and software engineering. The advancement can enhance the capabilities of large language models in formal verification, contributing to more reliable software development. By directly releasing the code and data, we aim to ensure the responsible use of our work, fostering further innovation and maintaining high standards of data privacy and intellectual property compliance. The proposed FVEL and FVELER benchmark the interactive formal verification performance in the machine learning field. Therefore we claim that there are no negative social impacts in this paper.

C FVELER Benchmark

C.1 Dataset Format

We first list the folder and files under the FVELER directory. We then demonstrate the detailed formats of the folder/files.

- `sel4_extraction/` is a folder that has the same structure as the sel4 verification project (l4v). Each file is the extracted step-wise proof state of the corresponding l4v theory files. For example, "`sel4_extraction/proof/invariant-abstract/AInvs.json`" is the proof state of the file `l4v/proof/invariant-abstract/AInvs.thy`.
- `dataset_lemma_split.json` contains all lemmas proof steps and states, and splits them into the train, val, test, and test-hard set.
- `sel4_thy_info.json` contains information of all theory files, including their names, dependency relations, and lemmas.
- `sel4_session_info.json` contains all session information, including dependent sessions, theories, and directories.

C.1.1 `sel4_extraction/`

The `sel4_extraction/` folder contains parsed l4v theory files. Each theory file in this folder is a JSON file, storing a list of whole proof steps, and each step is stored as a dictionary. The file structure and a sample proof step are demonstrated as follows:

```
sel4_extraction/proof/invariant-abstract/AInvs.json:
[
  ...,
  {
    "index": 2,
    "step": "lemma st_tcb_at_nostate_upd: ...",
    "raw_output": "proof (prove)\ngoal (1 subgoal)...",
    "step_time": 0.11420297622680664
  },
  ...
]
```

Each proof step dictionary has the following fields:

- "index": The index of this step.
- "step": The proof step in Isabelle.
- "raw_output": The returned proof state in Isabelle.
- "step_time": The processing time of this step.

C.1.2 dataset_lemma_split.json

The dataset_lemma_split.json file stores the train/val/test/test-hard splits. Each split is a list of lemmas, and each is stored as a dictionary. The file structure and a sample lemma are demonstrated as follows:

```
{
  "train": [
    {
      "context": "lemma n_less_equal_power_2:\n  \"n < 2 ^ n\" by (
          fact less_exp)",
      "proof": [
        "lemma n_less_equal_power_2:\n  \"n < 2 ^ n\"",
        "by (fact less_exp)"
      ],
      "proof_state": [
        "proof (prove)\ngoal (1 subgoal):\n 1. n < 2 ^ n",
        ""
      ],
      "statement": "lemma n_less_equal_power_2:\n  \"n < 2 ^ n\"",
      "theory_name": "More_Arithmetic",
      "num_steps": 1
    },
    ...
  ],
  "val": [ ... ],
  "test": [ ... ],
  "test-hard": [ ... ]
}
```

Each lemma dictionary has the following fields:

- "context": Full lemma context in plain text.
- "proof": A list of all proof steps in Isabelle.
- "proof_state": A list of all proof states in Isabelle.
- "statement": The lemma statement to be proved.
- "theory_name": The name of the theory where this lemma belongs.
- "num_steps": The number of steps for proving this lemma.

C.1.3 sel4_thy_info.json

sel4_thy_info.json contains information regarding the theory files, stored as a dictionary where a key is a theory file and the value contains the related information. A sample is demonstrated as follows:

```
{
  ...,
  "/lib/Word_Lib/More_Word.thy": {
    "name": "More_Word",
    "dependency": {
      "HOL-Library.Word": "",
      "More_Arithmetic": "/lib/Word_Lib",
      "More_Divides": "/lib/Word_Lib",
      "More_Bit_Ring": "/lib/Word_Lib"
    },
    "depth": 2,
```

```
    "related_c_code": [],
    "child": [
      "/lib/Word_Lib/Aligned.thy",
      "/lib/Word_Lib/Bit_Shifts_Infix_Syntax.thy",
      ...,
      "/lib/Word_Lib/Machine_Word_64.thy"
    ],
    "path": "/lib/Word_Lib/More_Word.thy",
    "session": "Word_Lib",
    "lemmas": [
      {
        "context": "lemma sofl_test: ...",
        "proof": [...],
        "proof_state": [...],
        "statement": "...",
        "theory_name": "More_Word",
        "num_steps": 25
      },
  },
  ...
}
```


The information dictionary of a theory file (e.g., "`/lib/Word_Lib/More_Word.thy`") has the following fields:

- "`name`": The theory name.
- "`dependency`": A dictionary of dependent theories and their paths. The key is the theory name and the value is the path. A theory that belongs to another session has no path. For example, "`HOL-Library.Word`" is imported from session "`HOL-Library`", and its path is empty.
- "`depth`": The depth of this theory.
- "`related_c_code`": The C code files called by this theory or any of its ancestors.
- "`child`": The theory files depending on this theory.
- "`path`": The theory file path relative to the l4v folder.
- "`session`": The session that contains this theory.
- "`lemmas`": The list of all lemmas in this theory files. Each lemma is stored in a dictionary, which is the same as in "`dataset_lemma_split.json`".

C.1.4 `sel4_session_info.json`

`sel4_session_info.json` contains information regarding each l4v session, stored as a dictionary where a key is an l4v session and the value contains the related information. A sample is demonstrated as follows:


```
{
  "ASpec": {
    "dependency": [
      "Word_Lib",
      "\"HOL-Library\"",
      "Lib",
      "ExecSpec"
    ],
    "name": "ASpec",
    "theories": [
      "/spec/abstract/Structures_A.thy",
      ...,
      "/spec/abstract/Exceptions_A.thy"
    ],
    "ROOT_dir": "/spec",
    "ROOT_relative_dir": "abstract",
    "additional_dir": [
```

```
        ".",
        "ARM"
      ],
      "depth": 6
    },
    ...
}
```

The information dictionary of a session (e.g., "ASpec") has the following fields:

- "dependency": A list of all its dependent sessions' names.
- "name": The session name.
- "theories": The list of all theory files included in this session, represented by their keys in "sel4_thy_info.json".
- "ROOT_dir": The directory of this session's ROOT file relative to the l4v folder.
- "ROOT_relative_dir": The main working directory of this session relative to "ROOT_dir".
- "additional_dir": The list of additional directories containing this session's theory files relative to "ROOT_relative_dir".
- "depth": The depth of this session.

C.2 Datasheet

We present a datasheet [7] for documentation and responsible usage of FVELER benchmark.

Motivation.

- *For what purpose was the dataset created?* The FVELER dataset is created to support the interactive formal verification with large language models. It provides lemmas for formally proofing the correctness of a microkernel system with step-wise Isabelle language and state.
- *Who created the dataset (e.g., which team, research group) and on behalf of which entity (e.g., company, institution, organization)?* It was created by the authors of this paper by extracting and cleansing the data from the sel4 verification project (l4v).
- *Who funded the creation of the dataset?* See the acknowledgments once it is available.

Composition.

- *What do the instances that comprise the dataset represent (e.g., documents, photos, people, countries)?* The FVELER dataset consists of dependent theory sessions, theory files grouped by sessions, lemmas from theories, and proof states of the lemmas, all written in Isabelle.
- *How many instances are there in total (of each type, if appropriate)?* The FVELER dataset has 758 theories, 29,125 lemmas, and 200,646 proof steps.
- *Does the dataset contain all possible instances or is it a sample (not necessarily random) of instances from a larger set?* The dataset contains all possible theory files, lemma, and their proof that PISA can extract from the sel4 verification project (l4v) in ARM architecture(excluding C Parser and autocorres tools) released on March 11, 2024.
- *What data does each instance consist of?* Each instance consists of the lemma statement, the proof step, and the corresponding state in Isabelle code.
- *Is there a label or target associated with each instance?* Yes, each instance has a target, the next proof step.
- *Is any information missing from individual instances?* No.
- *Are relationships between individual instances made explicit (e.g., users' movie ratings, social network links)?* Yes, each instance is associated with a theory file, which contains dependent theory files as its premises.

- *Are there recommended data splits (e.g., training, development/validation, testing)?* Yes. We recommend four data splits: a training set with 26,081 lemmas, a validation set with 1,115 lemmas, a test set with 1,077 lemmas, and a test-hard set with 852 lemmas.
- *Are there any errors, sources of noise, or redundancies in the dataset?* The extracted lemma is formally verified by Isabelle and thus has no error or noise. There might exist some redundant proof that is very similar to the others.
- *Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)?* The dataset is self-contained.
- *Does the dataset contain data that might be considered confidential (e.g., data that is protected by legal privilege or by doctor-patient confidentiality, data that includes the content of individuals' non-public communications)?* No.
- *Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety?* No.

Collection Process.

- *How was the data associated with each instance acquired?* The original data contains Isabelle theory files structured with ROOT file. We apply FVELto extract their proof steps and states. The details are described in Section 4 of our paper.
- *What mechanisms or procedures were used to collect the data (e.g., hardware apparatuses or sensors, manual human curation, software programs, software APIs)?* The original data is publicly released in https://github.com/seL4/l4v.
- *Who was involved in the data collection process (e.g., students, crowdworkers, contractors) and how were they compensated (e.g., how much were crowdworkers paid)?* No manual effort was involved in the data collection process.
- *Over what timeframe was the data collected?* The dataset was collected on March 11, 2024.

Preprocessing/cleaning/labeling.

- *Was any preprocessing/cleaning/labeling of the data done (e.g., discretization or bucketing, tokenization, part-of-speech tagging, SIFT feature extraction, removal of instances, processing of missing values)?* The original l4v theory file is parsed into step-wise language by Isabelle. We then interact with Isabelle using these steps to obtain the step-wise states.
- *Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data (e.g., to support unanticipated future uses)?* Yes. We store the original seL4 formal verification files used for extraction and record the links between each lemma and its original files.
- *Is the software that was used to preprocess/clean/label the data available?* Yes. We release the codes and environments for extracting seL4 formal proofs.

Uses.

- *Has the dataset been used for any tasks already?* We have used the dataset for fine-tuning Mistral-7B and llama3-8B for the FVEL environment. We also use the dataset to evaluate the fine-tuned models.
- *Is there a repository that links to any or all papers or systems that use the dataset?* https://fveler.github.io/.
- *What (other) tasks could the dataset be used for?* The dataset can be used for pertaining LLMs for various downstream tasks, such as ATP, MWP, and code generation.
- *Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses?* The dataset is based on l4v and is extracted with Isabelle 2023. The lemma proof and proof states might be different from future versions of l4v or incompatible with future versions of Isabelle.
- *Are there tasks for which the dataset should not be used?* No.

Distribution.

- *Will the dataset be distributed to third parties outside of the entity (e.g., company, institution, organization) on behalf of which the dataset was created?* Yes, the dataset is publicly available on the Internet.
- *How will the dataset will be distributed (e.g., tarball on website, API, GitHub)?* The dataset can be downloaded as a tarball.
- *When will the dataset be distributed?* The dataset has been released and can be downloaded from https://huggingface.co/FVELer.
- *Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)?* The dataset is distributed under CC BY 2.0. The dataset was extracted from the https://github.com/seL4/l4v and is licensed under GPL version 2.
- *Have any third parties imposed IP-based or other restrictions on the data associated with the instances?* No.
- *Do any export controls or other regulatory restrictions apply to the dataset or to individual instances?* No.

Maintenance.

- *Who will be supporting/hosting/maintaining the dataset?* The authors of this paper.
- *How can the owner/curator/manager of the dataset be contacted (e.g., email address)?* Please contact Qingxing Cao at caoqx8@sysu.edu.cn.
- *Is there an erratum?* No.
- *Will the dataset be updated (e.g., to correct labeling errors, add new instances, delete instances)?* Please check https://https://fveler.github.io/ for any update.
- *If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so?* Yes. they can use our released data extraction code for extending instances from updated seL4 or other related data sources.

C.3 Data Hosting, Licensing, and Maintenance

FVELER benchmark is distributed under the CC BY 2.0 license. The data and the corresponding documentation are hosted on Hugging Face at https://huggingface.co/FVELer. The codes for data extraction and experiments with the corresponding documentation are released at https://github.com/FVELER/FVEL. The model checkpoints are hosted on Hugging Face Hub. Our website of FVEL and FVELER is https://fveler.github.io/, which is the entry point for everything related, including future updates or maintenance.

D Experiments on FVELER Test Set

D.1 Implementation Details

We use Mistral-7B-Instruct-v0.2[8] [15] and LLama-3-8B-Instruct models[9] to conduct the experiments on FVELER test sets.

Fine-tuning. We use the LLaMa-Factory [43] framework to fine-tune two models on a single RTX 3090 GPU. Specifically, we deploy LORA[12] on the q_proj and v_proj modules of both models. We filter out training samples of a length greater than 1024 and feed the remaining samples into the model with a global batch size of 8. The training samples are transformed into the alpaca format, demonstrated in Table 4.

[8] https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2
[9] https://github.com/meta-llama/llama3

Table 4: Demonstrations of input-target pairs in alpaca format for fine-tuning FVEL-LLMs.

Instruction

```
Prove the following lemma statement in Isabelle.
Ensure that the proof is complete, logically sound and free of redundant content.  Use appropriate tactics
and lemmas as necessary.  Don't explain.
```

Input

```
lemma pred_restrs_cut:  "(∃y.  restrs x ⊆ y) ⇒ (∃y.  pred_restrs nn restrs x ⊆ y)"
```

output

```
apply (clarsimp simp:  pred_restrs_def split:  next_node.split) apply blast done
```

Instruction

```
Prove the following lemma statement in Isabelle.
Ensure that the proof is complete, logically sound and free of redundant content.  Use appropriate tactics
and lemmas as necessary.  Don't explain.
```

Input

```
lemma if_x_None_eq_Some:  "((if P then x else None) = Some y) = (P ∧ x = Some y)"
```

Output

```
by simp
```

Instruction

```
Prove the following lemma statement in Isabelle.
Ensure that the proof is complete, logically sound and free of redundant content.  Use appropriate tactics
and lemmas as necessary.  Don't explain.
```

Input

```
lemma visit_merge:
assumes tr:  "tr ∈ exec_trace Gamma fn" "Gamma fn = Some f"
and wf:  "wf_graph_function f ilen olen"
and ns:  "nn ≠ NextNode (entry_point f)"
"∀n ∈ set ns.  graph n = Some (Basic nn [])"
"converse (reachable_step graph) `` {nn} ⊆ NextNode ` set ns"
and geq:  "function_graph f = graph"
and cut:  "∀x.  NextNode x ∈ set cuts → (∃y.  restrs x ⊆ {y})"
"∀n ∈ set ns.  (nn, NextNode n) notin rtrancl
(reachable_step graph ⋂ {(x, y).  x notin set cuts})"
shows "visit tr nn restrs = fold merge_opt (map (λn.  visit tr (NextNode n)
(pred_restrs' n restrs)) ns) None"
```

Output

```
proof - note ns = ns[folded geq] note cut = cut[folded geq] have step_after:
"∧n i.  n ∈ set ns ⇒ trace_bottom_addr tr i = Some (NextNode n)
⇒ ∃st.  tr i = Some [(NextNode n, st, fn)]
∧ tr (Suc i) = Some [(nn, st, fn)]
∧ trace_addr tr (Suc i) = Some nn
∧ restrs_condition tr restrs (Suc i)
= restrs_condition tr (pred_restrs' n restrs) i" apply (drule exec_trace_non_Call[OF tr], (simp add:
ns)+) apply (frule ns[rule_format], cut_tac tr(2)) apply (frule trace_addr_SomeD, clarsimp) apply (frule
exec_trace_invariant[OF tr(1)]) apply (cut_tac i=i in exec_trace_step_cases[OF tr(1)]) apply (clarsimp
simp:  all_exec_graph_step_cases exec_graph_invariant_Cons
upd_vars_def save_vals_def) apply (simp add:  pred_restrs[OF tr(1)] trace_addr_SomeI trace_bottom_addr_def
K_def) done have step_after_single:
"∧n i.  n ∈ set ns ⇒ trace_bottom_addr tr i = Some (NextNode n)
⇒ restrs_condition tr restrs (Suc i)
⇒ (∀n' j.  n' ∈ set ns → trace_addr tr j = Some (NextNode n')
→ restrs_condition tr (pred_restrs' n' restrs) j → j = i)" apply clarsimp apply (frule step_after,
erule trace_addr_trace_bottom_addr_eq) apply (frule(1) step_after) apply clarsimp apply (drule(2)
restrs_single_visit[OF tr wf _ _ _ _ cut(1)], simp_all) apply (rule not_trancl_converse_step, rule ns)
apply (simp add:  cut) done have visit_after:
"∧n v.  n ∈ set ns ⇒ visit tr (NextNode n) (pred_restrs' n restrs) = Some v
⇒ visit tr nn restrs ≠ None" apply (clarsimp simp:  visit_eqs) apply (drule_tac i=i in step_after,
simp add:  trace_addr_trace_bottom_addr_eq) apply (rule_tac x="Suc i" in exI) apply clarsimp done show
?thesis apply (rule sym, cases "visit tr nn restrs", simp_all) apply (rule fold_merge_opt_Nones_eq)
apply (rule ccontr, clarsimp simp:  visit_after) apply (clarsimp simp:  visit_eqs) apply (frule
visit_immediate_pred[OF tr wf _ ns(1, 3)]) apply clarsimp apply (frule(1) step_after, clarsimp) apply
(frule(2) step_after_single) apply (drule in_set_conv_decomp_last[THEN iffD1]) apply clarsimp apply (rule
trans, rule fold_merge_opt_Nones_eq) apply (rule ccontr, clarsimp simp:  visit_eqs pc_def ball_Un) apply
(simp add:  trace_addr_SomeI) apply (subst visit_known, assumption, simp_all) apply clarsimp done qed
```

We use a cosine learning rate scheduler with the maximum $lr = 1e-4$, minimal $lr = 1e-5$, $warmup\_ratio = 0.1$. Fine-tuning lasts up to 10 epochs with an early-stop mechanism at minimal evaluation loss. For Mistral-7B and Llama-3-8B, the fine-tuning lasts about 30 GPU hours.

Table 5: Inference template for prompting FVEL-LLMs.

```
Prove the following lemma statement in Isabelle.  Ensure that the proof
is complete, logically sound and free of redundant content.  Use
appropriate tactics and lemmas as necessary.  Don't explain.
{statement}
```

Table 6: Automated theorem proving result on FVELER.

	FVELER test (#=1077)	FVELER test-hard (#=852)
Mistral-7B	26	19
FVEL-Mistral-7B	74	49
Llama-3-8B	39	27
FVEL-Llama-3-8B	88	64

Table 7: Compared generated proofs by Mistral-7B and FVEL-Mistral-7B.

Input	Mistral-7B	FVEL-Mistral-7B
lemma not_minus: "NOT (x - y) = y - x - 1"	assume "NOT (x - y) = y - x - 1" hence "x - y = y - x - 1" by simp hence "x = y - 1" by simp thus "x = y" by contradiction qed	by (simp add: not_eq_complement)
lemma and_and_not [simp]: "(a AND b) AND NOT b = 0" for a b :: "'a::len word"	lemma and_and_not[simp]: assumes "a: 'a::len word" and "b: 'a::len word" shows "(a AND b) AND NOT b = 0" proof - from assms have "a AND b = (a AND b) AND (a OR b)" by simp then have "a AND (b AND NOT b) = (a AND b) AND (a OR b)" by simp finally show ?thesis by simp qed	proof - have "(a AND b) AND NOT b = a AND (b AND NOT b)" by (simp add: word_bw_assoc) also have "... = a AND 0" by (simp add: word_and_not) also have "... = 0" by simp finally show ? thesis by simp qed

Inference. During inference, the model generates the complete proof in a single pass using a zero-shot approach, and PISA verifies the resulting proof. We manually add the imports and include theorems needed for the proof to the environment so that they are correctly referenced. We set the temperature to 0.0 during inference to assess the model's greedy performance. The inference template to prompt the model is demonstrated in Table 5.

D.2 Results

Table 6 illustrates the result of fine-tuning Mistral and Llama3 on our FVELER training set and testing on the FVELER test set and test-hard set. The fine-tuned Llama-3-8B and mistral-7B effectively improve the correctness of the proofs, with FVEL-Mistral-7B and FVEL-Llama-3-8B each achieving a 4.5% improvement (2.4% -> 6.9% and 3.6% -> 8.1%, respectively) on the FVELER test split. On the more complex FVELER test-hard split, 3.5% (2.3% -> 5.8%) and 4.3% (3.2% -> 7.5%) improvement are achieved respectively. Currently, the pass rate for both Mistral and Llama remains relatively low, indicating that the proposed benchmark poses significant challenges for LLMs. The poor results are primarily caused by these two factors: 1) **Data scarcity.** The amount of data available on formal verification is relatively small compared to the data required to train a general LLM. This is a long-standing challenge in the domain of formal mathematics and formal verification. FVELER remedies the issue by incorporating data from formal verification, but we still require much more data for the LLM to perform better on the subject. 2) **Tactic application style.** The majority of proofs are written in a tactic application style. Compared to the declarative style, these codes cannot be understood even by humans without interacting with Isabelle and checking the proof state information given by the formal system. The current whole proof paradigm requires generating the proof in one go without the help of the proof state information, which poses a significant challenge.

Table 8: Comparison of Original and Processed C Code

Original Code

```
extern void abort(void);
extern void __assert_fail(const char *,
     const char *, unsigned int, const
     char *) __attribute__ ((__nothrow__ ,
      __leaf__)) __attribute__ ((
     __noreturn__));
void reach_error() { __assert_fail("0", "
     nested3-2.c", 3, "reach_error"); }

void __VERIFIER_assert(int cond) {
  if (!(cond)) {
    ERROR: {reach_error();abort();}
  }
  return;
}

int main()
{
  unsigned int x = 0;
  unsigned int y = 0;
  unsigned int z = 0;
  unsigned int w = 0;

  while (x < 0x0fffffff) {
    y = 0;

    while (y < 0x0fffffff) {
    z =0;
  while (z <0x0fffffff) {
    z++;
  }
    __VERIFIER_assert(!(z % 4));
  y++;
    }
    __VERIFIER_assert(!(y % 2));

    x++;
  }
  __VERIFIER_assert(!(x % 2));
 return 0;

}
```

Processed Code

```
extern void abort(void);

void VERIFIER_assert(int cond) {
  if (!(cond)) {
    {abort();}
  }
  return;
}

int main()
{
  unsigned int x = 0;
  unsigned int y = 0;
  unsigned int z = 0;
  unsigned int w = 0;

  while (x < 0x0fffffff) {
    y = 0;

    while (y < 0x0fffffff) {
    z =0;
  while (z <0x0fffffff) {
    z++;
  }
    VERIFIER_assert(!(z % 4));
  y++;
    }
    VERIFIER_assert(!(y % 2));

    x++;
  }
  VERIFIER_assert(!(x % 2));
 return 0;

}
```

D.3 Case Study

Table 7 demonstrates compared generated proofs by Mistral-7B and FVEL-Mistral-7B after being fine-tuned with FVELER. The upper row shows a case in which FVEL-Mistral-7B correctly applies the lemma learned from fine-tuning, thus correcting and simplifying the proof. Contrastively, Mistral-7B generates common `not_eq_complement` without considering a reasonable proof strategy, resulting in a failed proof. In the second case, Mistral-7B rewrites the lemma statement into "`assumes`" and "`shows`" statements, according to which gives an incorrect proof. FVEL-Mistral-7B, on the other hand, expands the brackets in the equation and then is able to derive contradiction according to "(b AND NOT b)", and completes the proof via the contradiction of the right-hand side of the equation.

E Implementations Details on Code2Inv and SV-COMP

This section provides supplementary details regarding the benchmark study in Section 5.

E.1 Evaluation Datasets

Code2Inv [35]. The code2inv dataset contains 133 programs in c, each containing a pre-condition, a loop body (while or for statement), and a post-condition. The verifier needs to verify that the post-condition (an assertion) holds. It is worth pointing out that the condition of a loop or branch in the program may be indeterminate.

SV-COMP [2]. The Software-Verification Competition provides a diverse set of benchmarks for formal verification. sv-comp benchmark contains over 23k c programs, which tend to be more complex than those in code2inv, and each program is accompanied by a .yml file to declare its specifications. These specifications cover requirements such as ReachSafety, MemSafety, ConcurrencySafety, NoOverflows, Termination, etc. The verifier is required to determine whether a program satisfies the given specifications. We sampled the SV-COMP benchmark into two subsets: a 47-sample subset sampled by Lemur [39], which contains samples with multiple nested loops, and a 1,000-sample subset which is randomly sampled from the full set. In particular, we exclude samples that contain floating-point type because the C-parser cannot parse them correctly.

E.2 Pre-processing

Table 8 demonstrates a randomly selected sample before pre-processing (original code) and after pre-processing (processed code). The pre-processing stages are explained as follows.

Data Preprocess. Since C-parser supports only part of the C99 standard, some C features (e.g. "`goto`" statements, side effects in expressions, etc.) are not supported, we normalize the C code to make C-parser work properly. Specially, for C code which includes:

String Literal and Illegal Function Name. Functions with string literals are often used to give warnings to the verifier, we remove these functions and keep only "`extern void abort(void);`" In addition, we fix illegal function names, for example, by removing the underlines at the beginning of the name.

Assertion and Assumption. We replace all the "`assert(statement);`" and "`assume(statement);`" with "`if (not (statement) {return -1;}`". Note that all assertions appear in the "`main()`" function, so the semantics before and after the replacement are equivalent.

Unknown Condition. "`unknown()`" is often used in the Code2Inv dataset as a condition in "`while`" or "`if`" expressions, and we add external declarations to this function: "`extern int unknown(void);`".

E.3 Fine-tuning and Inference

See Appendix D.1 for fine-tuning and inference details.