

NEURAL THEOREM PROVING FOR VERIFICATION CONDITIONS: A REAL-WORLD BENCHMARK

Qiyuan Xu^{1,3}, Xiaokun Luan², Renxi Wang³, Joshua Ong Jun Leang^{4,6}, Peixin Wang⁵, Haonan Li^{3*}, Wenda Li⁶, Conrad Watt¹

¹ Nanyang Technological University ² Peking University ³ MBZUAI
⁴ Imperial College London ⁵ East China Normal University ⁶ University of Edinburgh

ABSTRACT

Theorem proving is fundamental to program verification, where the automated proof of Verification Conditions (VCs) remains a primary bottleneck. Real-world program verification frequently encounters hard VCs that existing Automated Theorem Provers (ATPs) cannot prove, leading to a critical need for extensive manual proofs that burden practical application. While Neural Theorem Proving (NTP) has achieved significant success in mathematical competitions, demonstrating the potential of machine learning approaches to formal reasoning, its application to program verification—particularly VC proving—remains largely unexplored. Despite existing work on annotation synthesis and verification-related theorem proving, no benchmark has specifically targeted this fundamental bottleneck: automated VC proving. This work introduces **Neural Theorem Proving for Verification Conditions (NTP4VC)**, presenting the first real-world multi-language benchmark for this task. From real-world projects such as Linux and Contiki-OS kernel, our benchmark leverages industrial pipelines (Why3 and Frama-C) to generate semantically equivalent test cases across formal languages of Isabelle, Lean, and Rocq. We evaluate large language models (LLMs), both general-purpose and those fine-tuned for theorem proving, on NTP4VC. Results indicate that although LLMs show promise in VC proving, significant challenges remain for program verification, highlighting a large gap and opportunity for future research.

1 INTRODUCTION

Program verification has been fundamental to software reliability for over half a century (Hoare, 1969). While numerous industrial program verifiers have been developed and deployed in history (Cousot et al., 2005), the adoption of program verification remains limited to safety-critical domains (Rushby, 2009; Woodcock et al., 2009). A primary reason is the heavy manual effort required in the theorem proving of *Verification Conditions* (VCs) (Barnett et al., 2006): the logical propositions that encode program correctness.

VC plays a central role in the conventional workflow of program verification (Cohen et al., 2009; Leino, 2010) as shown in Fig. 1: the Verification Condition Generator (VCG) component aims to generate VCs and the prover aims to prove them. Conventionally, VC proving is carried out by Automated Theorem Provers (ATPs). However, ATPs excel only at specific domains of problems, and require human intervention (e.g., manual proofs and annotations) when automatic proof attempts fail or time out. Taking the widely-used industry tool Frama-C (Baudin et al., 2021) as an example, existing ATPs’ insufficient capability necessitates \sim 600 lines of annotations for a linked list library, nearly matching the original C code length. Consequently, due to the central role of VC proving and the inadequacy of current automated approaches, VC proving has become a key bottleneck in automated program verification.

Large language models (LLMs) have opened the door to Neural Theorem Proving (NTP) (Minervini et al., 2018), where models generate formal proofs to conduct theorem proving. While existing NTP research has focused primarily on mathematical domains, proving competition problems (Zheng

*Corresponding Author

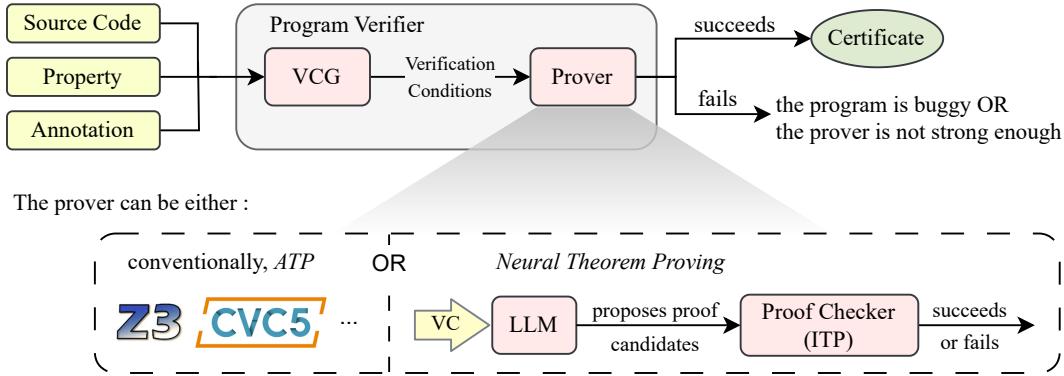


Figure 1: The conventional and NTP-based workflow of program verification.

et al., 2022; Tsoukalas et al., 2024) and formalizing mathematics (Xin et al., 2025), theorem proving extends naturally to VC proving (Harrison et al., 2014).

This motivates our central question: *can NTP automate VC proving?* To answer this, we introduce **Neural Theorem Proving for Verification Conditions (NTP4VC)** — a task that applies machine-learning-based proof generation to conduct the theorem proving of VCs.

To evaluate this task, we construct the first benchmark for NTP4VC, whose major features are compared with prior works in Tab. 1. A challenge of this construction is that Lean (de Moura & Ullrich, 2021), a mainstream language in the NTP community, has relatively fewer mature program verification frameworks built on top of it and large-scale industrial verification projects using it. Despite our best efforts, we find no sufficient native VCs available in Lean for a NTP4VC benchmark.

We overcome this issue by translating the VCs generated from other industrial verification pipelines (Why3 (Filliatre & Paskevich, 2013) and Frama-C (Baudin et al., 2021)) into Lean. This approach also allows us to translate VCs to Isabelle (Paulson, 1990), Rocq (Coquand & Huet, 1988) (which are already implemented), and potentially other target languages, forming the first multi-language benchmark in NTP-based program verification. More crucially, this approach further allows extracting VCs from existing verification projects for industrial software, such as the Linux kernel’s scheduler (Lawall et al., 2025), library functions Efremov et al. (2018), and Contiki OS’s memory allocator (Mangano et al., 2017) and linked-list library (Blanchard et al., 2018).

Unlike LLM-based translation approaches that suffer from LLMs’ unreliability, our translation pipeline is based on ~ 800 expert-written translation rules for each of the three target languages (so $\sim 3 \times 800$ in total). These rules are explicitly chosen to ensure semantic preservation from the origins to the translations, thereby better ensuring the quality of the benchmark cases compared to LLM-based translation approaches.

We further evaluate several existing provers and LLMs on NTP4VC. For language-specific fine-tuned provers, the best model achieves only 2.08% pass@1, while general-purpose LLMs achieve lower performance, with GPT-o4-mini-high achieving 1.19% pass@1. These results highlight the substantial difficulty of VC proving and the need for progress in NTP and LLM reasoning.

To summarize, our contribution includes:

1. We define the task of NTP4VC (§ 1), which aims to attack the automated proving of VC, a key bottleneck in program verification.
2. We propose a *reliably automatic* method for extracting corpora from real-world verification projects (§ 3). The implementation is open-sourced.
3. We present the first real-world, multi-language benchmark for NTP4VC, with open-sourced implementation and extensive evaluation of existing provers and LLMs (§ 5).

Table 1: Comparison between our benchmark and previous ITP-based benchmarks for program verification. **VC**: the proportion of VC test cases. **Industrial pipeline**: whether the work uses industrial program verification pipelines. **Language**: the proof language supported by the benchmark.

Benchmarks	Focus	VC	Industrial Pipeline	Language		
				Lean	Isabelle	Rocq
Lin et al. (2024)	verification-related lemmas	< 17%	✓	✗	✓	✗
Thompson et al. (2025)		< 20%	✓	✗	✗	✓
Thakur et al. (2025)		0%	✗	✓	✗	✗
Dougherty & Mehta (2025)	programming puzzles in Lean	0%	✗	✓	✗	✗
Lohn & Welleck (2024a)		0%	✗	✓	✗	✗
Ours	VCs from puzzles & industrial projects	100%	✓	✓	✓	✓

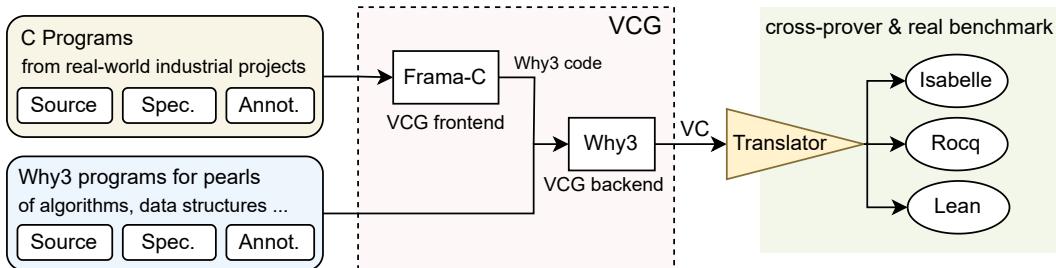


Figure 2: Our pipeline for extracting benchmark cases.

2 BACKGROUND

Theorem proving falls broadly into two categories: **Automated Theorem Proving (ATP)** and **Interactive Theorem Proving (ITP)**. ATP achieves full automation within specific domains of proof problems. These domains are limited, and VCs in real-world verification projects often exceed these domains, leading to proof failures and inevitable human intervention (e.g., manual proofs and annotations) in order to complete the proofs. By contrast, ITP provide highly expressive languages that enable users to construct proofs across broad domains, capable of handling almost all program verification problems. Mainstream ITP languages include Isabelle, Rocq, and Lean.

Program verification aims to verify that a program satisfies a given property. Ideally, a strong enough verifier should be able to complete the verification solely given the source code and the property. In practice, however, due to limitations in both VCG and VC provers, users often have to provide manual proofs and annotations to guide the verifier in completing the verification. The manual effort for these proofs and annotations constitutes a huge cost burden in program verification.

Why3 and **Frama-C** are famous program verifiers widely used in the industry. Why3 provides 1) a language for both programming, annotation, and specifying functional correctness, 2) a VCG, and 3) powerful ATPs. A limitation is that Why3 can only verify programs written in its abstract specification language. In order to verify programs written in industrial languages, Why3 is widely used as the verification backend of well-known toolchains which translate their input language to the Why3 specification language — including Frama-C, Cameleer (Pereira & Ravara, 2021), Creusot (Denis et al., 2022), and EasyCrypt (Barthe et al., 2011). Frama-C is an industrial verifier for the C language. It provides a frontend to process C source code and then calls Why3 to complete the verification. The input of Frama-C is C source code with properties and annotations provided as comments, and the output is Why3 code that Why3 can continue to verify. Finally, Frama-C is widely used, having verified enormous industrial programs, such as air traffic management algorithm (Dutle et al., 2021), embedded operating system (Mangano et al., 2017; Blanchard et al., 2018), cryptographic modules (Peyrard et al., 2018), Linux kernel scheduler (Lawall et al., 2025), and JavaCard virtual machine (Djoudi et al., 2021).

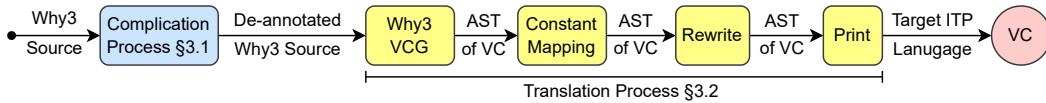


Figure 3: The generation process of the benchmark cases and potentially training corpora.

3 A RELIABLE AND AUTOMATIC METHOD FOR CORPORA GENERATION

This section presents the method we use to extract real-world VCs that constitute our benchmark. The key idea is to reuse existing industrial VCGs to extract VCs from existing verification projects, and translate these VCs into the language of the target ITPs (§ 3.1). Since the projects have all passed the verifiers’ checks, the VCs are guaranteed to be provable. However, this also makes them too easy to serve as valuable benchmark cases, as they may already be within reach of existing ATPs. To produce challenging benchmark cases, we introduce a novel complication process (§ 3.2) to make VCs harder while keeping them provable. The complete process is illustrated in Fig. 3.

3.1 VC EXTRACTION & RULE-BASED TRANSLATION

Various VC languages are used in industry, such as Why3, TPTP (Sutcliffe, 2024), and SMT Lib (Barrett et al., 2010). We adopt Why3 because its logic system is Simple Typed Theory (Church, 1940), a relatively high-level system that is close to and entailed by the logic of mainstream ITPs like Lean, Isabelle, and Rocq, ensuring the feasibility of the translation.

The translation process begins with a given Why3 source code. It first runs Why3 VCG to generate VCs and calls our customized Why3 printer to dump the VCs into an XML representation of their Abstract Syntax Trees (ASTs). These ASTs are processed by a Python translation framework also written by us and finally mapped into the target ITPs' languages. The details are provided in appendix E.

While the above process enables the basic translation from Why3 to target ITP languages, our work goes beyond this to strive for idiomatic translations that closely approximate native expressions on the target ITP platforms. For this, our translation process incorporates enhancements from two aspects: First, at the syntactic level, we use printing rules to map specific term structures to their corresponding pretty syntax defined in the ITP, including prefix, infix annotations, and ad-hoc syntax sugars like if-then-else, match-case, and list[index]. Second, we build a rewriting system to rewrite specific combinations of terms into more idiomatic expressions. Examples include rewriting integer operations into natural number operations that are more common in ITP.

The implementation of the pipeline is made of more than 2400 mapping & rewriting rules written by human experts in ITP, in total for Isabelle, Lean, and Rocq. The correctness of the rules is supported by syntax checking over the translation results on one hand, and cross-validation by other experts (our first, second, and last authors) on the other hand. These expert-written rules form the foundation of the translations' correctness and quality. Once this foundation is built, the entire translation process is automatic, constituting a *reliably automatic* method for extracting VC corpora.

3.2 COMPLICATION PROCESS: EXTRACTING CHALLENGING VCS

As mentioned at the beginning of this section, the VCs extracted from real-world projects are already provable by existing ATPs, thus providing insufficient challenge for benchmark evaluation. However, these VCs are provable by the ATPs as human developers have already written sufficient annotations to make them easy for ATPs to prove, rather than from inherent ATP strength. A direct idea is to erase these auxiliary annotations and restore the verification tasks to what they should ideally be in fully automated program verification.

Specifically, three sorts of annotations are dedicated to VC simplification: (1) `assert` annotation, which introduces a subgoal to ask the prover to first prove this subgoal and then use the proven subgoal as a lemma in the subsequent proofs; (2) `lemma` annotation, which explicitly introduces a global lemma so that the prover can later reference it to prove subsequent propositions; (3) annotation of lemma application, which explicitly instantiates (the free variables in) a lemma and advises

the prover to use it. All these annotations can be safely erased without affecting the VC’s provability (by a strong enough prover) (Bobot et al., 2025a; Correnson et al., 2025). In addition, they exhibit clear syntactic patterns enabling us to identify and erase them. Indeed, the exact job of our compilation process is erasing the annotations. The results show this process effectively reduces the pass rate of Why3’s strongest ATP from ~99% to ~62% on Why3’s bundled examples.

4 NTP4VC BENCHMARK

The method discussed in § 3 enables effective extraction of real-world VCs from existing verification projects. By applying the method, we extract >7.5k VCs from various sources. From there, we carefully select 600 VCs to constitute a benchmark, with consideration for breadth, diversity, and the balance of difficulty levels as described below.

Breadth, Diversity, and the Difficulty Level. Real-world industrial projects certainly possess high value in a verification benchmark like ours, while at the same time, challenging algorithms and data structures are equally valuable verification targets due to their complexity. An issue is that a conflict exists between them: challenging algorithmic content is sparse in industrial project source code. If a benchmark focused solely on VCs from industrial projects, it would underrepresent algorithms and data structures. In order to balance the breadth of the verification scenarios involved, we divide our benchmark into two equal parts (50% vs 50%). (1) **Pearls of Programs** consists of minimal working programs that capture verification pain points, including algorithms, data structures, and well-known “hard nuts to crack”, such as Binomial Heap, VerifyThis’24 competition, and Hillel challenge (Wayne, 2018). These programs are written in Why3’s abstract specification language. (2) **Real C Verification:** VCs from industrial C programs used in real-world projects, such as the memory allocator (Mangano et al., 2017) and the linked-list library (Blanchard et al., 2018) from the Contiki Operating System.

Each category is further divided into sub-categories (Tab. 2), with roughly balanced numbers of cases in each sub-category to maintain diversity. The pearl of programs consists of 1) Well-known algorithms such as sorting, string operations, searching, shortest path, and graph; 2) Data structures, including (balanced) trees, heaps, hash, and arrays; 3) Numerical and other calculations, such as arbitrary precision arithmetics, square root, exponentiation by squaring, and bitwise operations; 4) Engineering optimization tricks (e.g., in-place reversal of linked lists and N-queens by bitvector) and common engineering tasks (e.g., string padding, list element removal, space-insensitive comparison between strings, and the challenges by Wayne (2018)); 5) Cases from well-known verification competitions, e.g., VerifyThis (Ernst et al., 2019) and VSCOMP (Klebanov et al., 2011).

While the pearl of programs is organized by source programs’ functionality, cases in the real C verification are categorized by the properties that VCs validate: 1) Function category verifies that programs’ logical behavior meets the desired functionalities from a big-picture view, assuming the absence of runtime errors; 2) Loop category verifies loop termination, and loop invariants are established and maintained; 3) Memory category rules out the runtime error of invalid memory access; 4) Invalid Arg. checks that arguments and operands are valid. For example, the operands of multiplication do not cause arithmetic overflow, and the dividend is not zero.

Beyond breadth, we design the benchmark to balance difficulty across categories. We measure difficulty using the pass rate of Why3’s strongest predefined ATP tactic, `Auto_Level_3` (AL3) (Bobot et al., 2025b). AL3 is a hybrid tactic that combines sophisticated heuristics and five industrial cutting-edge ATPs, Z3 (de Moura & Bjørner, 2008), CVC4 (Barrett et al., 2011), SPASS (Weidenbach et al., 2009), Alt-Ergo (Conchon et al., 2018), and E-prover (Schulz, 2002), such that a goal is proved once any of the ATPs proves the goal. The pass rate of AL3 then indicates the state-of-the-art of Why3 ATP over the benchmark cases, denoted as *ATP pass@n* in Tab. 2. A VC is deemed *hard* if AL3 fails to prove it, making its solution an open problem. Our design goal is to set each

Table 2: Categories of the benchmark

Category	Number	ATP pass
<i>pearls of programs</i>		
Algorithm	55	20.00%
Data Structure	73	19.18%
Calculation	66	19.70%
Engineering	54	20.37%
Competition	52	19.23%
<i>real C verification</i>		
Function	81	23.46%
Loop	81	24.69%
Memory	74	24.32%
Invalid Arg.	64	25.00%
Total	600	22.00%

Table 3: Sources of cases in real C verification. LoC = Lines of C Code (comments are excluded).

Project	# of VCs	LoC	License
Linked List Library in Contiki OS (Blanchard et al., 2018)	78	833	BSD-3-Clause
Memory Allocator in Contiki OS (Blanchard et al., 2018)	21	145	BSD-3-Clause
Selected Cases from C++ STL (Burghardt et al., 2015)	53	3263	MIT
X.509 Parser (Ebalard et al., 2019)	9	5044	BSD-3-Clause
Linux Kernel Scheduler’s SWB Routine (Lawall et al., 2025)	24	216	GPLv2
Linux Kernel Library Functions (Efremov et al., 2018)	85	3533	GPLv3
String & Stdio Library in kLIBC (Carvalho et al., 2014)	14	1220	GPLv2
Paparazzi UAV Autopilot’s Math Lib (Pollien et al., 2021)	16	3159	GPLv2
Total	300	17413	-

Table 4: Statistics of involved operations. Format: *average* (25^{th} – 75^{th} percentile)

Operations	# of cases	# of operations	# of distinct oprs	Size	Depth	# of $\forall \exists$
Integer Arith	578	69.5 (10 – 73.5)	5.0 (4 – 6)	600.0 (133 – 643)	55.8 (28 – 74)	11.0 (1 – 13)
Non-Linear Arith	118	10.7 (2 – 14)	1.2 (1 – 1)	1191 (228 – 1370)	80.3 (42.5 – 115)	15.2 (1 – 20)
Float Arith	18	48.9 (38 – 61)	7.1 (6 – 7)	337 (209 – 544)	73.8 (40.5 – 105.5)	0.11 (0 – 0)
List, Sequence	202	46.7 (8 – 62)	4.1 (2 – 6)	963 (198 – 1161)	54.8 (24 – 69)	18.6 (5 – 23)
Set, Map, Bag	59	50.3 (10 – 55)	3.8 (1 – 6)	904 (321.5 – 1134)	45.3 (28 – 55.5)	26.7 (8 – 40)
Tree, String, Matrix	26	64.5 (14 – 95)	5.0 (4 – 6)	822 (204 – 1062)	45.9 (25.5 – 62)	34.4 (6 – 42)
Memory	297	29.5 (13 – 31)	7.9 (6 – 10)	357 (129 – 452)	62.3 (37 – 80)	5.2 (0 – 6)
Custom Datatype	247	87.7 (7.5 – 94)	7.0 (3 – 10)	864 (193.5 – 1038.5)	61.1 (25 – 103)	16.3 (3 – 22)
All	600	295.1 (71 – 339)	24.8 (21 – 29)	583 (129 – 632)	54.5 (27 – 72)	10.8 (1 – 12)

category’s composition to target an AL3 pass rate between 20%–25%, a level that ensures sufficient open problems for advancing NTP while still allowing effective evaluation of existing approaches.

Diversity of VC Expressions While the previous subsection measures the diversity of the source and the purpose of the VCs, this subsection discusses the arithmetic and data structure operations involved in these VCs. We follow the taxonomic methodology conventionally used in the ATP field (Barrett et al., 2010; SMT-LIB Initiative), and categorize the operations according to the notions and the data types involved in their related reasoning. As listed in Tab. 4, the categories include integer arithmetics, non-linear arithmetic, and various common data structures. Some cases may define their custom datatypes beyond those provided in the standard libraries. This is captured by the *Custom datatype* category. Further details about this classification are given in appendix H.

For each of the categories, we count the benchmark cases that involve at least one such operation, and report the average, 25^{th} , and 75^{th} percentile of: # of operations, the total number of occurrences of these operations; # of unique oprs, the number of distinct operation types in each case in each case; size, the number of atomic terms; depth, the height of the abstract syntax tree of the VCs; # of $\forall \exists$, the number of quantifiers occurring in each case. As presented in Tab. 4, the result shows our benchmark cases exhibit a wide distribution across different data structures and arithmetic operations, and also span VCs of varying scales within each category.

Sources of the Benchmark Cases, and Their Licenses. All the VCs in the benchmark are drawn from open-sourced verification projects. The pearls of programs come from the Gallery of Verified Programs Toccata Team (2025), released under the LGPL v2.1 license alongside Why3’s source code. For real C verification, VCs are collected from multiple projects, as summarized in Tab. 3.

VC Selection Process. The 600 benchmark cases are selected from over 7.5k VCs. This subsection elaborates on the selection process. The process consists of three rounds: The first round determines the domain from which the benchmark cases will be selected; in the second round, one expert performs an initial screening to identify $\sim 1.5\text{k}$ candidate cases; three experts then collaboratively evaluate each candidate in the final round to finalize the benchmark set of 600 cases.

Recall that cases in Pearls of Programs are sourced from the Toccata Team (2025)’s collection of 224 individual projects. In the first round, we select 100 projects from which all Pearl benchmark cases will be drawn, leaving the remaining 124 projects for potential use as training data. To collect

as many hard VCs as possible, we prioritize selecting projects rich in hard VCs. To measure if a VC is hard, we run Why3’s AL3, and it is hard if and only if AL3 fails to solve it in 10 minutes on a 12-core workstation. For Real C Verification, we do not maintain such project-level separation, and the 8 projects are all used for benchmark cases.

In the second round, we first select all the hard VCs from the domain, totaling $\sim 1.2k$ cases. Since we aim for the benchmark to have a 20–25% pass rate on the ATP baseline, we correspondingly select ~ 350 cases from the easy VCs to balance the candidate set at this stage. When selecting each easy VC, we check whether it is trivially provable (e.g., true \wedge true). To do this, we examine the VC’s logical expression, the source program, related annotations, and the specifications to check that the property verified by the VC is meaningful and commonly encountered in program verification tasks.

In the final round, we apply the same evaluation method above to assess each case and refine the candidate set while additionally considering balanced coverage across the categories shown in Tab. 2. We also consider broad project coverage by selecting cases from different projects proportionally.

Format of the Benchmark Cases. Each benchmark case is a single VC (a single proof goal) placed individually in a theory file, and each such file contains exactly one VC. Every VC originates from a verification project and thus may contain project-specific concepts (e.g., the data type of binary tree), resulting in VCs with library dependencies. Consequently, this requires benchmark participants to be able to learn new concepts on-the-fly from the verification projects’ dependency libraries.

Dataset Contamination. Our benchmark is generally free of data contamination concerns, despite all the source programs, properties, and annotations are public. This is because: (1) The transformation from program and property source code to VCs is complex. Even if LLMs were trained on the original source code, they cannot trivially generate VC-level concepts. In typical program verification workflows, VCs are generated only transiently and are not persistently stored or published unless done deliberately. (2) The VCs we use are derived from Why3 source code after a complication process, making most of them unprovable by existing ATPs, proofs for these VCs have never existed. (3) Even if we assume the proof details of the VCs from the original Why3 source code can leak information about the proofs of the complicated Why3 code, no leakage of the proof details is discovered despite our best efforts. This is expected, since Why3 never stores detailed proofs, but only records the ATP tools used, replaying them when proofs are needed. In fact, many ATPs do not support dumping detailed proofs at all. In summary, the risk of meaningful data contamination in our benchmark is extremely low.

Caveat: Provability of Benchmark Cases. All benchmark cases are drawn (after the complication process) from verification conditions in the projects that are formally verified. Methodologically, this should guarantee the provability of all benchmark cases. However, in practice, all VCs are processed through Frama-C, Why3, and our translation pipelines. Implementation bugs in any of these components may potentially render some benchmark cases unprovable. During our selection process, we checked the provability of each case to the extent possible. However, due to the inherent complexity of theorem proving, we cannot guarantee the actual provability of every case. Therefore, we must include this caveat in the paper regarding the provability of benchmark cases.

5 EXPERIMENTS AND EVALUATION

To evaluate the challenges posed by NTP4VC, we assess seven models, covering both general-purpose language models such as GPT-4o-mini (Achiam et al., 2024) and specialized models like DeepSeek-Prover-V2 (Ren et al., 2025). We also include ITP hammers to provide a baseline for comparison, including the hammers: Sledgehammer (Böhme & Nipkow, 2010) tool in Isabelle/HOL and CoqHammer (Czajka & Kaliszyk, 2018) in Rocq.

Models We evaluate both proprietary models (GPT-o4-mini (Achiam et al., 2024)) and open-source models (K2-Think (Cheng et al., 2025), DeepSeek-V3.1 (Liu et al., 2024), Qwen3 (Yang et al., 2025), DeepSeek-Prover-V2 (Ren et al., 2025), Goedel-Prover (Lin et al., 2025), Minilang (Xu et al., 2025)). Among them, DeepSeek-Prover-V2 and Goedel-Prover are specialized for theorem proving using Lean, while others are general-purpose reasoning models. We use 1.0 as the default temperature, and set the maximum number of tokens to 32,000 during generation.

Table 5: Pass rates (Pass@1, Pass@4, Pass@8) of various NTP models and hammer-based automated theorem provers on the NTP4VC benchmark, evaluated across Lean, Rocq, and Isabelle. NTP models consistently achieve pass rates below 4%, while hammer-based provers such as CoqHammer and Sledgehammer obtain higher success rates, particularly on Rocq (5.67%) and Isabelle (18.00%).

Model	Lean			Rocq			Isabelle		
	P@1	P@4	P@8	P@1	P@4	P@8	P@1	P@4	P@8
GPT-04-mini-high	0.50	–	–	0.00	–	–	1.17	–	–
DeepSeek-V3.1	0.50	1.00	1.67	0.50	2.67	3.17	1.34	4.32	6.25
Qwen3-32B	0.33	0.83	1.17	0.33	1.17	1.67	0.74	2.53	3.42
Qwen3-235B-A22B	0.67	1.00	1.00	0.83	2.17	3.33	1.19	2.08	3.13
K2-think	0.00	0.00	0.00	0.67	0.67	0.67	0.00	0.00	0.00
Goedel-Prover-V2-32B	0.50	1.17	2.17	–	–	–	–	–	–
DeepSeek-Prover-V2-671B	1.67	2.17	3.00	–	–	–	–	–	–
Minilang	–	–	–	–	–	–	2.08	7.29	11.46
CoqHammer / Sledgehammer	–	–	–	5.67	–	–	18.00	–	–

Metrics Our primary evaluation metric is the *pass@n* metric. NTP models are queried multiple times for each problem, generating multiple proof attempts. A proof attempt is considered successful if it can be verified by the corresponding ITP and does not contain any fake proofs such as `admit` or `sorry`. Since hammers are mostly deterministic, we only report their *pass@1* performance. GPT-04-mini is evaluated with a single attempt per problem due to its cost, while other models are evaluated with 8 attempts per problem ($n = 8$).

Prompts We use zero-shot prompting for all models, providing only the problem statement and the necessary context such as definitions and previously proved lemmas. The full prompt structures are provided in Appendix F.

Proof Verification Our proof verification setup involves extracting the proof from the model’s output and checking it within the corresponding ITP environment. We use the Lean 4.21.0, Rocq 8.20.1, and Isabelle 2024. To prevent excessively long runtimes, we set a timeout of 10 minutes for each verification attempt. Sledgehammer in Isabelle is configured to use its default ATPs and SMT solvers, including CVC4 (Barrett et al., 2011), CVC5 (Barbosa et al., 2022), Z3 (de Moura & Bjørner, 2008), E (Schulz, 2002), SPASS (Weidenbach et al., 2009), Vampire (Kovács & Voronkov, 2013), veriT (Schurr et al., 2021), and Zipperposition (Vukmirović et al., 2022). CoqHammer is configured to use all its supported ATPs, including E, Vampire, Z3, and CVC4. All proof verification is performed on a machine with an AMD Ryzen 9 7900X CPU and 64GB RAM.

5.1 RESULTS

The results summarized in Tab. 5 highlight the difficulty of program verification for NTP models. Across all three ITPs, our experiments demonstrate that all NTP models fail to achieve *pass@8* scores above 12% when evaluated on Lean, Rocq, and Isabelle. This stands in sharp contrast to their strong performance on mathematics benchmarks. For example, DeepSeek-Prover-V2 achieves 55.5% *pass@1* on miniF2F, while Goedel-Prover-V2-32B achieves 88.1% *pass@32*. On a more challenging baseline such as PutnamBench, these models obtain 7.15% and 13.09% *pass* rates, respectively, with various attempts. This performance gap suggests that program verification requires fundamental

Table 6: Number of problems solved and corresponding pass rates of NTP models and hammer-based provers on the NTP4VC benchmark, broken down by problem category.

Category	NTP Models		Hammers	
	Pass / Total	Pass Rate	Pass / Total	Pass Rate
Algorithm	2 / 55	3.64%	4 / 55	7.27%
Data Structure	3 / 73	4.11%	10 / 73	13.70%
Calculation	5 / 66	7.58%	8 / 66	12.12%
Engineering	5 / 54	9.26%	7 / 54	12.96%
Competition	3 / 52	5.77%	3 / 52	5.77%
<i>Pearls of Prog.</i>	15 / 300	5.00%	32 / 300	10.67%
Function	6 / 81	7.41%	25 / 81	30.86%
Loop	3 / 81	3.70%	19 / 81	23.46%
Memory	2 / 74	2.70%	18 / 74	24.32%
Invalid Arg.	5 / 64	7.81%	20 / 64	31.25%
<i>Real C Verif.</i>	16 / 300	5.33%	82 / 300	27.33%
Total	34 / 600	5.67%	114 / 600	19.00%

```

lemma decompose_front_node'vc: removing the first element in an AVL tree is correctly implemented
proof
fix d2 res
assume pre: "case o1 of AEmpty => d2 = d & res = r
| ANode l1 d21 r2 h s => ∃res1. node_model (seq (m1 l1)) d21 (seq (m1 r2)) = Cons d2 (seq (m1 res1)) ∧
(0 ≤ (1 + (if hgt (m1 l1) < hgt (m1 r2) then hgt (m1 r2) else hgt (m1 l1))) - hgt (m1 res1)) ∧
(1 + (if hgt (m1 l1) < hgt (m1 r2) then hgt (m1 r2) else hgt (m1 l1))) - hgt (m1 res1) ≤ 1) ∧
...
(-int balancing ≤ hgt (m1 res1) - hgt (m1 r) ∧ hgt (m1 res1) - hgt (m1 r) ≤ int balancing →
(1 + (if hgt (m1 res1) < hgt (m1 r) then hgt (m1 r) else hgt (m1 res1))) = hgt (m1 res))"

```

Missing parenthesis ()

Redundant parenthesis))

Figure 4: An Isabelle proof generated by DeepSeek-V3.1 for a VC in the benchmark. This proof contains syntax errors, including a missing closing parenthesis and two redundant closing parentheses. The `seq` returns a tree’s elements as a sequence in order; the `hgt` gives a tree’s height; `balancing` is the balancing factor of AVL tree. The full example is provided in Appendix G.

tally different reasoning capabilities than complex mathematical benchmarks.

By comparison, hammer-based provers show stronger results on NTP4VC: Sledgehammer reaches a pass rate of 18.00%, outperforming all the language models. By combining Sledgehammer, Minilang’s language model achieves 11.56% pass rate, outperforming all other models. These results indicate that current state-of-the-art NTP models have yet to surpass the classical reasoning techniques in VC proving. Notably, although Minilang’s model already incorporates Sledgehammer, it still fails to surpass Sledgehammer alone on NTP4VC. This further confirms that NTP4VC represents a novel domain for the model, distinct from the mathematical problems it excels at.

To better understand this performance difference across problem types, we report the number of problems solved per category by NTP models and hammers in Table 6. The result confirms that hammers consistently outperform NTP models in almost all categories. However, the magnitude of this outperformance is non-uniform: the advantage of hammers over NTP models is substantially larger in Real C Verification tasks than in Pearls of Programming. We consider this unsurprising, particularly noting that NTP models perform comparably across both categories, but the hammers exhibit markedly stronger performance on Real C Verification. This is likely because the classical reasoning techniques underlying hammers, such as SMT solvers and ATP provers, have been optimized for industrial verification scenarios, whereas they are less suited to the logic puzzles in Pearls of Programming, which involve a substantially broader search space.

5.2 ERROR ANALYSIS OF NTP MODELS

To understand the limitations of current NTPs on verification tasks, our qualitative analysis of failure cases reveals three recurring themes: syntactic errors, semantic confusion, and hallucination. More details are available in Appendix G.

Syntactic Errors A primary hurdle for NTPs is generating syntactically correct terms. For instance, a proof for an AVL tree VC (see Fig. 4) failed to parse due to mismatched parentheses. Correcting these purely syntactic errors allowed the term to be successfully parsed. More than 24% of generated Isabelle proofs contain syntactic errors. This highlights a key challenge of VCs: unlike typical math problems that prioritize semantic insight, VCs are often long, deeply-nested, machine-generated formulas. Their structure places extreme demands on a model’s ability to maintain long-range syntactic coherence.

Semantic and Pragmatic Confusion A more profound failure is the model’s misunderstanding of the proof paradigm itself. This is common in Lean, where models produce syntactically plausible but pragmatically incorrect code, leading to type errors. For example, they often use imperative-style assignments (e.g.,`:= i1 + i2`) instead of declarative, tactic-based reasoning. This confusion is further evidenced by proof scripts degenerating into repetitive and meaningless tactic applications (e.g., “`have h16 := h0; have h17 := h1 ...`”), which occurs in more than 64% of Lean proofs generated by Goedel-Prover-V2-32B, one of the state-of-the-art NTP models. Even powerful mod-

els like DeepSeek-Prover-V2 exhibit this behavior, suggesting they become overwhelmed by VC complexity and resort to semantically inappropriate code, fundamentally misinterpreting the task.

Hallucination of Non-Existent Entities Finally, models frequently hallucinate non-existent constants, lemmas, or tactics. For instance, GPT-04-mini often invokes a tactic called `why3`, which does not exist in Rocq, as a standalone proof for an entire VC. Similarly, many models introduce undefined constants or lemmas not found in the context or standard libraries. At least 9% of proof attempts in Isabelle failed due to these undefined entities. This demonstrates a failure to ground the generation process within the strict formal context provided by the prover.

6 RELATED WORKS

Prior benchmarks by [Mugnier et al. \(2025\)](#); [Loughridge et al. \(2025\)](#); [Sun et al. \(2024\)](#); [Yang et al. \(2024\)](#); [Zhong et al. \(2025\)](#) consider the **synthesis of annotations**: given source programs and properties, the task is to generate annotations that enable program verifiers to succeed. Like our work, they operate directly with industrial verifiers (e.g., Dafny ([Leino, 2010](#)), Verus ([Lattuada et al., 2023](#))). Besides, they tackle the end-to-end automation problem, which offers direct practical value by reducing the manual annotation burden. However, as mentioned in § 2, an ideal verifier should not require annotations in the first place, and a stronger VC prover brings us closer to this ideal verifier. In terms of automated program verification, our NTP4VC task is complementary to annotation synthesis approaches — we propose to tackle the VC proving bottleneck directly, while they approach the problem indirectly through annotation generation (e.g., generating `assert` annotations that decompose hard VCs into simpler subgoals such that the provers can handle). Both of them are effective ways to improve automation in program verification and can be applied orthogonally.

There are also NTP benchmarks ([Lin et al., 2024](#); [Thompson et al., 2025](#)) discussing **verification-related theorem proving**, typically consisting of proof goals collected from ITP projects about program verification engines and their applications. However, much of their work focuses on auxiliary lemmas used by program verifiers and specifications — such as those for preliminaries (e.g., arithmetic of bounded integers), programming language models (e.g., memory models), and abstract program models (e.g., binary tree algebra) — rather than VCs. In detail, no more than 17% of the test cases by [Lin et al. \(2024\)](#) might be VCs, and no more than 20% for [Thompson et al. \(2025\)](#)'s work (see appendix J for details). The gap between auxiliary lemmas and VCs is crucial because VCs are the direct theorem-proving targets that arise from program verification workflow (§ 2), while auxiliary lemmas cannot (completely) represent the theorem-proving tasks in program verification.

Besides, the Lean benchmarks by [Thakur et al. \(2025\)](#); [Dougherty & Mehta \(2025\)](#); [Lohn & Welleck \(2024a\)](#) are also designed for program verification. These works suffer from a limitation — they do not follow the mainstream program verification methodologies adopted in the real-world industry. Lean is a specialized language with integrated verification capabilities, where the programming language itself serves as a logical reasoning language. This enables users to write Lean programs and directly verify them using the Lean system, without requiring a separate VCG for program analysis. However, program verification tasks in the real-world industry typically have to face industrial programming languages that differ substantially from logical reasoning languages. Typical industrial programming languages feature complex constructs such as mutable references, memory models, functions with side effects, and pointer arithmetics — none of which are involved in the program verification tasks examined by these benchmarks. This contrast further underscores the necessity of employing industrial verification pipelines to extract VCs from real-world industrial projects for benchmark construction.

Finally, we also want to mention other NTP benchmarks involving much wider domains in theorem proving, such as the works by [Yang & Deng \(2019\)](#); [Li et al. \(2021\)](#); [Lohn & Welleck \(2024b\)](#); [Yang et al. \(2023\)](#); [Gauthier et al. \(2021\)](#); [Kaliszyk et al. \(2017\)](#); [Bansal et al. \(2019\)](#); [Huang et al. \(2019\)](#), which are also important benchmarks in NTP.

7 CONCLUSION

This work introduces Neural Theorem Proving for Verification Conditions (NTP4VC), presenting the first real-world multi-language benchmark for automated VC proving — a critical bottleneck in

program verification. Alongside this benchmark, this work develops a reliable extraction method using expert-written translation rules and industrial verification pipelines (Why3 and Frama-C) to extract VC corpora from real-world verification projects and generate semantically equivalent VCs across Isabelle, Lean, and Rocq. Our evaluation of 600 carefully selected VCs from industrial projects reveals the substantial difficulty of this task: the strongest neural theorem provers achieve only 2.08% pass@1. Our error analysis reveals that the lengthy, deeply nested structure of VCs presents fundamentally different challenges to NTP models compared to mathematics competition problems. The benchmark and the corpora extraction method establish a foundation for advancing neural approaches to program verification, with the potential to achieve significant breakthroughs in automated program verification.

REPRODUCIBILITY STATEMENT

We have made a comprehensive artifact to ensure the reproducibility of our results and to encourage future research. The artifact contains the complete NTP4VC benchmark, the source code for our VC extraction tool, and all scripts required to replicate our experiments. It is available at:

<https://github.com/xqyww123/NTP4VC>

REFERENCES

- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 Technical Report, 2024. URL <https://arxiv.org/abs/2303.08774>.
- Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pp. 454–463. PMLR, 2019. URL <http://proceedings.mlr.press/v97/bansal19a.html>.
- Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In Dana Fisman and Grigore Rosu (eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 415–442, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99524-9.
- Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lecture Notes in Computer Science*, pp. 364–387. Springer, 2006. URL https://doi.org/10.1007/11804192_17.
- Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, pp. 14, 2010.
- Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer (eds.), *Computer Aided Verification*, pp. 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22110-1.
- Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pp. 71–90. Springer, 2011. URL https://doi.org/10.1007/978-3-642-22792-9_5.
- Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The

- dogged pursuit of bug-free c programs: the frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, July 2021. ISSN 0001-0782. URL <https://doi.org/10.1145/3470569>.
- Allan Blanchard, Nikolai Kosmatov, and Frédéric Loulergue. Ghosts for lists: A critical module of Contiki verified in Frama-C. In *NASA Formal Methods (NFM 2018)*, volume 10811 of *Lecture Notes in Computer Science*, pp. 37–53. Springer, 2018. doi: 10.1007/978-3-319-77935-5_3.
- Jasmin Christian Blanchette, Fabian Meier, Andrei Popescu, and Dmitriy Traytel. Foundational nonuniform (co)datatypes for higher-order logic. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pp. 1–12. IEEE Computer Society, 2017. URL <https://doi.org/10.1109/LICS.2017.8005071>.
- François Bobot, Jean-Christophe Filliatre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 Platform*. University Paris-Saclay, CNRS, Inria, version 1.8.2 edition, September 2025a. URL <https://www.why3.org/doc/>.
- François Bobot, Jean-Christophe Filliatre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. Basic strategies. <https://www.why3.org/doc/technical.html#basic-strategies>, September 2025b.
- Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In *Automated Reasoning (IJCAR 2010)*, volume 6173 of *Lecture Notes in Computer Science*, pp. 107–121. Springer, 2010. URL https://doi.org/10.1007/978-3-642-14203-1_9.
- Jochen Burghardt, Jens Gerlach, and Timon Lapawczyk. ACSL by example. Technical report, Fraunhofer FOKUS, 2015. URL <https://publica.fraunhofer.de/entities/publication/beb926ba-c3d6-4570-acc6-dd50da41843f>.
- Nuno Carvalho, Cristiano da Silva Sousa, Jorge Sousa Pinto, and Aaron Tomb. Formal verification of kLIBC with the WP Frama-C plug-in. In Julia M. Badger and Kristin Yvonne Rozier (eds.), *NASA Formal Methods (NFM 2014)*, volume 8430 of *Lecture Notes in Computer Science*, pp. 343–358. Springer, 2014. doi: 10.1007/978-3-319-06200-6_29.
- Zhoujun Cheng, Richard Fan, Shibo Hao, Taylor W Killian, Haonan Li, Suqi Sun, Hector Ren, Alexander Moreno, Daqian Zhang, Tianjun Zhong, et al. K2-think: A parameter-efficient reasoning system. *arXiv preprint arXiv:2509.07604*, 2025.
- Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2): 56–68, 1940. URL <https://doi.org/10.2307/2266170>.
- Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*, pp. 23–42. Springer, 2009. URL https://doi.org/10.1007/978-3-642-03359-9_2.
- Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018. URL <https://inria.hal.science/hal-01960203>.
- Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, 1988. URL [https://doi.org/10.1016/0890-5401\(88\)90005-3](https://doi.org/10.1016/0890-5401(88)90005-3).
- Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*, 2025. URL <https://frama-c.com/download/frama-c-user-manual.pdf>. Corresponds to Frama-C 31.0 (Gallium), released on 2025-06-24.
- Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ Analyzer. In Mooly Sagiv (ed.), *Programming Languages and Systems*, pp. 21–30, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-31987-0.

- Lukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1–4):423–453, 2018. URL <https://doi.org/10.1007/s10817-018-9458-4>.
- Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.
- Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof (eds.), *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pp. 337–340. Springer, 2008. URL https://doi.org/10.1007/978-3-540-78800-3_24.
- Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: A foundry for the deductive verification of rust programs. In *Formal Methods and Software Engineering (ICFEM)*, volume 13478 of *Lecture Notes in Computer Science*, pp. 90–105. Springer, 2022. URL https://doi.org/10.1007/978-3-031-17244-1_6.
- Adel Djoudi, Martin Hána, and Nikolai Kosmatov. Formal verification of a JavaCard virtual machine with Frama-C. In *Formal Methods (FM 2021)*, volume 13047 of *Lecture Notes in Computer Science*, pp. 427–444. Springer, 2021. doi: 10.1007/978-3-030-90870-6_23.
- Quinn Dougherty and Ronak Mehta. Proving the coding interview: A benchmark for formally verified code generation. In *2025 IEEE/ACM 2nd International Workshop on Large Language Models for Code (LLM4Code)*, pp. 72–79. IEEE, 2025. URL <https://doi.org/10.1109/LLM4Code66737.2025.00014>. Duplicate of entry 11028548.
- Aaron Dutle, Mariano Moscato, Laura Titolo, César Muñoz, Gregory Anderson, and François Bobot. Formal analysis of the compact positionreporting algorithm. *Formal Aspects of Computing*, 33(1):65–86, Jan 2021. ISSN 1433-299X. URL <https://doi.org/10.1007/s00165-019-00504-0>.
- Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila. Journey to a rte-free x.509 parser. In *Symposium sur la sécurité des technologies de l’information et des communications (SSTIC 2019)*, pp. 1–50, Rennes, France, 2019. URL https://www.sstic.org/media/SSTIC2019/SSTIC-actes/journey-to-a-rte-free-x509-parser/SSTIC2019-Article-journey-to-a-rte-free-x509-parser-ebalard_mouy_benadjila_3cUxSCv.pdf. Talk on 6 June 2019; paper PDF available via the presentation page.
- Denis Efremov, Mikhail Mandrykin, and Alexey Khoroshilov. Deductive verification of unmodified Linux kernel library functions. In Tiziana Margaria and Bernhard Steffen (eds.), *Leveraging Applications of Formal Methods, Verification and Validation. Verification (ISOLA 2018)*, volume 11245 of *Lecture Notes in Computer Science*, pp. 216–234. Springer, 2018. doi: 10.1007/978-3-030-03421-4_15.
- Gidon Ernst, Marieke Huisman, Wojciech Mostowski, and Mattias Ulbrich. Verifythis - verification competition with a human factor. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (eds.), *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part III*, volume 11429 of *Lecture Notes in Computer Science*, pp. 176–195. Springer, 2019. URL https://doi.org/10.1007/978-3-030-17502-3_12.
- Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *Programming Languages and Systems (ESOP 2013)*, volume 7792 of *Lecture Notes in Computer Science*, pp. 125–128. Springer, 2013. URL https://doi.org/10.1007/978-3-642-37036-6_8.
- Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 expert survey on formal methods. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 12327 of *Lecture*

- Notes in Computer Science*, pp. 3–69. Springer, 2020. URL https://doi.org/10.1007/978-3-030-58298-2_1.
- Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. TacticToe: Learning to prove with tactics. *J. Autom. Reason.*, 65(2):257–286, February 2021. ISSN 0168-7433. URL <https://doi.org/10.1007/s10817-020-09580-x>.
- John Harrison, Josef Urban, and Freek Wiedijk. History of Interactive Theorem Proving. In Jörg H. Siekmann (ed.), *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pp. 135–214. North-Holland, 2014. URL <https://doi.org/10.1016/B978-0-444-51624-4.50004-6>.
- C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. URL <https://doi.org/10.1145/363235.363259>.
- Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL <https://openreview.net/forum?id=r1xwKoR9Y7>.
- Cezary Kaliszyk, François Chollet, and Christian Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL <https://openreview.net/forum?id=ryuxYmvel>.
- Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholtz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark A. Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. The 1st verified software competition: Experience report. In Michael J. Butler and Wolfram Schulte (eds.), *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *Lecture Notes in Computer Science*, pp. 154–168. Springer, 2011. URL https://doi.org/10.1007/978-3-642-21437-0_14.
- Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In Natasha Sharygina and Helmut Veith (eds.), *Computer Aided Verification*, pp. 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39799-8.
- Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023. URL <https://doi.org/10.1145/3586037>.
- Julia Lawall, Keisuke Nishimura, and Jean-Pierre Lozi. Should we balance? towards formal verification of the linux kernel scheduler. In Roberto Giacobazzi and Alessandra Gorla (eds.), *Static Analysis (SAS 2024)*, volume 14995 of *Lecture Notes in Computer Science*, pp. 194–215. Springer, 2025. doi: 10.1007/978-3-031-74776-2_8.
- K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 6355 of *Lecture Notes in Computer Science*, pp. 348–370. Springer, 2010. URL https://doi.org/10.1007/978-3-642-17511-4_20.
- Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. Isarstep: a benchmark for high-level mathematical reasoning. In *ICLR 2021*, 2021. URL <https://arxiv.org/abs/2006.09265>. OpenReview version.
- Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. FVEL: Interactive formal verification environment with large language models via theorem proving. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 54932–54946. Curran Associates, Inc., 2024. URL <https://doi.org/10.52202/079017-1742>.

- Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, et al. Goedel-prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, 2025.
- Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.
- Evan Lohn and Sean Welleck. minicodeprops: a minimal benchmark for proving code properties. *arXiv preprint arXiv:2406.11915*, 2024a. URL <https://arxiv.org/abs/2406.11915>. submitted 16 June 2024; version v2.
- Evan Lohn and Sean Welleck. minicodeprops: a minimal benchmark for proving code properties, 2024b. URL <https://arxiv.org/abs/2406.11915>.
- Chloe R Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. Dafnybench: A benchmark for formal software verification. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL <https://openreview.net/forum?id=yBgTVWccIx>.
- Frédéric Mangano, Simon Duquennoy, and Nikolai Kosmatov. Formal verification of a memory allocation module of Contiki with Frama-C: A case study. In *Risks and Security of Internet and Systems (CRiSIS 2016)*, volume 10158 of *Lecture Notes in Computer Science*, pp. 114–120. Springer, 2017. doi: 10.1007/978-3-319-54876-0_9.
- Pasquale Minervini, Matko Bosnjak, Tim Rocktäschel, and Sebastian Riedel. Towards neural theorem proving at scale. *arXiv preprint arXiv:1807.08204*, 2018.
- Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Yuanyuan Zhou. Laurel: Unblocking automated verification with large language models. *Proc. ACM Program. Lang.*, 9(OOPSLA1), April 2025. URL <https://doi.org/10.1145/3720499>.
- Lawrence C. Paulson. Isabelle: The Next 700 Theorem Provers. In Piergiorgio Odifreddi (ed.), *Logic and Computer Science*, pp. 361–386. Academic Press, London, 1990.
- Mário Pereira and António Ravara. Cameleer: A Deductive Verification Tool for OCaml. In *Computer Aided Verification (CAV)*, volume 12760 of *Lecture Notes in Computer Science*, pp. 677–689. Springer, 2021. URL https://doi.org/10.1007/978-3-030-81688-9_31.
- Alexandre Peyrard, Nikolai Kosmatov, Simon Duquennoy, and Shahid Raza. Towards Formal Verification of Contiki: Analysis of the AES–CCM* Modules with Frama-C. In *RED-IOT 2018 - Workshop on Recent advances in secure management of data and resources in the IoT*, Madrid, Spain, February 2018. URL <https://inria.hal.science/hal-01670119>.
- Baptiste Pollien, Christophe Garion, Gautier Hattenberger, Pierre Roux, and Xavier Thirioux. Verifying the mathematical library of an UAV autopilot with Frama-C. In Alberto Lluch Lafuente and Anastasia Mavridou (eds.), *Formal Methods for Industrial Critical Systems (FMICS 2021)*, volume 12863 of *Lecture Notes in Computer Science*, pp. 167–173. Springer, 2021. doi: 10.1007/978-3-030-85248-1_10.
- ZZ Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *arXiv preprint arXiv:2504.21801*, 2025.
- John Rushby. Software Verification and System Assurance. In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pp. 3–10, 2009. URL <https://doi.org/10.1109/SEFM.2009.39>.
- Stephan Schulz. E – A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002. URL <https://doi.org/10.3233/EAI-2002-260>.

- Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable Reconstruction of Fine-grained Proofs in a Proof Assistant. In André Platzer and Geoff Sutcliffe (eds.), *Automated Deduction – CADE 28*, pp. 450–467, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.
- SMT-LIB Initiative. SMT-LIB — logics. <https://smt-lib.org/logics.shtml>. Accessed: 2025-11-18.
- Chuyue Sun, Ying Sheng, Oded Padon, and Clark W. Barrett. Clover: Closed-loop verifiable code generation. In Guy Avni, Mirco Giacobbe, Taylor T. Johnson, Guy Katz, Anna Lukina, Nina Narodytska, and Christian Schilling (eds.), *AI Verification - First International Symposium, SAV 2024, Montreal, QC, Canada, July 22-23, 2024, Proceedings*, volume 14846 of *Lecture Notes in Computer Science*, pp. 134–155. Springer, 2024. URL https://doi.org/10.1007/978-3-031-65112-0_7.
- Geoff Sutcliffe. Stepping stones in the TPTP world. In *Automated Reasoning (IJCAR 2024)*, volume 14739 of *Lecture Notes in Computer Science*, pp. 30–50. Springer, 2024. doi: 10.1007/978-3-031-63498-7_3.
- Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetsche, Greg Durrett, Yisong Yue, and Swarat Chaudhuri. Clever: A curated benchmark for formally verified code generation, 2025. URL <https://arxiv.org/abs/2505.13938>.
- Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. Rango: Adaptive retrieval-augmented proving for automated software verification. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pp. 347–359. IEEE, 2025. URL <https://doi.org/10.1109/ICSE55347.2025.00161>.
- Toccata Team. Gallery of verified programs. <https://toccata.gitlabpages.inria.fr/toccata/gallery/index.en.html>, 2025. Joint team of Inria, CNRS and University of Paris-Saclay. Page generated on 2025-09-16. Accessed 2025-09-17.
- George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. Putnambench: Evaluating neural theorem-provers on the putnam mathematical competition. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/1582eaf9e0cf349e1e5a6ee453100aa1-Abstract-Datasets_and_Benchmarks_Track.html.
- Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. Making Higher-Order Superposition Work. *Journal of Automated Reasoning*, 66(4):541–564, Nov 2022. ISSN 1573-0670. URL <https://doi.org/10.1007/s10817-021-09613-z>.
- Hillel Wayne. The great theorem prover showdown, 2018. URL <https://www.hillelwayne.com/post/theorem-prover-showdown/>. Blog post.
- Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischniewski. SPASS version 3.5. In Renate A. Schmidt (ed.), *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pp. 140–145. Springer, 2009. URL https://doi.org/10.1007/978-3-642-02959-2_10.
- Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), October 2009. ISSN 0360-0300. URL <https://doi.org/10.1145/1592434.1592436>.

Huajian Xin, Luming Li, Xiaoran Jin, Jacques Fleuriot, and Wenda Li. Ape-bench i: Towards file-level automated proof engineering of formal math libraries, 2025. URL <https://arxiv.org/abs/2504.19110>.

Qiyuan Xu, Renxi Wang, Peixin Wang, Haonan Li, and Conrad Watt. A Minimalist Proof Language for Neural Theorem Proving over Isabelle/HOL, 2025. URL <https://arxiv.org/abs/2507.18885>.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu K. Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. Autoverus: Automated proof generation for rust code. *CoRR*, abs/2409.13082, 2024. URL <https://doi.org/10.48550/ARXIV.2409.13082>.

Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6984–6994. PMLR, 09–15 Jun 2019. URL <https://proceedings.mlr.press/v97/yang19a.html>.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Neural Information Processing Systems (NeurIPS), Datasets & Benchmarks Track*, 2023.

Z3. Arithmetic — online Z3 guide, 2025. URL <https://microsoft.github.io/z3guide/docs/theories/Arithmetic>. Accessed: 2025-11-19.

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. URL <https://openreview.net/forum?id=9ZPegFuFTFv>.

Sicheng Zhong, Jiading Zhu, Yifang Tian, and Xujie Si. Rag-verus: Repository-level program verification with llms using retrieval augmented generation. *CoRR*, abs/2502.05344, 2025. URL <https://doi.org/10.48550/ARXIV.2502.05344>.

A LICENSING AND RULES OF ENGAGEMENT

A.1 LICENSE

As listed in Tab. 3, the source projects of our benchmark cases are released under different open-source licenses, some of which are mutually incompatible (e.g., GPLv2 vs. GPLv3). However, since we do not compile these code bases into a single target, we can still distribute the data under a hybrid license. Accordingly, the VCs generated from these projects are released under the same licenses as their original source projects. The code written by us is released under the MIT License.

A.2 PROTECTION TO PREVENT DATA LEAKAGE

To prevent data leakage, if any VC from a given `mlw` file is selected for the NTP4VC benchmark; all VCs from that `mlw` file should be protected and excluded from training. In contrast, other `mlw` files provided in our associated artifact are not protected, as long as no VC from those files is included in the NTP4VC benchmark. Note that the `mlw` files in the Real C Verification category are generated from C projects, with each `mlw` file corresponding to a single C function. The identity of an `mlw` file should be considered bound to its original C function. This means that `mlw` files generated through alternative means (e.g., different compilation flags) that produce varied content should also be subject to the above protection.

A.3 ACCEPTABLE IMPROVEMENTS TO THE TRANSLATION PIPELINE

The purpose of this benchmark is to advance Neural Theorem Proving capabilities on proving Verification Conditions. Note that NTP models built on top of specific ITP platforms are inevitably influenced by the language features and idioms of their respective platforms. The rule-based translation provided in this work may not fully capture all such features and idioms, resulting in some VCs being translated into representations that are unnatural or unconventional for the target ITP platform. These should be considered as deficiencies in the translation pipeline. Therefore, to allow for addressing these potential issues, improvements to the translation pipeline that address such unnatural or unconventional representations should be considered legitimate and acceptable. To be clear, regenerating the NTP4VC benchmark using an improved translation pipeline as described above and evaluating on the regenerated benchmark is likewise considered legitimate and acceptable. However, any evaluation based on a regenerated NTP4VC benchmark must explicitly report any modifications made to the translation pipeline when presenting results.

B ADDITIONAL BACKGROUND

Although VC has been introduced in § 1, given its significance to our work, we provide a precise definition as follows.

Definition 1. Given a program and a property, a *Verification Condition* (VC) is a mathematical proposition that, when proven true, guarantees the program satisfies the desired property.

Additionally, another aspect that remains unspecified in the main text is the target property of the program verification discussed in our work. Any program verification task always considers a target property. The **target property** considered in our benchmark is **Functional Correctness**, which guarantees a program correctly implements its desired function — for any allowed input, the output of the program always satisfies a separately written logical specification of the program’s behaviour (see appendix C for a concrete example). Functional correctness is a verification goal widely adopted in real-world industrial practice (Garavel et al., 2020), and it is also a primary capability of our toolchain components Why3 and Frama-C.

C AN EXAMPLE OF VC

This section presents an example Why3 program and its VC to provide readers with a concrete sense of how VCs relate to traditional mathematical theorems.

The left side of Fig. 5 presents a Why3 program for binary search. Its functional correctness property is given by the `requires`, `ensures`, and `raises` clauses. `requires` specifies the domain of valid inputs, i.e., the given array a must be sorted. `ensures` and `raises` specify the expectation of the output — conditions that the $result$ has to satisfy, which are, 1) the $result$ is a valid index (i.e., between 0 and the length) such that array a ’s element at the index has a value of v , if no exception raises, or 2), if exception `NF` raises, no element in the array has a value of v .

This program involves mutable references and an effectful loop, which makes direct reasoning with ITPs extremely tedious. The mature academic and industrial solution is to apply a specialized program reasoning engine, like Why3’s VCG, to first extract pure logical proof goals, so-called VCs.

The `invariant` and `variant` clauses are annotations that help the VCG to work. The `invariant` clause declares a loop invariant, which is a formula that remains true throughout every loop iteration, and is required by the VCG process. The `variant` clause declares a metric which is strictly decreasing in each loop iteration. It helps to generate the VCs for ensuring loop termination.

The `assert` at line 13 is an annotation to ease the burden of VC prover. It introduces a subgoal and instructs the verifier to first prove this subgoal and then use the proven subgoal as a premise (as shown in pink in Fig. 5) in the subsequent proofs. Essentially, it helps the prover to decompose VCs into simpler subgoals.

The right side of Fig. 5 is one of the generated VCs for the functional correctness, a mathematical statement that encodes the logic behind the program’s behaviors. First, invariant $\mathcal{I}(l, u)$ represents that l, u are valid boundaries of the indices of the elements of value v . Then, consider the case of

```

1 Define  $\mathcal{I}(l, u) \triangleq 0 \leq l \wedge u < \text{length}(a) \wedge (\forall i. 0 \leq i < \text{length } a \wedge a[i] = v \rightarrow l \leq i \leq u)$ 
2 exception NF (* standing for not found *)
3 let binary_search (a: array int) (v: int) : int
4   requires  $\forall i j. 0 \leq i \leq j < \text{length}(a) \rightarrow a[i] \leq a[j]$ 
5   ensures  $0 \leq \text{result} < \text{length}(a) \wedge a[\text{result}] = v$ 
6   raises NF  $\rightarrow \forall i. 1 \leq i < \text{length}(a) \rightarrow a[i] \neq v$ 
7   = let ref l = 0 in
8   let ref u = length a - 1 in
9   while l <= u do
10    invariant  $\mathcal{I}(l, u)$ 
11    variant u - l
12    let m = l + div (u - 1) 2 in
13    if a[m] < v then
14      assert  $\forall i. l \leq i < m + 1 \rightarrow a[i] < v$ 
15      l := m + 1
16    else if a[m] > v then u := m - 1
17    else return m
18  done;
19  raise NF
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
786
787
787
788
789
789
789
790
791
792
793
794
795
796
796
797
797
798
798
799
799
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1
```

intellectual creativity, but are complicated and require the prover to process enormous formulas, potentially extracting key information from noise to simplify the proof goals and ultimately complete the proofs.

D LIMITATION & MITIGATION

From a methodological perspective, our VC extraction method ensures all obtained VCs are provable by construction. However, implementation bugs may occur in Why3, Frama-C, or our translation pipeline, potentially rendering some VCs unprovable. To address such potential invalidation, we design the benchmark to be updatable: we will repair the VC extraction pipeline and refresh the benchmark when invalidation occurs. Since the intended semantics of VCs are grounded in the source verification projects, these updates primarily address representation issues while preserving the essential semantics of the verification problems. However, should an invalid benchmark case be irreparable in rare instances, we will eliminate it from the benchmark to guarantee all remaining cases are provable.

E DETAILED EXTRACTON PIPELINE

This section provides further details on our extraction pipeline from two perspectives: approach and implementation

E.1 METHODOLOGY DETAILS

The translation process begins with a given Why3 source code. The process first runs Why3 VCG to generate VCs and calls our customized Why3 printer to dump the VCs into an XML representation of their Abstract Syntax Trees (ASTs). These ASTs are processed by a Python translation framework also written by us and finally mapped into the target ITPs’ languages.

A verification project typically contains multiple VCs that depend on shared Why3 theories consisting of lemmas, axioms, functions, and datatype definitions. These theories may further depend on others, forming a complex dependency graph across the project. To successfully translate the VCs, we must translate all dependent theories. Our translation process, therefore, recursively handles every theory in this dependency graph, mapping the entire verification project into the target ITPs.

In terms of structure, a Why3 theory is a sequence of declarative elements consisting of axioms, definitions of functions, and algebraic data types. All three sorts of declarations have similar counterparts in the target ITPs and can be mapped to them, despite two minor gaps. One is regrading the non-uniform data type (Blanchette et al., 2017), which is not natively supported by Isabelle. Therefore, we circumvent all VCs involving such data types. The other gap pertains to discharging the termination check of recursive function definitions, a conventional requirement for ITPs to ensure the soundness of their logics. Some ITPs’ termination checkers (Isabelle and Rocq) are not strong enough to automatically prove the well-foundedness of certain complicated recursions, even though Why3 has checked all the termination. Since the proof obligation of the termination is irrelevant to the semantics of VCs’ proof obligation, we trust Why3’s termination check and axiomatize this in the ITP translation in case ITP’s termination checker fails.

Having the theory dependencies and theory-level declarations translated, the last work is to translate the term language. Both Why3’s and the ITPs’ term languages are based on the lambda calculus, a core language involving only variables, constants, applications, and function abstractions. This similarity simplifies a lot of the translation process. Overall, the process maps Why3 constants to the target ITPs’ constants, and preserves all other variables, application, and function structures. One exception unsupported by Isabelle is Why3’s add-on feature, the `as`-binding used in pattern matching, which annotates a sub-pattern with a variable and binds the term captured by this sub-pattern to the variable. We convert this into semantically equivalent let-bindings.

E.2 IMPLEMENTATION DETAILS

The implementation of the VC extraction and translation pipeline consists of six main components:

1. A Why3 patch to export Why3’s internal Abstract Syntax Tree (AST) into an XML representation (in ~ 200 lines of OCaml).
2. A Python parser to read the XML representation into an S-expression representation of an extended simply-typed lambda calculus (in ~ 160 lines of Python).
3. Python library functions providing basic support for manipulating the lambda calculus, such as substitution, variable deconfliction, rewriting, and folding over atomic terms (in ~ 800 lines of Python).
4. A Python module for managing Why3 sessions, managing translation contexts (e.g., allocated constant/variable names in the context), and chaining all the components together to run them automatically (in ~ 500 lines of Python).
5. Translation rules, rewriting rules, ad-hoc term rewriting procedures, package management, and syntax check adapter, for each of the Isabelle, Lean, and Rocq (in $\sim 800/790/770$ lines of YAML, $\sim 930/780/970$ lines of Python, for Isabelle, Lean, Rocq, respectively).
6. ITP libraries that map Why3 notions into the ITPs’ native builtins (in $\sim 500/160/200$ lines of Isabelle/Lean/Rocq, respectively)

The subsection elaborates on some of the nontrivial components as follows.

The Why3 patch is modified from Why3’s existing Isabelle printer, which exports Why3 AST in XML format but with Isabelle-specific adaptations. We neutralize these adaptations to make it output the raw Why3 internal AST. Specifically, we remove its mapping from Why3 terms to Isabelle terms; add Rocq and Lean keywords to the blacklist of variable names; fix its escaping of XML special characters; add support for the as-binding syntax in pattern matching; add type annotations to definition exports.

The **S-expression** used in our internal process is a simply-typed (HOL style) lambda calculus extended with native AST nodes for finite Cartesian products, pattern matching (the `case` statement), literal numbers and strings, and the `as`-bindings (which bind the sub-term that matches a sub-pattern to a variable, in a usual pattern matching). Bound variables are represented in the same way as free variables; we do not use De Bruijn indices, but instead maintain contextual variables and deconflict names of bound variables explicitly (because it simplifies our parsing and printing work, while computational efficiency can be compromised in our context).

The **substitution**, **variable deconfliction**, and **folding** are all standard. We use Python’s functional programming features to implement these operations. The **rewriting** system is simplified such that 1) all reducible expression (redex) patterns have the form $(\text{constant } arg_1 \dots arg_n)$ where all $\{arg_i\}_{1 \leq i \leq n}$ are free variables and the arity n is schematic; 2) no lambda abstraction is allowed to appear in the contractum, so the contracta can only be atoms or (nested) function applications. This simplification allows representing a rewriting rule as merely a tuple of the redex’s constant name, the constant’s arity, and a list-represented S-expression for the contractum. We use YAML’s dictionary datatype to represent a set of rewriting rules, e.g., $(\text{Why3.length}, 1, [\text{Int.int}, [\text{Isabelle.length}, arg_0]])$ rewrites $(\text{Why3.length } l)$ into $\text{Int.int}(\text{Isabelle.length } l)$, for any l . This greatly simplifies the writing of rewriting rules. For more complex rewritings that require more complex redex patterns, we use hard-coded Python `match-case` to work over the S-expression directly.

F PROMPTS

Our work employs two types of prompts: general prompts designed for broad-purpose LLMs and specialized prompts tailored for particular fine-tuned models.

The templates of the general prompts are shown as follows.

General Prompt for Isabelle

Given the following Isabelle theories as context, prove the Isabelle proposition given at the end.

File ‘NTP4Verif.thy’:

{content of the theory file}

And many other libraries

File ‘imp_SymStateSet.thy’:
{content of the theory file}

Given the context above, consider the proposition in the following Isabelle code:
{the target proof goal together with its contextual theory}

Response the Isabelle proof only. Do not repeat any context nor the statement.

General Prompt for Lean

Given the following Lean 4 theories as context, prove the Lean 4 proposition given at the end.

File ‘Base.lean’:
{content of the theory file}

And many other libraries

File ‘SymStateSet.lean’:
{content of the theory file}

Given the context above, consider the proposition in the following Lean 4 code:
{the target proof goal together with its contextual theory}

Response the Lean 4 proof only. Do not repeat any context nor the statement.

General Prompt for Rocq

Given the following Rocq theories as context, prove the Rocq proposition given at the end.

File ‘Base.v’:
{content of the theory file}

And many other libraries

File ‘SymStateSet.v’:
{content of the theory file}

Given the context above, consider the proposition in the following Rocq code:
{content}

Response the Rocq proof only. Do not repeat any context nor the statement.

The template specifically for Goedel-Prover and DeepSeek-Prover is as follows.

Prompt for Specialized Models

Complete the following Lean 4 code:
{the target proof goal together with its contextual theory}

Before producing the Lean 4 code to formally prove the given theorem, provide a detailed proof plan outlining the main proof steps and strategies.
The plan should highlight key ideas, intermediate lemmas, and proof structures that will guide the construction of the final formal proof.

```

1 lemma decompose_front_node_vc:
2   fixes l :: "'a t2"
3   fixes r :: "'a t2"
4   fixes o1 :: "'a view"
5   fixes d :: "'a t1"
6   assumes fact0: "int balancing ≤ hgt (m1 l) - hgt (m1 r)"
7   assumes fact1: "hgt (m1 l) - hgt (m1 r) ≤ int balancing"
8   assumes fact2: "case o1 of (AEmpty :: 'a view) ⇒ hgt (m1 l) = (0 :: int) ∧ ..."
9   shows "case o1 of (AEmpty :: 'a view) ⇒ True
10    | ANode l1 d2 r2 _ _ ⇒ (((0 :: int) ≤ hgt (m1 l) ∧ ...) ∧
11      "∀(d2 :: 'a t1) (res :: 'a t2). (case o1 of (AEmpty :: 'a view) ⇒ d2 = d ∧ res = r
12      | ANode l1 d21 r2 _ _ ⇒ (exists(res1 :: 'a t2). (node_model (seq (m1 l1)) ...)))"
13    proof -
14    {
15      fix d2 res
16      assume pre: "case o1 of AEmpty ⇒ d2 = d ∧ res = r
17      | ANode l1 d21 r2 h s ⇒ ∃res1. node_model (seq (m1 l1)) d21 (seq (m1 r2)) = Cons d2 (seq (m1 res1)) ∧
18        (0 ≤ (1 + (if hgt (m1 l1) < hgt (m1 r2) then hgt (m1 r2) else hgt (m1 l1))) - hgt (m1 res1) ∧
19        (1 + (if hgt (m1 l1) < hgt (m1 r2) then hgt (m1 r2) else hgt (m1 l1))) - hgt (m1 res1) ≤ 1) ∧
20        seq (m1 res) = node_model (seq (m1 res1)) d (seq (m1 r)) ∧
21        (0 ≤ (1 + (if hgt (m1 res1) < hgt (m1 r) then hgt (m1 r) else hgt (m1 res1))) - hgt (m1 res) ∧
22        (1 + (if hgt (m1 res1) < hgt (m1 r) then hgt (m1 r) else hgt (m1 res1))) - hgt (m1 res) ≤ 1) ∧
23        (-int balancing ≤ hgt (m1 res1) - hgt (m1 r) ∧ hgt (m1 res1) - hgt (m1 r) ≤ int balancing →
24        (1 + (if hgt (m1 res1) < hgt (m1 r) then hgt (m1 r) else hgt (m1 res1))) = hgt (m1 res)))"
25
26      show "node_model (seq (m1 l)) d (seq (m1 r)) = Cons d2 (seq (m1 res)) ∧
27        (0 ≤ (1 + (if hgt (m1 l) < hgt (m1 r) then hgt (m1 r) else hgt (m1 l))) - hgt (m1 res) ∧
28        (1 + (if hgt (m1 l) < hgt (m1 r) then hgt (m1 r) else hgt (m1 l))) - hgt (m1 res) ≤ 1)"
29    proof (cases o1)
30      case AEmpty
31      with pre have d2_def: "d2 = d" and res_def: "res = r" by simp_all
32      from fact2[unfolded AEmpty] have seq_l: "seq (m1 l) = []" and hgt_l: "hgt (m1 l) = 0" by simp_all
33
34      show ?thesis unfolding d2_def res_def
35      by (simp add: seq_l node_model_def, auto simp: hgt_l fact0 fact1)
36    next
37      case (ANode l1 d21 r2 h s)
38      with pre obtain res1 where
39        decomp: "node_model (seq (m1 l1)) d21 (seq (m1 r2)) = Cons d2 (seq (m1 res1))" and
40        res_seq: "seq (m1 res) = node_model (seq (m1 res1)) d (seq (m1 r))" by auto
41
42      from fact2[unfolded ANode] have seq_l:
43        "seq (m1 l) = node_model (seq (m1 l1)) d21 (seq (m1 r2))"
44      by simp
45
46      show ?thesis
47      by (simp add: seq_l decomp res_seq node_model_def,
48        metis (mono_tags, lifting) fact2 ANode case_prod_conv)
49    qed
50  qed
51 }
52 }
53 qed

```

Listing 1: The complete proof generated by DeepSeek-V3.1 for the correctness of the `decompose_front_node` function on AVL trees.

G FAILURE CASES

To investigate the failure modes of NTP models on verification conditions, we analyzed the error logs and proof scripts from our evaluation. We highlight three dominant categories of errors: syntactic invalidity, semantic degeneration, and hallucination. It is important to note that the statistics presented below represent *conservative lower bounds*. For syntax and hallucination errors, proof assistants abort execution at the first error; thus, a single proof might contain multiple subsequent errors that remain uncounted. Similarly, our detection of semantic degeneration relies on rigid regular expressions for some common cases, likely missing more subtle forms of degeneration.

Generating syntactically well-formed terms remains a primary hurdle, particularly for complex nested expressions in VCs. In our analysis of Isabelle proof attempts, we found that **at least 24%** failed solely due to syntax errors. Listing 1 shows the complete erroneous proof generated by DeepSeek-V3.1 for proving the correctness of the `decompose_front_node` function on AVL trees. This function is responsible for decomposing the front node of an AVL tree, and its correctness is specified by the corresponding VC. Specifically, the term `seq (m1 l)` represents the sequence of elements in the left subtree `l`, `d` refers to the data element of the current node, and `hgt (m1 r)` denotes the height of the right subtree `r`. The generated proof attempts to first introduce the universally quantified variables `d2` and `res`, followed by a case analysis on `o1`, which represents the structure of the AVL tree. However, the term cannot be parsed due to two subtle syntax errors:

```

1 lemma goal10 (a : Memory.addr) (t_1 : Memory.addr -> Z) (t_4 : Memory.addr -> Memory.addr) (t : Z -> Z)
2 (t_3 : Memory.addr -> Z) (t_2 : Memory.addr -> Z) :
3   let a_1 : Memory.addr := Memory.shift a (1 : Z);
4   let x : Z := t_1 a_1;
5   let a_2 : Memory.addr := Memory.shift a (0 : Z);
6   let x_1 : Z := t_1 a_2;
7   let x_2 : Z := x * x_1;
8   let a_3 : Memory.addr := Memory.shift a (2 : Z);
9   let a_4 : Memory.addr := Memory.shift a (3 : Z);
10  let a_5 : Memory.addr := t_4 a_3;
11  Memory.region (Memory.addr.base a) <= (0 : Z) -> -(2147483648 : Z) <= x_2 -> x_2 <= (2147483647 : Z) ->
12  Memory.framed t_4 -> Memory.linked t -> Memory.sconst t_3 -> Cint.is_uint16 x -> Cint.is_uint16 x_1 ->
13  Axiomatic13.p_valid_memb t t_4 t_1 a -> Memory.valid_rd t a_3 (1 : Z) -> Memory.valid_rd t a_4 (1 : Z) ->
14  Memory.valid_rd t a_1 (1 : Z) -> Memory.valid_rd t a_2 (1 : Z) -> Axiomatic12.p_valid_or_empty t a_5 x ->
15  A_MemSet.l_memset (Memory.havoc t_2 t_3 (Memory.shift a_5 (0 : Z)) x) a_5 (0 : Z) x = true ->
16  Axiomatic12.p_valid_or_empty t (t_4 a_4) (Cint.to_uint32 x_2)
17 :=
18  intro a_1 x a_2 x_1 x_2 a_3 a_4 a_5 h_region h_x2_lower h_x2_upper h_framed t_4_linked t_3_sconst h_x_uint16
19  h_x1_uint16 h_p_valid_memb h_valid_rd_a3 h_valid_rd_a4 h_valid_rd_a1 h_valid_rd_a2 h_p_valid_or_empty_a5
20  h_memset
21  have h_main : Axiomatic12.p_valid_or_empty t (t_4 a_4) (Cint.to_uint32 x_2) := by
22    have h1 : Axiomatic12.p_valid_or_empty t (t_4 a_4) (Cint.to_uint32 x_2) := by
23      -- Use the assumption `h_framed` to deduce the required property
24      -- This step is hypothetical and depends on the actual definition of `Axiomatic12.p_valid_or_empty`.
25      -- and how it interacts with `Memory.framed`.
26      -- In practice, we might need more specific information or lemmas to make this step rigorous.
27      -- For the sake of this example, we suppose that `h_framed` directly implies the conclusion.
28      exfalso
29      have h2 := h_framed
30      have h3 := t_4_linked
31      have h4 := t_3_sconst
32      have h5 := h_p_valid_memb
33      have h6 := h_valid_rd_a3
34      have h7 := h_valid_rd_a4
35      have h8 := h_valid_rd_a1
36      have h9 := h_valid_rd_a2
37      have h10 := h_p_valid_or_empty_a5
38      have h11 := h_memset
39      simp_all [Axiomatic12.p_valid_or_empty, Axiomatic13.p_valid_memb, Memory.framed,
40      Memory.linked, Memory.sconst]
41    <;>
42    (try contradiction) <;>
43    (try norm_num at *) <;>
44    (try aesop)
45    <;>
46    (try
47      {
48        simp_all [Cint.is_uint16]
49      <;>
50      norm_num at *
51      <;>
52      omega
53    })
54    exact h1
55  exact h_main

```

Listing 2: Example of semantic degeneration: Redundant variable renaming in a Lean proof.

(1) a missing closing parenthesis in a deeply nested arithmetic expression on line 16, and (2) two extraneous closing parentheses on lines 18 and 25, respectively. In fact, if one only removes the last extraneous closing parenthesis, the term can be parsed. However, it will result in a term in the form of “ $\dots \wedge (0 \leq (1 + expr) - hgt(m1_res1) \wedge \dots)$ ”, which is syntactically valid but semantically incorrect (the height of $res1$ is being conjoined with another inequality). What one would expect is instead “ $\dots \wedge (0 \leq (1 + expr) - hgt(m1_res1)) \wedge \dots$ ”, which requires removing the extraneous parenthesis on line 16 and adding a closing parenthesis after “ $res1$ ”. The lengthy logical formulas with deeply nested constructs is a common pattern in VCs, which poses significant challenges for NTP models to maintain long-range syntactic coherence.

NTP models frequently lose track of the proof state, resulting in repetitive, meaningless steps. We detected this behavior by matching patterns of continuous “renaming” (e.g., using `have h1 := h2` where both `h1` and `h2` are simple identifiers) repeated at least three times. In Lean, **more than 64%** of proofs generated by Goedel-Prover-V2-32B exhibited this specific degeneration pattern. Listing 2 exemplifies the generation of repetitive and meaningless tactic applications in Lean. The model (Goedel-Prover-V2-32B) engages in a redundant “renaming ritual” (`have h2 := h_framed`, etc.), erroneously assuming that automated tactics like `simp_all` require local variable aliases to access the context. This behavior likely stems from domain shift, where the proof context is more complex than the standard mathematical corpora used for training. Furthermore, the comments (e.g., “assume that `h_framed` directly implies the conclusion”) explicitly admit that the

logical step is hypothetical. This suggests its inability to derive the necessary lemmas to complete the proof.

Models often invoke non-existent constants, lemmas, or tactics due to hallucinations. For instance, GPT-04-mini frequently attempts to solve Rocq VCs using a `why3` tactic, which does not exist in the language. In Isabelle, **at least 9%** of failures were triggered by references to undefined constants or lemmas that are absent from the context. We identified these cases by explicitly matching keywords such as “Undefined fact” or “Undefined constant” in the error logs. Crucially, since the proof assistant terminates the checking process at the first encountered error, hallucinations present in the latter parts of proof scripts — especially those already halted by syntax errors or earlier tactic failures — remain uncounted. Consequently, this 9% figure represents a highly conservative lower bound.

H CLASSIFICATION & METRIC DETAILS OF TABLE 4

The operation classification is conducted on the Isabelle version of our benchmark. We developed Isabelle extensions to analyze the expressions of the obtained proof goals. We elaborate on the constitution of each category in Tab. 4 as follows.

- *Integer Arith* consists of addition, subtraction, multiplication, division, exponentiation, comparison, square root, and factorial operations whose operands are integers, natural numbers, or bounded integers (machine integers); and also bit-width conversions and bitwise operations.
- *Non-linear Arith* consists of multiplication, division, and exponentiation between non-constant expressions, following [de Moura & Bjørner \(2008\)](#) and [Z3 \(2025\)](#). In Tab. 4, most cases contain one distinct non-linear operator — multiplication — and sometimes contain additional operators like division and exponentiation.
- *Float Arith* consists of arithmetics on floating-point numbers. All floating-point numbers in our benchmark cases are modeled as real numbers, disregarding precision errors.
- *List, Sequence* consists of operations involving the `list` type and Why3’s `sequence`, `array31`, `array32`, and `array63`.
- *Set, Map, Bag* consists of operations whose types involve finite map, multiset, finite set, predicate-based set, and hash-table.
- *Tree, String, Matrix* consists of operations whose types involve Why3’s built-in binary tree, string, and matrix.
- *Memory* consists of operations whose types involve Frama-C’s memory encoding.
- *Custom Datatype* consists of operations whose types involve any datatype not provided by the system library but defined by the verification projects.

The metric *depth* is the height of the abstract syntax tree of the VCs, in the standard λ -calculus representation with all arguments of every function application represented as siblings.

I INTERSECTION ANALYSIS OF NTP AND HAMMER CAPABILITIES

To understand whether neural and symbolic approaches overlap or diverge in their capabilities, we analyze the intersection between the union of all problems solved by NTP models and the union of all problems solved by hammers. Table 7 presents the results. The results reveal a complementarity: many verification conditions are solved exclusively by one method or the other. This confirms that NTPs and hammers leverage distinct reasoning mechanisms and that neither approach is a subset of the other, highlighting the potential for hybrid solutions.

J IDENTIFYING VCS IN COQSTOP AND FVEL

In order to support the numbers given in Tab. 1, this section describes our approach to identifying VCs in the CoqStoq benchmark (Thompson et al., 2025) and FVEL (Lin et al., 2024). CoqStop’s test set contains 10,396 theorems from 12 Rocq projects; FVEL’s test set contains 1967 cases.

CoqStop CoqStop’s VCs are predominantly drawn from CompCert, which accounts for over 58% of the test set, while other verification-related projects constitute no more than 6%. Therefore, we focus solely on CompCert. In CompCert, the tactics and other constructs that are relevant to program analysis and VC generation are `TransfInstr`, `UseTransfer`, `monadInv`, `step_simulation`, `exploit`, and `match_states`. Among the CompCert VCs in CoqStop, only 1,325 cases involve these tactics, accounting for 12.7% of the total test set. Including other projects that may involve VCs (at most 6%), the total proportion would not exceed 20%.

FVEL All of FVEL’s test cases are extracted from seL4. seL4’s VCs are generated using the tactics `vcg`, `wp`, and `wpsimp`. Based on the test case list provided by FVEL, we analyzed cases whose proofs contain these tactics and found only 328. Therefore, the proportion of VCs in FVEL does not exceed $328/1967 < 17\%$.

K THE USE OF LARGE LANGUAGE MODELS (LLMs)

We have used LLM as a writing aid to assist with fluency and grammatical checking.

Table 7: The number of problems solved by both hammers and NTP models, only by hammers, and only by NTP models.

Category	Common	Hammer only	NTP only
Algorithm	1	3	1
Data Structure	3	7	0
Calculation	3	5	2
Engineering	5	2	0
Competition	0	3	3
Function	6	19	0
Memory	1	17	1
Loop	3	16	0
Invalid Arg.	5	15	0
Total	27	87	7