Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification

Kyle Thompson
University of California
San Diego, CA, USA
r7thompson@ucsd.edu

Nuno Saavedra
INESC-ID & IST, University of Lisbon
Lisbon, Portugal
nuno.saavedra@tecnico.ulisboa.pt

Pedro Carrott
Imperial College London
London, UK
pedro.carrott@imperial.ac.uk

Kevin Fisher
University of California
San Diego, CA, USA
kfisher@ucsd.edu

Alex Sanchez-Stern
University of Massachusetts
Amherst, MA, USA
sanchezstern@cs.umass.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

João F. Ferreira
INESC-ID & IST, University of Lisbon
Lisbon, Portugal
joao@joaoff.com

Sorin Lerner
University of California
San Diego, CA, USA
lerner@cs.ucsd.edu

Emily First
University of California
San Diego, CA, USA
emfirst@ucsd.edu

***Abstract*—Formal verification using proof assistants, such as Coq, enables the creation of high-quality software. However, the verification process requires significant expertise and manual effort to write proofs. Recent work has explored automating proof synthesis using machine learning and large language models (LLMs). This work has shown that identifying relevant premises, such as lemmas and definitions, can aid synthesis. We present Rango, a fully automated proof synthesis tool for Coq that automatically identifies relevant premises and also similar proofs from the current project and uses them during synthesis. Rango uses retrieval augmentation at every step of the proof to automatically determine which proofs and premises to include in the context of its fine-tuned LLM. In this way, Rango adapts to the project and to the evolving state of the proof. We create a new dataset, CoqStoq, of 2,226 open-source Coq projects and 196,929 theorems from GitHub, which includes both training data and a curated evaluation benchmark of well-maintained projects. On this benchmark, Rango synthesizes proofs for 32.0% of the theorems, which is 29% more theorems than the prior state-of-the-art tool Tactician. Our evaluation also shows that Rango adding relevant proofs to its context leads to a 47% increase in the number of theorems proven.**

***Index Terms*—Formal Verification, Theorem Proving, Large Language Models, Retrieval Augmentation, Software Reliability**

I. Introduction

The cost of poor software quality is alarmingly high, with estimates suggesting that it incurs trillions of dollars in losses annually in the United States alone [43]. Formal verification, which enables developers to mathematically prove that software adheres to its intended behaviors and specifications, has been shown to help improve software quality. Notably, a study investigating C compilers [92], among them CompCert [47], LLVM, and GCC, observed that the only compiler for which no bugs were found was CompCert [92], which is formally verified using the Coq proof assistant.

While formal verification can lead to high-quality software, it is expensive. For example, the code required to verify CompCert is 8 times as large as the code implementing its functionality [47]. A promising line of work towards automating formal verification is to automatically generate the proofs using machine learning techniques [10], [24], [25], [45], [73], [74]. Within this research space, there has been a recent and exciting line of work on exploring large language models (LLMs) for proof generation [32], [36], [37], [70], [91].

Prior LLM-based approaches for proof automation have shown that *premises*, such as lemmas and definitions, are important to add to the context [54], [91]. This builds off of *retrieval-augmented generation (RAG)* [49] approaches, which use a separate search step to add context to an LLM. For the task of proof synthesis, we call this technique *retrieval-augmented proving (RAP)*, where a separate search step retrieves relevant information for proving a given theorem. One limitation of prior approaches using RAP is that they do not fully exploit the local information that is available when synthesizing proofs.

In this paper, we show that an essential component of RAP is to *also* include similar proofs in the context – not only at the beginning of the proof, but to continue to give the LLM sources of inspiration and knowledge, adapting to the evolving proof state. The intuition is that having similar proofs in the context of the LLM, as determined at each step in the proof, can help guide the LLM in the right direction. Our work also distinguishes itself within the broader scope of all machine-learning-based proof generation approaches by using *both* premises and proofs in an online setting.

We reify this idea of adding relevant proofs, not just premises, to the context of an LLM in an approach and tool called

Rango. At each proof step, Rango first determines which proofs and premises from the current project are most relevant for generating the next step. By retrieving in-project information, Rango is able to learn local proof strategies, adapting itself to the current project. Then, the next step is generated using a language model where the most relevant proofs and premises are given as context in addition to the current proof script (the steps taken so far) and the proof state (a set of logical formulas describing the goals that remain to be proven). Furthermore, the assessment of which proofs are relevant is done *at each step in the proof*, thus being able to adapt to the evolving nature of the proof.

To train Rango effectively, we collected a new large corpus of Coq data, which we call CoqStoq. We mine this corpus from GitHub using the CoqPyt Python client for CoqLSP [13]. We split CoqStoq into training data and a benchmark, on which we evaluate Rango. The benchmark contains all of the projects from a previous benchmark, CoqGym [90], that compile in Coq 8.18. Additionally, it contains CompCert and four projects from the Coq Community repository that are committed to long-term maintenance [17].

In summary, this paper's main contributions are:

- An approach, Rango, to synthesizing proofs that adds to the context of the LLM not just premises but also relevant proofs. In this way, Rango adapts to the project *and* to the proof state at each step.
- A new dataset, CoqStoq, for proof synthesis in Coq, comprising 196,929 theorems, and 2,225,515 proof steps from 2,226 different GitHub repositories. The dataset is split into training data and an evaluation benchmark.
- An evaluation on CoqStoq comparing Rango to three state-of-the-art systems, Proverbot9001 [73], Tactician [45], and Graph2Tac [10]. Our evaluation shows that Rango does better than these tools, by proving 29% more theorems than Tactician, 66% more than Proverbot9001 and 4% more than Graph2Tac. Our evaluation also shows that adding relevant proofs to the context in Rango is important, leading to a 47% increase in the number of theorems proven.

We release Rango, CoqStoq, all trained models appearing in this paper, and all of the code required to reproduce the experiments in this paper at this link: https://github.com/rkthomps/coq-modeling.

II. Motivating Example in Coq

To motivate our approach, we explain how formal verification works in Coq, and then demonstrate through a real example how Rango helps automate the proof-writing process.

In Coq, a proof engineer can state a theorem about their code and then write a proof that the theorem holds true. Theorems in Coq are types, and so proving them true amounts to constructing a *proof term* of the same type in Coq's Gallina language. However, since writing that term directly is challenging, proof engineers typically write *proof scripts* in Coq's Ltac language, which consist of *tactics*, such as `induction`. When executed, these tactics guide Coq in the construction of a complete proof term. Coq provides feedback after each tactic application and displays the current *proof state*, which includes the goals left to prove and the local context of assumptions. The proof engineer knows that they have proven the theorem when there are no more goals.

A *proof development* in Coq consists of theorems and their associated proof scripts, potentially across multiple files. A proof engineer may even prove a series of lemmas with the sole intention of using them to prove a main theorem. Across a proof development, proof engineers often reuse bits and pieces from their proofs. However, different proofs have meaningful differences in, say, which lemmas are used. When building tools to help automate a proof engineer's proving process, it is important to fully utilize the expertise provided by the proof engineer. We accomplish this by directly leveraging information from existing proofs.

Let's take a closer look at the CompCert proof development, which is in CoqStoq's benchmark. The file `Memdata.v` has the following theorem.

```
Lemma memval_inject_compose:
  forall f g v1 v2 v3,
  memval_inject f v1 v2 -> memval_inject g v2 v3 ->
  memval_inject (compose_meminj f g) v1 v3.
```

To prove such a theorem, one can attempt to apply tactics step-by-step, considering the current proof state and context of assumptions. Alternatively, one can instead, at each step, also take inspiration from existing proofs in the project. This is the Rango approach. At each step, the Rango tactic prediction model draws inspiration from the following proof from the file `Values.v` in the CompCert proof development, just one of the many that it retrieves.

```
Lemma val_inject_compose: forall f g v1 v2 v3, Val.
    inject f v1 v2 -> Val.inject g v2 v3 -> Val.inject
    (compose_meminj f g) v1 v3.
Proof.
  intros. inv H; auto; inv H0; auto. econstructor.
  unfold compose_meminj; rewrite H1; rewrite H3; eauto.
  rewrite Ptrofs.add_assoc. decEq. unfold Ptrofs.add.
    apply Ptrofs.eqm_samerepr. auto with ints.
Qed.
```

Rango updates its sources of inspiration and knowledge at each step, as Rango also adapts to the current proof state. The proof script for `val_inject_compose` would not work outright for the theorem in question, but a part of it is useful, and so Rango begins to write the proof of `memval_inject_compose` as follows.

```
Proof.
  intros. inv H; inv H0; econstructor.
```

As Rango generates the next tactic, it relies both on learned knowledge from fine-tuning and on retrieved knowledge in the form of lemmas and proofs in the project. Later in the proof, Rango retrieves the following proof from earlier in `Memdata.v`.

```
Lemma memval_inject_incr: forall f g v1 v2,
    memval_inject f v1 v2 -> inject_incr f g ->
    memval_inject g v1 v2.
Proof.
  intros. inv H; econstructor. eapply val_inject_incr;
    eauto.
Qed.
```

Rango also identifies `val_inject_compose` as a relevant helper lemma. It determines that it may be able to use `val_inject_compose` like `val_inject_incr` and closes the proof as follows.

```
Proof.
  intros. inv H; inv H0; econstructor. auto.
  eapply val_inject_compose; eauto.
Qed.
```

Without retrieval, a tool could get lucky and "hallucinate" that certain lemmas, like `val_inject_compose` exist. However, Rango does not have to rely on luck because it can retrieve relevant lemmas and proofs.

Thus, through retrieving both proofs and lemmas from the project at each step, Rango is able to prove the theorem in question, while other state-of-the-art tools like Tactician and Proverbot are not.

III. The Rango Approach

Rango synthesizes Coq proofs using relevant proofs and lemmas from the current project at every step of the proof, adapting to the project and to the state of the proof. To do this, Rango uses two subsystems. The first subsystem, the *tactic generator*, generates individual proof steps, or *tactics*. The second subsystem, the *searcher* (Section III-D), uses the tactic generator to search for a complete proof by composing generated tactics. Figure 1 illustrates the interaction between the tactic generator and the searcher.

Rango's tactic generator can be further broken down into three components. The first two components, the *proof retriever* (Section III-A) and the *lemma retriever* (Section III-B), determine which proofs and lemmas, respectively, are relevant for generating the next tactic. The third component, the *language model* (Section III-C), takes as input relevant proofs, relevant lemmas, a proof script, and a proof state, and then generates the next tactic in the proof.

In the remainder of this section, we describe each of the three components in the tactic generator, and we describe the searcher.

A. Proof Retriever

The proof retriever selects relevant previously completed proofs in the current project to provide as context to the language model as it generates the next tactic. At a given proof step, the proof retriever selects from a set of proofs called the *proof bank*. The *proof bank* consists of proofs from earlier in the file, or from one of the file's dependencies. Only proofs from the current project are in the proof bank. At every proof step, the proof retriever selects the k most relevant proofs from the proof bank for generating the next tactic.

To find the k most relevant proofs at a given point in the proof, the proof retriever compares the current proof state, s, with proof states from proofs in the proof bank. The proof retriever defines the relevance of a proof P from the proof bank to be the maximum similarity between s and a proof state s_i from P. Specifically, given a function *similarity* that determines the similarity between two proof states, the proof retriever defines the relevance of a proof in the proof bank as

$$\textit{relevance(P)} = \max_{s_i \in \textit{states(P)}} \textit{similarity}(s, s_i)$$

where *states(P)* is the set of proof states in P.

To determine the similarity between two proof states, the proof retriever uses the BM-25 information retrieval technique [72]. Given a set of documents and a query, BM-25 assigns each document a score based on its relevance to the given query. BM-25 determines the relevance of a document by comparing the word frequencies from the document to the word frequencies from the query. If the document and the query have similar word frequencies, then BM-25 considers the document to be relevant. In its calculation, BM-25 downweighs words that appear across many documents as these words are often not helpful for determining relevance. Note that BM-25 is a *sparse retrieval* technique because it retrieves based on word frequencies.

When the proof retriever uses BM-25, "documents" correspond to proof states from proofs in the proof bank, and the "query" corresponds to the current proof state. The "words" in a proof state are the identifiers used in the proof state. We then define *similarity* in terms of BM-25 as follows: the *similarity* between the current proof state s and a proof state from a proof in the proof bank s_i is defined to be the relevance assigned to s_i by BM-25.

Note that information retrieval techniques other than BM-25 can be substituted into Rango's proof retriever. In Section V-E, we explore the use of retrieval techniques that rely on neural networks, known as *dense retrieval* techniques.

B. Lemma Retriever

The lemma retriever retrieves the statements of the lemmas previously defined in the current project that could be directly used in the current proof. Note that the lemma retriever does not retrieve the proofs of these lemmas, just the statements.

The structure of the lemma retriever is similar to the structure of the proof retriever. The lemma retriever has access to the *lemma bank*, which is defined as the set of lemmas appearing earlier in the current file, or in one of the file's dependencies. The lemma bank only considers lemmas from the current project.

Like the proof retriever, the goal of the lemma retriever is to select the j most relevant lemmas. The lemma retriever uses the sparse retrieval algorithm TF-IDF [77] to assign a relevance score to each lemma in the lemma bank with respect to the current proof state. Note that in this case, the "query" given to TF-IDF is the current proof state, the set of "documents" are the lemmas in the lemma bank, and the "words" in a lemma correspond to the identifiers in the lemma. Again, like in the proof retriever, the lemma retriever can use information retrieval techniques besides TF-IDF.

C. Language Model

At a given proof step, the language model generates the next tactic using the following inputs:

- *relevant proofs* retrieved from the proof retriever

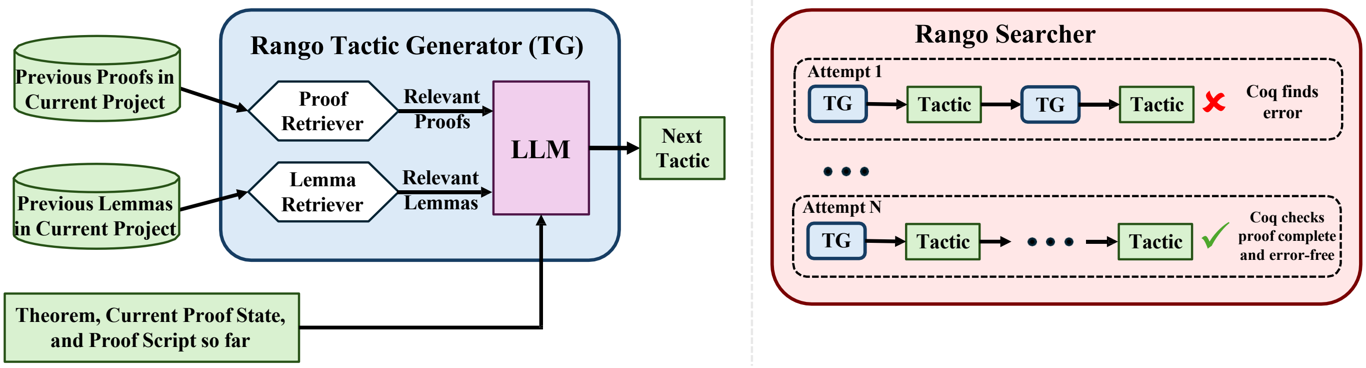
Previous Proofs in Current Project
Previous Lemmas in Current Project
Rango Tactic Generator (TG)
Proof Retriever
Relevant Proofs
Lemma Retriever
Relevant Lemmas
LLM
Next Tactic
Theorem, Current Proof State, and Proof Script so far
Rango Searcher
Attempt 1
TG
Tactic
TG
Tactic
Coq finds error
Attempt N
TG
Tactic
Tactic
Coq checks proof complete and error-free

Fig. 1. Overview of Rango's architecture. Rango's tactic generator uses retrieved relevant proofs and lemmas from the current project as input to an LLM (along with the theorem, current proof state, and proof script so far) to predict the next tactic. Rango's searcher uses the tactic predictions to attempt to synthesize a complete proof. A proof attempt is correct if Coq determines that it has no errors and there are no more goals left to solve.

- *relevant lemmas* retrieved from the lemma retriever
- the *theorem statement* and the *proof script* thus far
- the current *proof state*

To obtain a language model that can effectively use this information, we *fine-tune* a pretrained decoder-only LLM. We construct fine-tuning examples from a set of training projects (see Section IV). Each fine-tuning example consists of a prompt containing the four inputs mentioned above, and a target containing the next tactic. Since the language model is a decoder-only LLM, the inputs and targets are concatenated during fine-tuning. Following prior work [26], the loss is only computed over the target so that the model learns to conditionally generate the target given the input and not the input itself.

We create each training example exactly as we would during inference. That is, we run the proof retriever and lemma retriever at every proof step in our dataset, and then construct the prompt using the retrieved proofs and lemmas.

Note that language models can only process a limited number of tokens. To account for this constraint, we allocate a maximum number of tokens to each of the four inputs in each training example. Furthermore, we allocate a maximum number of tokens that can be generated by the language model. We truncate each input as follows. We keep the largest whole number of relevant proofs that does not exceed the token limit. We do the same for relevant lemmas. We keep the longest suffix of the theorem statement and proof script that does not exceed the token limit. We keep the longest suffix of the proof state that does not exceed the token limit. We truncate the output at training time by keeping the longest prefix that does not exceed the token limit. At inference time, we limit the number of tokens that the model can generate.

D. Searcher

Given a tactic generator, the searcher attempts to find a sequence of tactics that completes the proof. To find a sequence of tactics, the searcher uses a procedure that we call *rollout search*. Rollout search consists of a sequence of *rollouts*, where in each rollout, the searcher samples a tactic from the tactic generator using temperature sampling. The searcher then appends the tactic to the current proof, and uses Coq to check the resulting proof attempt. The proof attempt will be in one of three states: complete, invalid, or incomplete. If the proof attempt is complete, meaning that Coq shows no more goals to be proven, then the search is successful. If the proof attempt is invalid, then the tactic sampled from the language model resulted in an error, and the searcher begins a new rollout. If the proof attempt is incomplete, the searcher continues the current rollout by sampling yet another tactic from the tactic generator. The searcher executes rollouts until it finds a complete proof, or until a timeout.

IV. The CoqStoq Dataset

As part of our work we created CoqStoq, a new dataset of Coq proofs mined from open-source GitHub projects. We collect theorems and their respective proofs from all open-source repositories that listed Coq as its primary language as of November 5th, 2023. We applied no other filters to our data collection (e.g., stars or number of contributors), since Coq itself ensures that successfully compiled files provide sound proof data.

We first attempted to automatically compile each repository by executing any existing *Makefile* or by compiling each individual file in the order specified by a *_CoqProject* file present in the repository. Then, we used CoqPyt [13] to validate each Coq file. We considered a file to be valid if it reports no errors during compilation. We only included repositories in our training dataset with at least 1 valid file. We note that both the compilation and validation steps are executed using Coq 8.18, so files not compatible with this version are excluded. For each valid Coq file obtained, we used CoqPyt to extract information from each proof step in the file. For each proof step, we extracted its textual representation, corresponding proof state, and its context. The context of a proof step includes the premises used in the proof step. We

TABLE I
ATTRIBUTES OF THE TRAINING SET, BENCHMARK, AND VALIDATION SET.

Attribute	**Count**			
	Training	*Benchmark*	*Validation*	*Total*
Repositories	2, 208	12	6	2, 226
Theorems / Proofs	181,562	10,396	4,971	196,929
Proof States / Steps	2,008,543	162,989	53,983	2,225,515

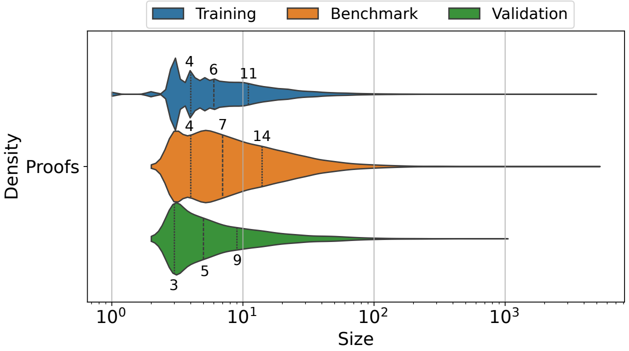

Fig. 2. Violin chart over the size of proofs in the training set, benchmark, and validation set. Proof size is the number of steps in the proof. The quartiles are indicated near the corresponding dashed lines.

also use CoqPyt to collect the set of premises that are available at every point in the file.

After collecting CoqStoq, we split it into a training set, a validation set, and a benchmark (test set). We use the training set to train the language model. We use the benchmark to evaluate Rango against other tools, and to evaluate the individual components of Rango. We use the validation set to tune model hyper-parameters, and to experiment with different configurations of our tool.

CoqStoq's benchmark consists of all CoqGym [90] projects that compiled in Coq 8.18. CoqGym is a benchmark that was used to evaluate previous tools [24], [25], [73], for which the projects were compiled in Coq 8.9. We also added CompCert [48] to our set of test projects to evaluate how Rango could help formally verify real-world software. Finally, we add projects from the Coq-Community that are committed to long-term maintenance to our benchmark and validation set. We put any project not in our benchmark or validation set in the training set. Finally, to prevent against "copy-and-pasted" theorems, we omit files from our training set where a theorem statement exactly matched a theorem statement from the validation set or the benchmark.

Table I shows how the relevant proof data is numerically distributed across all splits. The training set contains a substantial portion of the repositories, proofs, and proof steps in comparison to the benchmark and validation set. These results showcase the abundance of proof data used for training, which is guaranteed to pertain to valid Coq proofs due to the use of CoqPyt during the mining process.

Figures 2 and 3 plot the distribution of proofs and projects, respectively, according to their size. In terms of proof size, the number of steps per proof follows a similar distribution for all splits. In terms of project size, the training set contains mostly smaller projects with fewer proofs, while the reverse is true for the benchmark and validation set. This asymmetry is justified as we intend to evaluate how the training generalizes to other, possibly larger, projects.

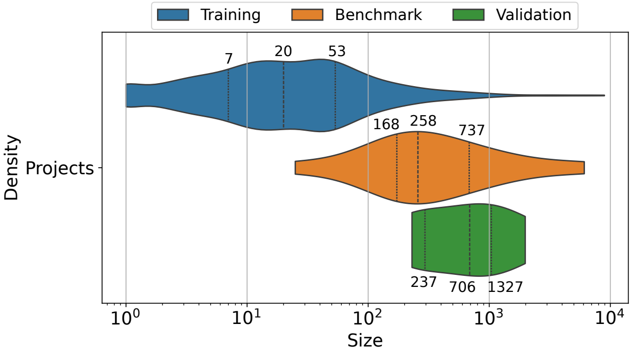

Fig. 3. Violin chart over the size of projects contained in the training set, benchmark, and validation set. Project size is the number of proofs in the project. The quartiles are indicated near the corresponding dashed lines.

V. EVALUATION

To understand Rango's performance, we propose the following research questions:

RQ1: How does Rango compare against other proof synthesis tools in Coq?

RQ2: How do the proof retriever and the lemma retriever contribute to Rango's ability to synthesize proofs?

RQ3: How does adapting proof retrieval to the current proof step contribute to Rango's ability to synthesize proofs?

RQ4: How do alternative retrieval techniques perform in Rango's proof retriever?

RQ5: How does Rango perform when it is instantiated with a naïve retrieval algorithm?

RQ6: How does Rango's rollout search compare to best-first search?

RQ7: What kinds of theorems can Rango prove and how do the proofs generated by Rango compare to the proofs generated by other tools?

A. Experimental Setup

Rango's language model is a fine-tuned version of the 1.3 billion parameter model DeepSeek-Coder [31]. We fine-tune the LLM on a set of examples gathered from the training projects in our dataset CoqStoq. We fine-tune for 60,000 training steps with a batch size of 16 on 4 NVIDIA A100 GPUs. We use 2 gradient accumulation steps so that our effective batch size is 32. We choose the checkpoint with the best loss on our validation set. We fine-tune using LoRA [34] and FSDP [65]. We use the Adam Optimizer [42] with a learning rate of 10^{-3}. We allocate 1,024 tokens to retrieved proofs, 512 tokens to retrieved lemmas, 512 tokens to the theorem and proof script, 1,024 tokens to the proof state, and 128 tokens to the output.

TABLE II
COMPARISON OF THEOREMS PROVEN BETWEEN RANGO AND STATE-OF-THE-ART PROOF SYNTHESIS TOOLS

Project	**CoqStoq Evaluation**				**Graph2Tac Evaluation**		
	Rango	*Tactician*	*Proverbot*	# Theorems	*Rango* *	*Graph2Tac**	# Theorems
CompCert	**1,977 (32.5%)**	1,427 (23.4%)	1,308 (21.5%)	6,091	—	—	—
FourColor	**212 (15.8%)**	155 (11.6%)	133 (9.9%)	1,341	—	—	—
MathClasses	**303 (39.7%)**	242 (31.7%)	98 (12.8%)	763	—	—	—
BuchBerger	**180 (27.4%)**	150 (22.8%)	103 (15.7%)	658	—	—	—
RegLang	**42 (13.2%)**	38 (11.9%)	29 (9.1%)	318	—	—	—
PolTac	**216 (83.4%)**	**216 (83.4%)**	140 (54.1%)	259	127 (83.5%)	**138 (90.8%)**	152
Huffman	**82 (32.0%)**	57 (22.3%)	47 (18.4%)	256	—	—	—
Zfc	75 (36.2%)	**78 (37.7%)**	37 (17.9%)	207	**75 (36.2%)**	59 (28.5%)	207
ZornsLemma	**51 (29.1%)**	36 (20.6%)	22 (12.6%)	175	—	—	—
ExtLib	**105 (63.6%)**	102 (61.8%)	32 (19.4%)	165	—	—	—
DBLib	**74 (53.6%)**	67 (48.6%)	51 (37.0%)	138	**74 (54.0%)**	68 (49.6%)	137
HoareTut	**8 (32.0%)**	7 (28.0%)	7 (28.0%)	25	—	—	—
Total	**3,325 (32.0%)**	2,575 (24.8%)	2,007 (19.3%)	10,396	**276 (55.6%)**	265 (53.4%)	496
Rango + Tool	—	3,866 (37.2%)	3,724 (35.8%)	10,396	—	319 (64.3%)	496

TABLE III
POST TRAINING CUTOFF COMPARISON OF THEOREMS PROVEN BETWEEN RANGO AND STATE-OF-THE-ART PROOF SYNTHESIS TOOLS

	Rango	*Tactician*	*Proverbot*	# Theorems
BB5	**186 (38.5%)**	163 (33.7%)	104 (21.5%)	483
PnV	166 (24.1%)	**168 (24.4%)**	113 (16.4%)	688
Total	**352 (30.1%)**	331 (28.3%)	217 (18.5%)	1,171

During proof synthesis, Rango is allocated a single NVIDIA RTX 2080 for inference, and a single CPU with 16GB of RAM for proof checking. We use a 10 minute timeout for all of our proof attempts. Our timeout does not include the initialization costs of loading and compiling the file. We use a temperature of 1.0 for sampling.

*B. **RQ1**: Rango versus Other Tools*

We evaluate Rango against three state-of-the-art proof synthesis tools for Coq: Proverbot9001 [73], Tactician [45], and Graph2Tac [10]. Proverbot9001 (which we refer to as Proverbot) uses a custom architecture involving several Gated Recurrent Units and feed-forward neural networks [73]. Graph2Tac also uses a custom architecture based on Graph Neural Networks. It uses this architecture both to predict which tactic to use next, and which definitions in the environment (including helper lemmas) should be given as an argument to the tactic. Therefore, Graph2Tac can retrieve helper lemmas and definitions from the environment just like Rango. Lastly, Tactician maintains a database of proof states which is defined similarly to Rango's proof bank. At every proof step, Tactician finds the most similar proof states in its database by comparing sets of identifiers using k-NN [45]. Then, Tactician performs a search by directly applying tactics that were used at similar proof states.

We evaluate Rango, Tactician, and Proverbot on all 12 projects from CoqStoq's benchmark using Coq 8.18. Note that neither Tactician nor Proverbot are built to utilize a GPU during their respective proof search procedures. Therefore, we run each tool using a single CPU. We run Tactician using a 10 minute timeout. Proverbot is configured to run with depth limits instead of timeouts, so for most of the reported proofs, Proverbot fails before 10 minutes. In Table II, we report the results for Rango, Tactician, and Proverbot on the 10,396 theorems in the CoqStoq benchmark. Rango finds 29% more proofs than Tactician, and 66% more proofs than Proverbot.

In Table II, we also report results for Graph2Tac. Note that since Graph2Tac is only compatible with Coq 8.11, we evaluate Graph2Tac on different project versions than Rango. Furthermore, there are only 3 of the projects in CoqStoq's benchmark that Graph2Tac was not directly trained on. Therefore, we can only fairly evaluate on these three projects. We only compare theorems whose statements match exactly between project versions. Note that this does not guarantee that a proof in one project version will translate to a proof in the other project version since other definitions in the project may have changed between versions. For each theorem, we ran Graph2Tac on a single NVIDIA RTX 2080 and a single CPU with a timeout of 10 mins. Keeping differences between project versions in mind, Rango proves 4% more theorems than Graph2Tac.

In the last row of Table II, we show the combined number of theorems proven between Rango and each other tool. We can see that each tool finds proofs for a significant subset of theorems where Rango could not find a proof. Running Rango alongside Tactician, Proverbot, and Graph2Tac leads to 16%, 12%, and 16% respective increases in the number of theorems proven over running Rango alone.

One limitation of the CoqStoq benchmark is that its projects were created before the pretraining cutoff of Rango's underlying LLM. So, although we *do not* fine-tune on the projects in the CoqStoq benchmark, there is a risk that Rango's underlying LLM saw them during pre-training. Therefore, we evaluate Rango on two projects whose first commit on GitHub occurred after the pretraining cutoff of Rango's underlying LLM [1], [2]. The results of this evaluation are shown in Table III. On these

TABLE IV
PROOF & LEMMA RETRIEVER ABLATION

System	Theorems Proven
Rango	150/500 = 30.0%
Rango w/o Lemmas	145/500 = 29.0%
Rango w/o Proofs	102/500 = 20.4%
Rango w/o Retrieval	93/500 = 18.6%

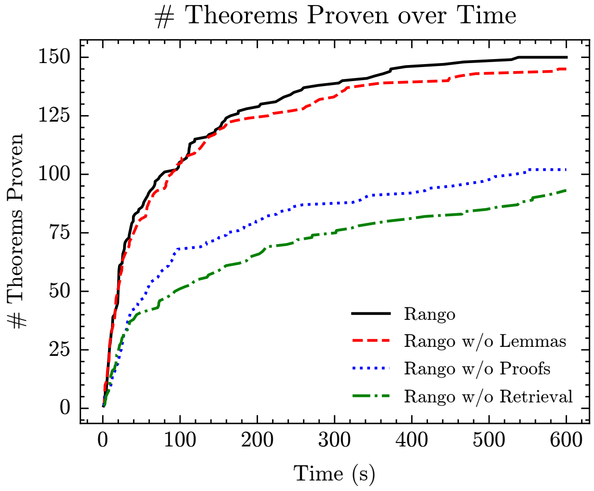

Fig. 4. Number of Theorems proven by Rango variants over time in seconds.

two projects, Rango proves 6% more theorems than Tactician and 62% more theorems than Proverbot. Unfortunately, we cannot compare Rango to Graph2Tac on these projects because they do not compile with Coq 8.11.

Takeaway 1: Rango synthesizes 29% more proofs than Tactician and 66% more proofs than Proverbot9001 on CoqStoq's benchmark. Rango also synthesizes 4% more proofs than Graph2Tac on its subset of CoqStoq's benchmark.

C. *RQ2: Contribution of Proof and Lemma Retrievers*

To determine the contribution of the proof retriever and the lemma retriever to Rango's performance, we train one version of Rango without the proof retriever, one version without the lemma retriever, and one version with neither the proof retriever nor the lemma retriever. We evaluate each of these variants on a randomly selected subset of 500 theorems from CoqStoq's benchmark. We call this subset of theorems our *ablation set*. We report the percentage of theorems that each of these variants prove in Table IV.

Table IV shows that retrieval-augmentation is essential to Rango's success. Rango proves 61% more theorems than the variant without retrieval. Table IV also shows that the proof retriever contributes to Rango's success more than the lemma retriever. Indeed, Rango proves 47% more theorems than the variant without a proof retriever whereas Rango only proves 3% more theorems than the variant without a lemma retriever. This is indeed one of our main contributions: we show that adding relevant *proofs*, not just lemmas (as prior work had done) is important to making *retrieval-augmented proving* perform better. We show the number of proofs found by Rango and its variants over time in Fig 4, which visualizes the large effect of Rango's proof retriever.

TABLE V
PROOF & LEMMA RETRIEVER ABLATION ON A RANDOM FILE-WISE SPLIT

System	Theorems Proven	
	CoqStoq Split	File-Wise Split
Rango	160/500 = 32.0%	184/500 = 36.8%
Rango w/o Lemmas	159/500 = 31.8%	177/500 = 35.4%
Rango w/o Proofs	111/500 = 22.2%	140/500 = 28.0%
Rango w/o Retrieval	89/500 = 17.8%	127/500 = 25.4%

We also investigate the contribution of Rango's proof and lemma retrievers to its ability to adapt to held-out projects. We show that Rango's proof and lemma retrievers capture information about a project that would otherwise need to be stored in the weights of the underlying LLM. We show this by first training a separate version of each variant on an *inter-file split* as opposed to CoqStoq's inter-project split. That is, we randomly split the *files* in the CoqStoq dataset into a training, validation, and testing set. Then, we train each variant on the training set of the inter-file split. This way, the inter-file versions of the variants have information about all CoqStoq projects in their weights. Then, we compare inter-file variants to inter-project variants on a subset of 500 theorems that are in the testing sets of *both* the inter-file split and the inter-project split. We show the results in Table V. From these results, we see that all variants benefited from having project-specific information in the weights of their underlying LLMs. However, the variants that did not have proof retrieval benefited more than variants that did have proof retrieval. Specifically, Rango without proofs and Rango without proofs and lemmas obtained increases of 26% and 43% in the number of theorems proven when they were trained on an inter-file split. In contrast, Rango and Rango without lemmas obtained more modest increases of 15% and 11%. This indicates that Rango's proof retriever captures a significant amount of the information that would be gained by training directly on files from the current project.

Takeaway 2: Rango's proof retriever and lemma retriever are imperative to its success. Together, they contribute to a 61% increase in the number of theorems proven, and Rango's proof retriever alone contributes to a 47% increase in the number of theorems proven. This demonstrates the importance of adding relevant proofs, not just lemmas as prior work had done.

D. *RQ3: Retrieving at each Proof Step*

Recall that Rango's proof retrieval mechanism adapts to the current proof state by retrieving relevant proofs at every step in the proof search. To measure the effect of retrieving

TABLE VI
RETRIEVING AT EACH PROOF STEP

Proof Retrieval Frequency	**Theorems Proven**
Every proof step	${}^{150}/_{500} = 30.0\%$
Only at beginning of proof	${}^{111}/_{500} = 22.2\%$

TABLE VII
PROOF RETRIEVAL VARIANTS

Proof Retrieval	**Theorems Proven**
BM-25	${}^{3,325}/_{10,396} = 32.0\%$
TF-IDF	${}^{3,291}/_{10,396} = 31.7\%$
Codebert	${}^{2,283}/_{10,396} = 22.0\%$

at each proof step, we compare Rango to a variant that only performs proof retrieval at the first step in the proof search. In other words, the variant retrieves proofs based on the theorem statement, and uses the same retrieved proofs throughout the search. We compare Rango with this variant on our ablation set and show the results in Table VI. We observe that Rango proves 35% more theorems by retrieving proofs at every proof step than it does by retrieving proofs solely at the beginning of the proof search. Thus, retrieving at every proof step is critical to Rango's success.

Takeaway 3: By retrieving similar proofs at every proof step, rather than solely retrieving at the beginning of the proof, Rango proves 35% more theorems.

E. *RQ4*: Comparing Retrieval Algorithms for Proof Retrieval

We compare *sparse retrieval* techniques to *dense retrieval* techniques for retrieving proofs. Sparse retrieval techniques, such as BM-25, are those that retrieve information based on word counts. In contrast, dense retrieval techniques retrieve information based on vector embeddings derived from a neural network.

Note that a comparison has been made between sparse retrieval techniques and dense retrieval techniques for the selection of premises [91]. However, the techniques for training models to select premises do not transfer to training models to retrieve proofs due to a difference in objectives. In premise selection, the objective is to predict whether or not a premise will be used in the next proof step. For proof retrieval, the objective is to retrieve the proofs that are most helpful for generating the next proof step. However, at training time, there is no way to know which proofs satisfy this objective. Therefore, standard supervised learning techniques are not applicable.

Recall that a main requirement of the proof retriever is the ability to compare proof states. Rango uses the BM-25 algorithm to determine the similarity between proof states. Alternatively, to compare two proof states, Rango could use a neural network to compute a vector embedding that contains semantic information about each proof state. Then, the similarity between two proof states could be defined as the cosine similarity between the vector embeddings. We implemented this kind of proof retriever using the CodeBert 125M parameter model [23]. We compare the proof retriever used in Rango to a version that uses dense embeddings in Table VII. We also include another popular sparse retrieval algorithm, TF-IDF in Table VII. Because the difference in performance between BM-25 and TF-IDF is small, we ran this ablation on the entire CoqStoq benchmark instead of our ablation set to ensure the precision of our comparison. Rango proves 46% more theorems than a variant that uses CodeBert embeddings to retrieve proofs. Thus, embeddings from CodeBert do not capture similarities between proof states that are relevant for proof retrieval. We also see that Rango proves 1% more theorems than a variant that uses TF-IDF for proof retrieval.

TABLE VIII
COMPARING RANGO, RANGO-PRE, AND A HYBRID APPROACH

Rango Variant	**CoqStoq**	**Cutoff**
Rango	${}^{3,325}/_{10,396} = 32.0\%$	${}^{352}/_{1,171} = 30.1\%$
Rango-PRE	${}^{3,259}/_{10,396} = 31.3\%$	${}^{350}/_{1,171} = 29.9\%$
Rango-Hybrid	${}^{3,478}/_{10,396} = 33.5\%$	${}^{380}/_{1,171} = 32.5\%$

Takeaway 4: Using the BM-25 sparse retrieval technique for proof retrieval led to 46% more proven theorems than using CodeBert dense embeddings and 1% more proven theorems than using another sparse retrieval technique, TF-IDF.

F. *RQ5*: Rango with Naïve Retrieval Variant

Rango uses a proof retriever and a lemma retriever to gather relevant context for a fine-tuned LLM to use when generating the next tactic in a proof. A naïve form of retrieving proofs and lemmas could use the lines directly preceding the theorem being proven as context to the LLM. We call this retrieval technique *prefix retrieval*. In this section, we explore Rango-PRE: a variant of Rango whose only retrieval mechanism is prefix retrieval. Table VIII shows Rango-PRE's performance on the CoqStoq benchmark, and on our two post-cutoff projects. Note that while Rango proves more theorems than Rango-PRE, the margin is small. We know that Rango-PRE cannot retrieve context outside of the current file. However, we observed that it still performed well compared to Rango, which can retrieve proofs and lemmas throughout a project. This led us to the following hypothesis: when the required context is in close proximity to the current theorem, Rango-PRE is preferable since it presents this context to the LLM as it was originally written. However, when the required context is not in close proximity to the current theorem, Rango is preferable since it is able to retrieve proofs and lemmas across the project. Following this hypothesis, we created Rango-Hybrid to capitalize on advantages from both Rango and Rango-PRE. Rango-Hybrid simply alternates between rollouts using Rango and Rango-PRE. We can see that Rango-Hybrid leads to a 4% increase in

TABLE IX
SEARCHER VARIANTS

Searcher	Theorems Proven
Rollout	$^{150}/_{500} = 30.0\%$
Best-First Search (Beam)	$^{142}/_{500} = 28.4\%$
Best-First Search (Temp)	$^{125}/_{500} = 25.0\%$

the number of theorems proven by Rango, and a 7% increase in the number of theorems proven by Rango-PRE.

> **Takeaway 5:** Rango proves more theorems than Rango-PRE. Rango and Rango-PRE can be combined into a hybrid search procedure that proves 4% more theorems than Rango, and 7% more theorems than Rango-PRE.

G. *RQ6: Searcher Variants*

We also explore proof search alternatives in Rango's searcher. Recall from Section III-D that Rango uses rollout search to search for a complete proof using its tactic generator. One weakness of the rollout search is that it does not use previous proof attempts to inform subsequent proof attempts. In this section, we compare rollout search to a *best-first search*, which is standard in LLM proof search implementations [32], [36], [37], [69], [91]. In our best-first search, we maintain a set of candidate proofs. In each search step, we select the candidate with the highest score given by Rango's language model. Then, we use the tactic generator to generate b distinct possible next tactics. Each tactic corresponds to a new candidate proof. We continue the search in this way until we either find a complete proof, or the search times out. Note that this search algorithm guarantees that each explored proof is distinct. Finally, we do not include invalid proofs or redundant proofs [73] as candidates.

We compare the searcher used in Rango, based on rollout search to two best-first search configurations. In one configuration, the searcher samples tactics at each search step using beam decoding. In the other configuration, the searcher samples tactics using temperature sampling. In each configuration, the searcher samples $b = 4$ tactics at each search step. Note that temperature sampling does not guarantee that the sampled tactics will be distinct. If the sample tactics are not distinct, we remove the duplicates.

We compare Rango's rollout search with these two best-first search configurations on our ablation set. We show the results of this comparison in Table IX. We can see from Table IX that Rango proves 6% more theorems than the configuration using beam decoding and 20% more theorems than the configuration using temperature sampling.

> **Takeaway 6**: Rollout search is a simple, yet effective technique for synthesizing proofs. When used in Rango, it synthesizes 6% more theorems than best-first search.

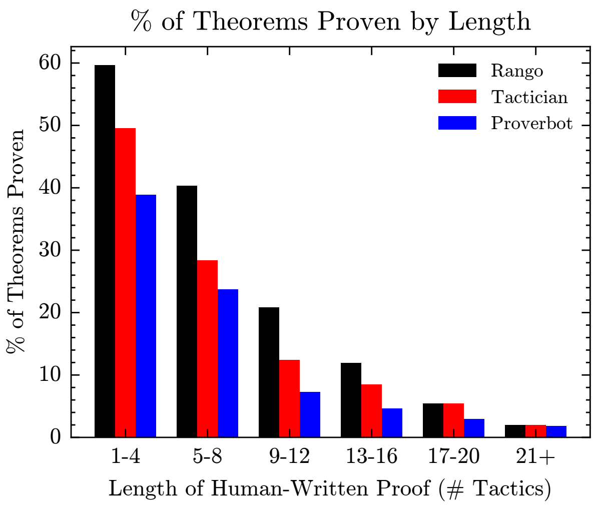

Fig. 5. Percentage of theorems proven by Rango, Tactician, and Proverbot by the length of the human-written proof.

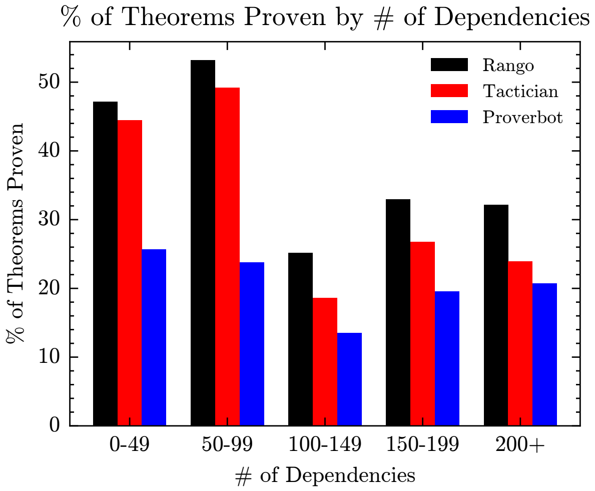

Fig. 6. Percentage of theorems proven by Rango, Tactician and Proverbot by the number of dependencies of the current file to other files in the project.

H. *RQ7: Understanding Rango's Proofs*

In this section, we investigate which kinds of theorems Rango can prove, and we compare the proofs generated by Rango to the proofs generated by other proof synthesis tools.

1) Kinds of Theorems Rango can Prove: The strongest indicator that we have found for whether or not Rango can prove a theorem is the length of its corresponding human-written proof. We show the success rate of Rango, Tactician, and Proverbot by human-written proof length in Figure 5. For all proof synthesis tools, Figure 5 shows a sharp decrease in the percentage of theorems proven as the length of human-written proofs increases.

A weaker indicator for whether or not Rango can prove a theorem is the number of dependencies of the file containing the theorem. We show the success rates for Rango, Tactician and Proverbot in Fig 6 for files with varying numbers of

TABLE X
PROOF LENGTH AND EDIT DISTANCE BETWEEN MACHINE-GENERATED AND HUMAN-WRITTEN PROOFS

System	**Proof Length in # Tactics**		**Edit Distance to Human Proof**	
	Mean	Median	Mean	Median
Rango	4.5	4	39.7	23
Tactician	4.6	3	52.8	37
Proverbot	6.7	6	61.2	48
Human	4.0	3	0	0

dependencies where a dependency is a Coq file that is imported either directly or transitively. We notice that the success rates for all three proof synthesis tools drop for files that have a hundred or more dependencies. Despite Rango's decreased success rate on files with many dependencies, it is still able to find more proofs than other tools. We speculate that Rango's retrieval mechanisms allow it to remain competitive in files with many dependencies. For example, consider the following theorem from the file `Values.v` in the CompCert proof development which has 300 dependencies.

```
Theorem swap_cmpu_bool:
  forall valid_ptr c x y,
  cmpu_bool valid_ptr (swap_comparison c) x y =
    cmpu_bool valid_ptr c y x.
```

Rango finds the following proof for this theorem using the lemma `Int.swap_cmpu` which is defined in a different file, and is not used anywhere in `Values.v` prior to this theorem.

```
Proof.
  destruct x; destruct y; simpl; auto.
  rewrite Int.swap_cmpu. auto.
Qed.
```

In this case, Rango's lemma retriever identified `Int.swap_cmpu` as a relevant lemma for this proof, and its proof retriever found proofs that inspired this proof's structure.

> **Takeaway 7.1**: Like other proof synthesis tools, Rango's success-rate has a strong relationship with the length of the human-written proof. We also found that the performance of Rango and other proof synthesis tools decreased for files with many dependencies.

2) Qualities of Proofs Generated by Rango: To understand the qualities of proofs generated by Rango, we measure two attributes: proof length and edit distance to the human-written proof. We measure these attributes for Rango, Tactician and Proverbot. To calculate proof length, we first collected the 1,252 theorems for which all three proof synthesis tools found proofs. Then, we calculated the number of tactics in each proof. We report the mean and median of these proof lengths in Table X. We also report the corresponding statistics for human-written proofs. Human-written proofs are shorter on average than proofs generated by all proof synthesis tools. Proofs generated by Rango and Tactician are of similar length, and are shorter on average than proofs generated by Proverbot.

We use string edit distance to the human-written proof as a measure of how "human-like" the proofs generated by Rango are. For each proof of the 1,252 theorems where Rango, Tactician, and Proverbot all found proofs, we computed the string edit distance between the proofs found by the proof synthesis tools, and their corresponding human-written proofs. We report these edit distances in Table X. On average, Rango produces proofs that are 12 edit operations closer to human-written proofs than Tactician, and 21 edit operations closer to human-written proofs than Proverbot. A prior study on proof engineers found that they are constantly making repetitive repairs across their proofs due to changes to specifications or dependencies [71]. It is likely that automatically generated proofs that are in a similar style to the proof engineer's hand-written proofs would be easier for them to repair and maintain, though future work should explore this.

> **Takeaway 7.2**: On average, the proofs synthesized by Rango have similar or shorter lengths and smaller edit distances to human-written proofs than the proofs generated by other proof synthesis tools.

I. Threats to Validity

A threat to internal validity is that while the pretraining data for LLMs often consist of data from public repositories, such as GitHub, it is not publicly known what is in the pretraining data for the LLM we fine-tune for Rango, DeepSeek-Coder 1.3B. Since CoqStoq's benchmark was taken from GitHub, it is possible that it intersects with the LLM's pretraining data. Most evaluations involving LLMs suffer from this *test set contamination* problem. We mitigate this issue by evaluating on two projects, Coq-BB5 and PnVRocqLib, that were created after the pretraining cutoff for DeepSeek-Coder 1.3B.

Another threat to internal validity is that, in our comparison of Rango to other tools, there are some differences in the Coq versions and machines used (CPU vs GPU). The evaluations of Rango, Tactician, and Proverbot all use Coq 8.18, and are all run on CoqStoq's benchmark. However, Graph2Tac only runs in Coq 8.11. We evaluated Rango and Graph2Tac on GPUs, while Tactician and Proverbot were run on CPUs since they are not intended to use GPUs.

A threat to external validity is that while the Rango approach could be implemented for other proof assistants, such as Isabelle and Lean, it is not known whether our results generalize across proof assistants. This is an interesting and important direction to be explored in future work.

VI. RELATED WORK

Formal verification aims to improve software quality, a problem that takes up 50–75% of the total software development budgets [63]. Other methods of improving software quality include debugging [11], [40], [93], and testing [5], but only formal verification can guarantee code correctness. Automated techniques can similarly improve program quality [3], [39], [46], [51], [57], [86], [96], and can also help developers debug manually [21], but still do not guarantee correctness, and, in fact, often introduce new bugs [58], [76].

Recent work in automating theorem proving in proof assistants, such as Coq [80], Lean [20], Isabelle [62], and Metamath [52], has focused on machine-learning based approaches. Typically, with a machine learning approach, *neural theorem prover* uses a model to predict the next tactic to apply, which guides a search through the space of possible proofs. Early work explored the use of LSTM [24], [25], [90], RNN [35], [73], and GNN-based models [66]. Recent work has focused on the use of LLMs in neural theorem proving, either fine-tuning models [32], [36], [37], [70], [91] or prompting pretrained ones [8], [38], [79], [95].

Large foundation models have demonstrated incredible capabilities on a wide range of tasks [16], [64], [78], [82]. To adapt to knowledge-intensive tasks and new domains, researchers have explored the use of retrieval-augmented generation (RAG) [49] to boost performance [15], including sparse [72], [77] and dense [41] retrieval techniques.

Recent work has explored retrieval augmentation for theorem proving, where they train models to retrieve premises, such as lemmas and definitions, that are relevant to the current proof goal, and then condition the next tactic generation on those premises [54], [91]. LeanDojo [91] trains a model to select which premises should be included as input to its LLM tactic generation model. Magnushammer [54], for Isabelle, takes this approach one step further and additionally trains a "reranker" model to prioritize which premises are its LLM tactic prediction model's input, though this extra step is costly. Prior work explored the use of TF-IDF for retrieving portions of premises [9] with the aim of training a reinforcement learning approach to theorem proving without access to human-written proofs, which our approach uses.

Most similar to our work are Graph2Tac [10] and Tactician [45], which explore online representation learning for Coq. Graph2Tac uses GNNs to incorporate information from new definitions and theorems. Tactician uses online *k*-NN to select tactics from other in-project proofs to apply in the current proof. Rango goes beyond these approaches by being able to retrieve both new lemmas and proofs to serve as input to an LLM tactic prediction model.

Baldur [26] uses in-project information, using the lines preceding a theorem as context to its whole-proof-generation LLM. Section V-F showed that Rango can be instantiated with this retrieval mechanism, and that a hybrid search procedure between Rango and Rango-PRE leads to the best results on the CoqStoq benchmark. LEGO-Prover [84] builds a skill library of lemmas and theorems while proving so that it may retrieve new skills learned instead of relying on a fixed library.

Neural theorem provers have also been shown to make use of other sources of information. Deepseek-Prover [89] learns from synthetic data. TrialMaster [6] learns from trial-and-error paths. Baldur [26] learns from proof assistant error messages. Passport [74] explores the use of rich identifier information. QEDCartographer [75] uses reinforcement learning to estimate progress toward a complete proof to improve search during proof synthesis. Autoformalization techniques [7], [88] are guided by informal proofs in their construction of formal proofs.

TacticToe [28] employs an A* search. Evariste [44] uses a Monte-Carlo tree search to outperform a best-first search approach in Lean. TacticZero [87] learns proof search strategies, not just tactics, through deep reinforcement learning for HOL4.

Hammers, such as CoqHammer [18] and Sledgehammer [68], are automation techniques used in proof assistants that also perform premise retrieval. They employ SMT solvers, like Z3 [19], and perform efficient automated reasoning to iteratively apply the set of available facts.

In languages like F* that allow for SMT-assisted proof oriented programming, new work has shown promise in using RAG when synthesizing whole programs [14]. Other work uses RAG to generate and summarize Java and Python code [50], [67]. RAG has also been shown to be useful for program repair [61], [85].

Our work has focused on proving properties, which is complementary to specifying them, e.g., by generating formal specification from natural language [22], [30], [55], [56], [94]. Formal languages can be extended to be more expressive, to capture privacy properties [83], data-based properties [59], [60], fairness properties [12], [27], among others. Some of these kinds of properties can be automatically verified probabilistically [4], [29], [33], [53], [81].

VII. Contributions

We developed an adaptive retrieval-augmented proving approach and tool, Rango, to synthesize proofs for formal software verification. Rango improves on prior retrieval-augmented proving approaches by retrieving *proofs* in addition to lemmas. Rango employs its retrieval mechanisms at every step in the proof to obtain the most relevant proofs and lemmas for the current proof state. To train Rango, we collected CoqStoq, a new dataset of Coq proofs, which includes both our training data and a curated benchmark of well-maintained projects. CoqStoq, mined from 2,226 open-source GitHub projects, contains 196,929 theorems, their respective proofs, and 2,225,515 proof steps. Rango proves 32.0% of theorems on CoqStoq's benchmark, which is 29% more than Tactician, a prior state of the art tool. Rango's proof retrieval is important to its success, leading to a 47% increase in the number of theorems proven. Overall, our research shows that retrieval augmentation using in-project proofs in addition to premises is a powerful technique that can increase the proving power of automated proof synthesis tools, reducing the costs of formal verification.

Acknowledgments

This work is supported by the National Science Foundation under grants no. CCF-1955457, CCF-2210243, and CCF-2220892, by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agencies (DARPA) under Contract No. FA8750-24-C-B044, by DARPA under Contract No. HR0011-24-2-0307, and by FCT, Fundação para a Ciência e a Tecnologia under grant BD/04736/2023 and project UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020).

References

[1] Coq-BB5. https://github.com/ccz181078/Coq-BB5, 2024.

[2] PnVRocqLib. https://github.com/PnVDiscord/PnVRocqLib, 2024.

[3] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. Le Goues. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering (TSE)*, 47(10), 2021. doi: 10.1109/TSE.2019.2944914

[4] A. Albarghouthi, L. D'Antoni, S. Drews, and A. Nori. FairSquare: Probabilistic verification for program fairness. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2017.

[5] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 1 ed., 2008.

[6] C. An, Z. Chen, Q. Ye, E. First, L. Peng, J. Zhang, Z. Wang, S. Lerner, and J. Shang. Learn from failure: Fine-tuning LLMs with trial-and-error data for intuitionistic propositional logic proving. *CoRR*, abs/2404.07382, 2024.

[7] Z. Azerbayev, B. Piotrowski, and J. Avigad. ProofNet: A benchmark for autoformalizing and formally proving undergraduate-level mathematics problems. In *Toward Human-Level Mathematical Reasoning*, 2022.

[8] Z. Azerbayev, H. Schoelkopf, K. Paster, M. D. Santos, S. McAleer, A. Q. Jiang, J. Deng, S. Biderman, and S. Welleck. Llemma: An open language model for mathematics. *CoRR*, abs/2310.10631, 2023.

[9] K. Bansal, C. Szegedy, M. Rabe, S. Loos, and V. Toman. Learning to reason in large theories without imitation. *CoRR*, abs/1905.10501, 2019.

[10] L. Blaauwbroek, M. Olšák, J. Rute, F. I. S. Massolo, J. Piepenbrock, and V. Pestun. Graph2Tac: Online representation learning of formal math concepts. In *International Conference on Machine Learning*, 2024.

[11] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: Exploring future states of software. In *Future of Software Engineering Research (FoSER)*, pp. 59–63, 2010. doi: 10.1145/1882362.1882375

[12] Y. Brun and A. Meliou. Software fairness. In *ESEC/FSE NIER Track*, pp. 754–759, 2018. doi: 10.1145/3236024.3264838

[13] P. Carrott, N. Saavedra, K. Thompson, S. Lerner, J. F. Ferreira, and E. First. CoqPyt: Proof navigation in Python in the era of LLMs. In *Foundations of Software Engineering (FSE) Demo Track*, 2024.

[14] S. Chakraborty, G. Ebner, S. Bhat, S. Fakhoury, S. Fatima, S. Lahiri, and N. Swamy. Towards neural synthesis for SMT-assisted proof-oriented programming. *CoRR*, abs/2405.01787, 2024.

[15] J. Chen, H. Lin, X. Han, and L. Sun. Benchmarking large language models in retrieval-augmented generation. In *AAAI Conference on Artificial Intelligence*, vol. 38, pp. 17754–17762, 2024.

[16] A. Chowdhery et al. PaLM: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.

[17] Coq-Community. https://github.com/coq-community, 2024.

[18] Ł. Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1-4):423–453, 2018. doi: 10.1007/s10817-018-9458-4

[19] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2008. doi: 10.1007/978-3-540-78800-3_24

[20] L. de Moura and S. Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, 2021.

[21] H. Eladawy, C. Le Goues, and Y. Brun. Automated program repair, what is it good for? Not absolutely nothing! In *ICSE*, pp. 1017–1029, 2024. doi: 10.1145/3597503.3639095

[22] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM Software Engineering (PACMSE)*, 1(FSE):84:1–84:24, July 2024. doi: 10.1145/3660791

[23] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *CoRR*, abs/2002.08155, 2020.

[24] E. First and Y. Brun. Diversity-driven automated formal verification. In *ICSE*, 2022. doi: 10.1145/3510003.3510138

[25] E. First, Y. Brun, and A. Guha. TacTok: Semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4, 2020. doi: 10.1145/3428299

[26] E. First, M. Rabe, T. Ringer, and Y. Brun. Baldur: Whole-proof generation and repair with large language models. In *ESEC/FSE*, pp. 1229–1241, 2023. doi: 10.1145/3611643.3616243

[27] S. Galhotra, Y. Brun, and A. Meliou. Fairness testing: Testing software for discrimination. In *ESEC/FSE*, pp. 498–510, 2017. doi: 10.1145/3106237.3106277

[28] T. Gauthier, C. Kaliszyk, and J. Urban. TacticToe: Learning to reason with HOL4 tactics. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, vol. 46, pp. 125–143, 2017.

[29] S. Giguere, B. Metevier, Y. Brun, B. C. da Silva, P. S. Thomas, and S. Niekum. Fairness guarantees under demographic shift. In *International Conference on Learning Representations (ICLR)*, April 2022.

[30] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 213–224, July 2016. doi: 10.1145/2931037.2931061

[31] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, et al. DeepSeek-Coder: When the large language model meets programming–the rise of code intelligence. *CoRR*, abs/2401.14196, 2024.

[32] J. M. Han, J. Rute, Y. Wu, E. W. Ayers, and S. Polu. Proof artifact co-training for theorem proving with language models. *CoRR*, 2021.

[33] A. Hoag, J. Kostas, B. C. da Silva, P. Thomas, and Y. Brun. Seldonian toolkit: Building software with safe and fair machine learning. In *ICSE Demo*, 2023. doi: 10.1109/ICSE-Companion58688.2023.00035

[34] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models. *CoRR*, abs/2106.09685, 2021.

[35] D. Huang, P. Dhariwal, D. Song, and I. Sutskever. GamePad: A learning environment for theorem proving. *CoRR*, abs/1806.00608, 2018.

[36] A. Jiang, K. Czechowski, M. Jamnik, P. Milos, S. Tworkowski, W. Li, and Y. T. Wu. Thor: Wielding hammers to integrate language models and automated theorem provers. In *Neural Information Processing Systems (NeurIPS)*. New Orleans, LA, USA, 2022.

[37] A. Q. Jiang, W. Li, J. M. Han, and Y. Wu. LISA: Language models of ISAbelle proofs. In *Conference on Artificial Intelligence and Theorem Proving (AITP)*, pp. 17.1–17.3, September 2021.

[38] A. Q. Jiang, S. Welleck, J. P. Zhou, T. Lacroix, J. Liu, W. Li, M. Jamnik, G. Lample, and Y. Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *The Eleventh International Conference on Learning Representations*, 2023.

[39] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen. Shaping program repair space with existing patches and similar code. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 298–309, July 2018. doi: 10.1145/3213846.3213871

[40] B. Johnson, Y. Brun, and A. Meliou. Causal testing: Understanding defects' root causes. In *ICSE*, 2020. doi: 10.1145/3377811.3380377

[41] V. Karpukhin, B. Oğuz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih. Dense passage retrieval for open-domain question answering. *CoRR*, abs/2004.04906, 2020.

[42] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[43] H. Krasner. The cost of poor software quality in the us: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2, 2021.

[44] G. Lample, T. Lacroix, M.-A. Lachaux, A. Rodriguez, A. Hayat, T. Lavril, G. Ebner, and X. Martinet. Hypertree proof search for neural theorem proving. *Advances in Neural Information Processing Systems*, 35:26337–26349, 2022.

[45] B. Lasse, J. Urban, and H. Geuvers. The Tactician: A seamless, interactive tactic learner and prover for Coq. In *International Conference on Intelligent Computer Mathematics (CICM)*, pp. 271–277, 2020. doi: 10.1007/978-3-030-53518-6_17

[46] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, Nov. 2019. doi: 10.1145/3318162

[47] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 42–54, 2006. doi: 10.1145/1111037.1111042

[48] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM (CACM)*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814

[49] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[50] J. A. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin. Editsum: A retrieve-and-edit framework for source code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 155–166. IEEE, 2021.

[51] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. TBar: Revisiting template-based automated program repair. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 31–42, 2019. doi: 10.1145/3293882.3330577

[52] N. Megill and D. A. Wheeler. *Metamath: a computer language for mathematical proofs*. Lulu. com, 2019.

[53] B. Metevier, S. Giguere, S. Brockman, A. Kobren, Y. Brun, E. Brunskill, and P. S. Thomas. Offline contextual bandits with high probability fairness guarantees. In *Annual Conference on Neural Information Processing Systems (NeurIPS), Advances in Neural Information Processing Systems 32*, pp. 14893–14904, December 2019.

[54] M. Mikuła, S. Antoniak, S. Tworkowski, A. Q. Jiang, J. P. Zhou, C. Szegedy, Łukasz Kuciński, P. Miłoś, and Y. Wu. Magnushammer: A Transformer-based Approach to Premise Selection, 2023.

[55] M. Mirchev, A. Costea, A. K. Singh, and A. Roychoudhury. Assured automatic programming via large language models. *CoRR*, abs/2410.18494, 2024.

[56] M. Motwani and Y. Brun. Automatically generating precise oracles from structured natural language specifications. In *ICSE*, pp. 188–199, May 2019. doi: 10.1109/ICSE.2019.00035

[57] M. Motwani and Y. Brun. Better automatic program repair by using bug reports and tests together. In *ICSE*, pp. 1229–1241, May 2023. doi: 10.1109/ICSE48619.2023.00109

[58] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues. Quality of automated program repair on real-world defects. *IEEE Transactions on Software Engineering (TSE)*, 48(2):637–661, February 2022. doi: 10.1109/TSE.2020.2998785

[59] K. Muşlu, Y. Brun, and A. Meliou. Data debugging with continuous testing. In *ESEC/FSE NI Track*, pp. 631–634, August 2013. doi: 10.1145/2491411.2494580

[60] K. Muşlu, Y. Brun, and A. Meliou. Preventing data errors with continuous testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 373–384, July 2015. doi: 10.1145/2771783.2771792

[61] N. Nashid, M. Sintaha, and A. Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *ICSE*, pp. 2450–2462, 2023.

[62] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002.

[63] D. H. O'Dell. The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, 15(1):71–90, Feb. 2017. doi: 10.1145/3055301.3068754

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

[65] Ott, Myle and Shleifer, Sam and Xu, Min and Goyal, Priya and Duval, Quentin and Caggiano, Vittorio. Fully Sharded Data Parallel: faster AI training with fewer GPUs. https://engineering.fb.com/2021/07/15/open-source/fsdp/, 2021.

[66] A. Paliwal, S. M. Loos, M. N. Rabe, K. Bansal, and C. Szegedy. Graph representations for higher-order logic and theorem proving. In *Conference on Artificial Intelligence (AAAI)*, pp. 2967–2974. AAAI Press, 2020.

[67] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 2719–2734, 2021.

[68] L. Paulson and T. Nipkow. The Sledgehammer: Let automatic theorem provers write your Isabelle scripts! https://isabelle.in.tum.de/website-Isabelle2009-1/sledgehammer.html, 2023.

[69] S. Polu, J. M. Han, K. Zheng, M. Baksys, I. Babuschkin, and I. Sutskever. Formal mathematics statement curriculum learning. In *International Conference on Learning Representations (ICLR)*, 2023.

[70] S. Polu and I. Sutskever. Generative language modeling for automated theorem proving. *CoRR*, 2020.

[71] T. Ringer, A. Sanchez-Stern, D. Grossman, and S. Lerner. REPLica: REPL instrumentation for Coq analysis. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 99–113, 2020. doi: 10.1145/3372885.3373823

[72] S. Robertson, H. Zaragoza, et al. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

[73] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner. Generating correctness proofs with neural networks. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, pp. 1–10, 2020.

[74] A. Sanchez-Stern, E. First, T. Zhou, Z. Kaufman, Y. Brun, and T. Ringer. Passport: Improving automated formal verification using identifiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 45(2):12:1–12:30, June 2023. doi: 10.1145/3593374

[75] A. Sanchez-Stern, A. Varghese, Z. Kaufman, D. Zhang, T. Ringer, and Y. Brun. QEDCartographer: Automating formal verification using reward-free reinforcement learning. In *ICSE*, April 2025.

[76] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *ESEC/FSE*, pp. 532–543, September 2015. doi: 10.1145/2786805.2786825

[77] K. Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[78] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, et al. Gemini: A family of highly capable multimodal models. *CoRR*, abs/2312.11805, 2023.

[79] A. Thakur, Y. Wen, and S. Chaudhuri. A Language-Agent Approach to Formal Theorem-Proving, 2023.

[80] The Coq Development Team. Coq, v.8.7. https://coq.inria.fr, 2017.

[81] P. S. Thomas, B. C. da Silva, A. G. Barto, S. Giguere, Y. Brun, and E. Brunskill. Preventing undesirable behavior of intelligent machines. *Science*, 366(6468):999–1004, 22 November 2019. doi: 10.1126/science.aag3311

[82] H. Touvron et al. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[83] Véronique Cortier, N. Grimm, J. Lallemand, and M. Maffei. A type system for privacy properties. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 409–423. Association for Computing Machinery, 2017. doi: 10.1145/3133956.3133998

[84] H. Wang, H. Xin, C. Zheng, Z. Liu, Q. Cao, Y. Huang, J. Xiong, H. Shi, E. Xie, J. Yin, et al. Lego-prover: Neural theorem proving with growing libraries. In *The Twelfth International Conference on Learning Representations*, 2024.

[85] W. Wang, Y. Wang, S. Joty, and S. C. Hoi. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *ESEC/FSE*, pp. 146–158, 2023.

[86] A. Weiss, A. Guha, and Y. Brun. Tortoise: Interactive system configuration repair. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 625–636, October/November 2017. doi: 10.1109/ASE.2017.8115673

[87] M. Wu, M. Norrish, C. Walder, and A. Dezfouli. TacticZero: Learning to prove theorems from scratch with deep reinforcement learning. *CoRR*, abs/2102.09756, 2021.

[88] Y. Wu, A. Q. Jiang, W. Li, M. Rabe, C. Staats, M. Jamnik, and C. Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

[89] H. Xin, D. Guo, Z. Shao, Z. Ren, Q. Zhu, B. Liu, C. Ruan, W. Li, and X. Liang. DeepSeek-Prover: Advancing theorem proving in llms through large-scale synthetic data. *CoRR*, abs/2405.14333, 2024.

[90] K. Yang and J. Deng. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)*, 2019.

[91] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models, 2023.

[92] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 283–294, 2011. doi: 10.1145/1993498.1993532

[93] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002. doi: 10.1109/32.988498

[94] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang. C2S: Translating natural language comments to formal program specifications. In *ESEC/FSE*, pp. 25–37, 2020. doi: 10.1145/3368089.3409716

[95] C. Zheng, H. Wang, E. Xie, Z. Liu, J. Sun, H. Xin, J. Shen, Z. Li, and Y. Li. Lyra: Orchestrating dual correction in automated theorem proving. *CoRR*, abs/2309.15806, 2023.

[96] Q. Zhu, Z. Sun, Y. an Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE*, pp. 341–353, 2021. doi: 10.1145/3468264.3468544