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

Anonymous authors
Paper under double-blind review

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 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).

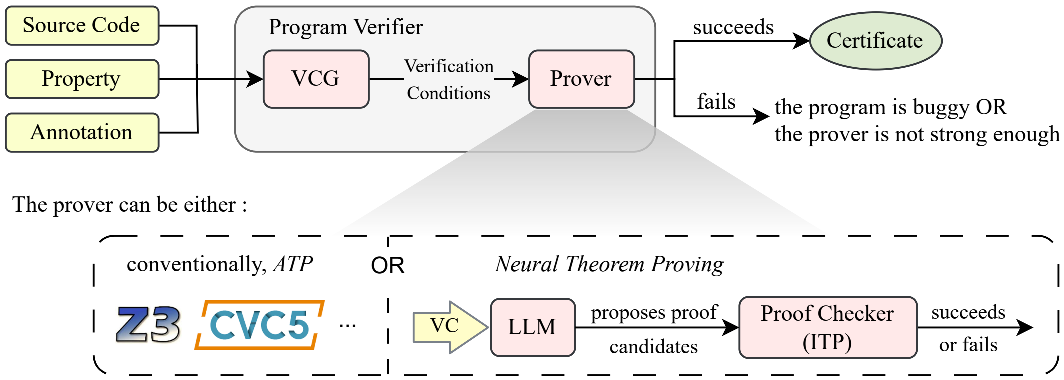

Figure 1: The conventional and NTP-based workflow of program verification.

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 constructing a NTP4VC benchmark.

We overcome this issue by translating the VCs generated from other industrial verification pipelines (Why3 (Filliâtre & 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 and Contiki OS's memory allocator and linked-list library.

Unlike LLM-based translation approaches that suffer from LLMs' unreliability, our translation pipeline is based on $\sim$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).

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

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

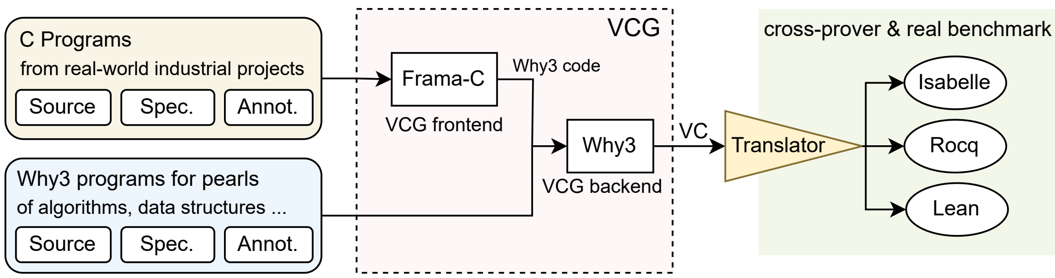

Figure 2: Our pipeline for extracting benchmark cases.

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).

3 A Reliable and Automatic Method for Corpora Generation

This section presents the method we use to extract real-world VCs that consitute 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

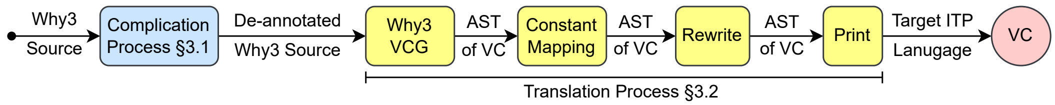

Figure 3: The generation process of the benchmark cases and potentially training corpora.

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., 2025; Correnson et al., 2025). In addition, they exhibit clear syntactic patterns enabling us to identify and erase them. Indeed, the exact job of our complication process is erasing the annotations. The results show this process effectively reduces the pass rate of Why3's strongest ATP from $\sim$99% to $\sim$62% on Why3's bundled examples by Toccata Team (2025).

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 >5.3k VCs from various sources. From there, we carefully select 672 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.

Table 2: Categories of the benchmark

Category	Number	ATP pass
pearls of programs		
Algorithm	62	19.35%
Data Structure	83	20.48%
Calculation	74	20.27%
Engineering	65	20.00%
Competition	52	19.23%
real C verification		
Function	87	16.09%
Loop	110	18.18%
Memory	70	21.43%
Invalid Arg.	69	20.29%
Total	672	19.35%

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). AL3 is a hybrid tactic that combines sophisticated heuristics and five industrial cutting-edge ATPs, Z3 (De Moura & Bjørner, 2008), CVC5 (Barbosa et al., 2022), SPASS (Weidenbach et al., 2009a), 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 category's composition to target an AL3 pass rate of roughly 20% — a level that ensures sufficient open problems for advancing NTP while still allowing effective evaluation of existing approaches.

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)	167	833	BSD-3-Clause
Memory Allocator in Contiki OS (Blanchard et al., 2018)	16	145	BSD-3-Clause
X.509 Parser (Ebalard et al., 2019)	70	5044	GPLv2
Linux Kernel Scheduler's SWB Routine (Lawall et al., 2025)	49	216	GPLv2
Selected Cases from C++ STL (Burghardt et al., 2015)	34	3263	MIT
Total	336	9501	-

Table 4: Statistics of involved operations. Format: $average\ (25^{\text{th}} - 75^{\text{th}}\ percentile)$

Operations	# of cases	# of operations	# of distinct oprs	Size	Depth	# of $\forall\exists$
Integer Arith	645	60.1 (13 – 68)	4.7 (4 – 6)	665 (171 – 772)	61.3 (29 – 82)	11.9 (1 – 15)
Non-Linear Arith	106	11.4 (2 – 13)	1.2 (1 – 1)	1391 (253.5 – 1473)	79.8 (37 – 116)	18.7 (3 – 22)
List, Sequence	234	47.2 (8 – 60.5)	4.0 (2 – 6)	945 (216 – 1151)	54.0 (25 – 62)	18.3 (4 – 23)
Set, Map, Bag	67	46.6 (9 – 48)	3.5 (1 – 5)	905 (288.5 – 1171.5)	44.8 (28 – 55.5)	26.3 (8 – 39)
Tree, String, Matrix	33	53.5 (12 – 84)	4.8 (4 – 6)	695 (180 – 921)	43.0 (25 – 59)	27.8 (4 – 31)
Memory	336	36.2 (13 – 36)	7.9 (6 – 9)	497 (172.5 – 602)	72.2 (50 – 88)	7.0 (0 – 8)
Custom Datatype	242	96.0 (15 – 102)	7.3 (3 – 10)	912 (173 – 1129.5)	58.1 (24 – 83)	17.5 (3 – 24)
All	672	320.0 (78 – 371)	24.9 (20 – 29)	653.5 (158 – 754)	59.8 (28 – 81)	11.7 (1 – 15)

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. The largest share comes from the Contiki OS linked-list library, which contributes most of the hard VCs, since linked lists are notoriously difficult to verify with current industrial tools.

VC Selection Process. The 672 benchmark cases are selected from over 5.3k 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$1.2k candidate cases; three experts then collaboratively evaluate each candidate in the final round to finalize the benchmark set of 672 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 5 projects are all used for benchmark cases.

In the second round, we first select all the hard VCs from the domain, totaling ∼900 cases. Since we aim for the benchmark to have a 20–30% pass rate on the ATP baseline, we correspondingly select ∼300 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., $\mathtt{true} \wedge \mathtt{true}$). 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.

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., 2023) 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., 2023)) 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), IsaMini (Xu et al., 2025)). Among them, DeepSeek-Prover-V2 and Goedel-Prover and 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.

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-o4-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 2025. 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

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 (4.61%) and Isabelle (15.33%).

Model	Lean			Rocq			Isabelle		
	P@1	P@4	P@8	P@1	P@4	P@8	P@1	P@4	P@8
GPT-o4-mini-high	1.19	–	–	1.34	–	–	1.93	–	–
K2-think	1.19	1.79	2.23	0.74	2.08	2.68	0.00	0.00	0.00
DeepSeek-V3.1	1.04	1.79	2.23	0.74	2.08	2.68	1.34	4.32	6.25
Qwen3-32B	0.60	0.60	0.74	0.74	1.34	1.49	0.74	2.53	3.42
Qwen3-235B-A22B	0.60	0.74	0.89	1.04	2.23	2.98	1.19	2.08	3.13
Goedel-Prover-V2-32B	1.19	2.83	2.98	–	–	–	–	–	–
DeepSeek-Prover-V2-671B	2.08	2.83	3.12	–	–	–	–	–	–
IsaMini	–	–	–	–	–	–	2.08	7.29	11.46
CoqHammer / Sledgehammer	–	–	–	4.61	–	–	15.33	–	–

solvers, including CVC4 (Barrett et al., 2011), CVC5 (Barbosa et al., 2022), Z3 (De Moura & Bjørner, 2008), E (Schulz, 2002), SPASS (Weidenbach et al., 2009b), Vampire (Kovács & Voronkov, 2013), veriT (Schurr et al., 2021), and Zipperosition (Vukmirović et al., 2021). 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 4% 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 fundamentally different reasoning capabilities than complex mathematical benchmarks.

By comparison, hammer-based provers show much stronger results on NTP4VC. Sledgehammer achieves a 15.33% pass rate on Isabelle, significantly outperforming all tested NTP models. Similarly, CoqHammer achieves 4.61% on Rocq, outperforming the best NTP model. These results indicate that current traditional automated reasoning techniques employed by hammers remain more effective than current NTP approaches for program verification.

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	6 / 62	9.68%	6 / 62	9.68%
Data Structure	8 / 83	9.64%	11 / 83	13.25%
Calculation	8 / 74	10.81%	13 / 74	17.57%
Engineering	20 / 65	30.77%	7 / 65	10.77%
Competition	1 / 52	1.92%	3 / 52	5.77%
Function	9 / 87	10.34%	18 / 87	20.69%
Loop	6 / 110	5.45%	18 / 110	16.36%
Memory	11 / 70	15.71%	15 / 70	21.43%
Invalid Arg.	7 / 69	10.14%	16 / 69	23.19%
Total	76 / 672	11.31%	107 / 672	15.92%

To better understand performance differences across problem types, we report the number of problems solved per category by NTP models and hammers in Table 6. The results show a clear discrepancy across categories. For instance, both approaches perform relatively well on *Engineering* problems, with NTP models even surpassing hammers (30.77% vs. 10.77%). In contrast, hammers consistently outperform NTP models in most other categories, particularly in *Function* (20.69% vs. 10.34%), *Loop* (16.36% vs. 5.45%), and *Invalid Argument* (23.19% vs. 10.14%).

```
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.

Table 6 also indicates performance disparities between language models and hammers. This disparity is reasonable given the fundamental differences in the underlying mechanisms of models and hammers. Language models analyze proof goals semantically based on their knowledge acquired during training. In domains where they have been exposed to relevant knowledge during training, they tend to perform better. This may explain their better performance in the Engineering category, as the benchmark cases stem from common engineering tricks and implementation optimizations that models have extensively encountered during pre-training. By contrast, hammers perform syntactic analysis of proof goal expressions — analyzing structural relationships between atomic formulas and logical connectives. For example, if we replace variable/constant names in proof goals with random words, hammers' behavior would remain unaffected because the logical structure is unchanged; NTP models would be significantly impacted due to the loss of semantic information embedded in the names. This may account for hammers' sustained performance on general domains.

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.,$:= i_1 + i_2$) instead of declarative, tactic-based reasoning. This confusion is further evidenced by proof scripts degenerating into repetitive and meaningless tactic applications (e.g., "`have` $h_{16} := h_0$`;` `have` $h_{17} := h_1$ ..."), 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 models 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-o4-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 672 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. Our artifact is already submitted as supplementary material.

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. *arXiv preprint arXiv:2303.08774*, 2023.

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 W. 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 - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pp. 415–442. Springer, 2022. doi: 10.1007/978-3-030-99524-9_24. URL https://doi.org/10.1007/978-3-030-99524-9_24.

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 Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (eds.), *Formal Methods for Components and Objects*, pp. 364–387, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-36750-5.

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 *International Conference on Computer Aided Verification*, pp. 171–177. Springer, 2011.

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway (ed.), *Advances in Cryptology – CRYPTO 2011*, pp. 71–90, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22792-9.

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. doi: 10.1145/3470569. 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 Aaron Dutle, César Muñoz, and Anthony Narkawicz (eds.), *NASA Formal Methods*, pp. 37–53, Cham, 2018. Springer International Publishing. ISBN 978-3-319-77935-5.

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. doi: 10.1109/LICS.2017.8005071. URL https://doi.org/10.1109/LICS.2017.8005071.

François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 Platform*. University Paris–Saclay, CNRS, Inria, version 1.8.2 edition, September 2025. URL https://www.why3.org/doc/.

Sascha Böhme and Tobias Nipkow. Sledgehammer: judgement day. In *International Joint Conference on Automated Reasoning*, pp. 107–121. Springer, 2010.

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.

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. doi: 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 Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (eds.), *Theorem Proving in Higher Order Logics*, pp. 23–42, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03359-9.

Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.

Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2–3):95–120, 1988. doi: 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 astrée analyzer. In *European Symposium on Programming*, pp. 21–30. Springer, 2005.

Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of automated reasoning*, 61(1):423–453, 2018.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Leonardo De Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe (eds.), *Automated Deduction – CADE 28*, pp. 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.

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. doi: 10.1007/978-3-540-78800-3_24. 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 Adrian Riesco and Min Zhang (eds.), *Formal Methods and Software Engineering*, pp. 90–105, Cham, 2022. Springer International Publishing. ISBN 978-3-031-17244-1.

Adel Djoudi, Martin Hána, and Nikolai Kosmatov. Formal verification of a javacard virtual machine with frama-c. In Marieke Huisman, Corina Păsăreanu, and Naijun Zhan (eds.), *Formal Methods*, pp. 427–444, Cham, 2021. Springer International Publishing. ISBN 978-3-030-90870-6.

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)*. IEEE, 2025. doi: 10.1109/LLM4Code66737.2025.00033. URL `https://www.computer.org/csdl/proceedings-article/llm4code/2025/261500a072/27uerjxJuPC`.

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. doi: 10.1007/s00165-019-00504-0. 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/2019/presentation/journey-to-a-rte-free-x509-parser/`. Talk on 6 June 2019; paper PDF available via the presentation page.

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. doi: 10.1007/978-3-030-17502-3_12. URL `https://doi.org/10.1007/978-3-030-17502-3_12`.

Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner (eds.), *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pp. 125–128. Springer, March 2013.

Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. The 2020 expert survey on formal methods. In Maurice H. ter Beek and Dejan Ničković (eds.), *Formal Methods for Industrial Critical Systems*, pp. 3–69, Cham, 2020. Springer International Publishing. ISBN 978-3-030-58298-2.

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. doi: 10.1007/s10817-020-09580-x. 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. doi: https://doi.org/10.1016/B978-0-444-51624-4.50004-6. URL `https://www.sciencedirect.com/science/article/pii/B9780444516244500046`.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. 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 Wüstholz, 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. doi: 10.1007/978-3-642-21437-0_14. URL https://doi.org/10.1007/978-3-642-21437-0_14.

Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pp. 1–35. Springer, 2013.

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. doi: 10.1145/3586037. 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*, pp. 194–215, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-74776-2.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov (eds.), *Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4.

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://proceedings.neurips.cc/paper_files/paper/2024/file/62c6d7893b13a13c659cb815852dd00d-Paper-Datasets_and_Benchmarks_Track.pdf.

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 Frédéric Cuppens, Nora Cuppens, Jean-Louis Lanet, and Axel Legay (eds.), *Risks and Security of Internet and Systems*, pp. 114–120, Cham, 2017. Springer International Publishing. ISBN 978-3-319-54876-0.

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 Zhou Yuanyuan. Laurel: Unblocking automated verification with large language models. *Proc. ACM Program. Lang.*, 9(OOPSLA1), April 2025. doi: 10.1145/3720499. 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 Alexandra Silva and K. Rustan M. Leino (eds.), *Computer Aided Verification*, pp. 677–689, Cham, 2021. Springer International Publishing. ISBN 978-3-030-81688-9.

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.

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. IEEE, 2009.

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

Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In *CADE*, volume 28, pp. 450–467, 2021.

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, SAIV 2024, Montreal, QC, Canada, July 22-23, 2024, Proceedings*, volume 14846 of *Lecture Notes in Computer Science*, pp. 134–155. Springer, 2024. doi: 10.1007/978-3-031-65112-0_7. URL https://doi.org/10.1007/978-3-031-65112-0_7.

G. Sutcliffe. Stepping Stones in the TPTP World. In C. Benzmüller, M. Heule, and R. Schmidt (eds.), *Proceedings of the 12th International Joint Conference on Automated Reasoning*, number 14739 in Lecture Notes in Artificial Intelligence, pp. 30–50, 2024.

Amitayush Thakur, Jasper Lee, George Tsoukalas, Meghana Sistla, Matthew Zhao, Stefan Zetzsche, 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. doi: 10.1109/ICSE55347.2025.00161. 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. In *Automated Deduction — CADE-28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 415–432. Springer, 2021. doi: 10.1007/978-3-030-79876-5_24.

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 Wischnewski. 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, 2009a. doi: 10.1007/978-3-642-02959-2_10. URL https://doi.org/10.1007/978-3-642-02959-2_10.

Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *International Conference on Automated Deduction*, pp. 140–145. Springer, 2009b.

Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4):1–36, 2009.

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, Haonan Li, David Sanan, and Conrad Watt. Isamini: Redesigned isabelle proof language for machine learning, 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. doi: 10.48550/ARXIV.2409.13082. 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. doi: 10.48550/ARXIV.2502.05344. URL https://doi.org/10.48550/arXiv.2502.05344.

A 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 B 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.

B 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 $l \leq u$, where the VC verifies the loop iterations: if either $a[m] < v$ or $a[m] > v$, the updated boundary $(m + 1, u)$ or $(l, m - 1)$ must preserve the invariant, and the metric $u - l$ must strictly decrease; if the program exits and returns m at line 16, the VC judges whether the return value m

Define $\mathcal{I}(l,u) \triangleq 0 \le l \land u < \text{length}(a) \land (\forall i.\ 0 \le i < \text{length}\,a \land a[i] = v \longrightarrow l \le i \le u)$

```
1  exception NF (* standing for not found *)
2  let binary_search (a: array int) (v: int) : int
3    requires ∀i j. 0 ≤ i ≤ j < length(a) ⟶ a[i] ≤ a[j]
4    ensures  0 ≤ result < length(a) ∧ a[result] = v
5    raises   NF ⟶ ∀i. 1 ≤ i < length(a) ⟶ a[i] ≠ v
6  = let ref l = 0 in
7    let ref u = length a - 1 in
8    while l <= u do
9      invariant I(l, u)
10     variant u − l
11     let m = l + div (u - l) 2 in
12     if a[m] < v then
13         assert ∀i. l ≤ i < m + 1 ⟶ a[i] < v
14         l := m + 1
15     else if a[m] > v then u := m - 1
16     else return m
17   done;
18   raise NF
```

$$
\begin{array}{l}
\forall u\, l.\\
\quad \mathcal{I}(l,u) \land \text{sorted}(a) \longrightarrow\\
\quad\quad \text{if } l \le u \text{ then let } m = l + (u-l)/2 \text{ in}\\
\quad\quad\quad 0 \le m < \text{length}\,a \land\\
\quad\quad\quad (\ \text{if } a[m] < v\\
\quad\quad\quad\quad \text{then } (\forall i.\ l \le i < m+1 \longrightarrow a[i] < v) \longrightarrow\\
\quad\quad\quad\quad\quad\quad 0 \le u-l \land u-(m+1) < u-l\\
\quad\quad\quad\quad\quad \land\, \mathcal{I}(m+1,u)\\
\quad\quad\quad\quad \text{else if } v < a[m]\\
\quad\quad\quad\quad \text{then } 0 \le u-l \land m-1-l < u-l\\
\quad\quad\quad\quad\quad \land\, \mathcal{I}(l,m-1)\\
\quad\quad\quad\quad \text{else } 0 \le m < \text{length}\,a \land a[m] = v)\\
\quad\quad \text{else } (\forall i.\ 0 \le i \land i < \text{length}\,a \longrightarrow a[i] \ne v)
\end{array}
$$

Figure 5: (Left) A Why3 program for binary search, with the functional correctness property in cyan and annotations in orange. (Right) One of the generated VCs for its functional correctness (simplified).

```
exception Not_found

let binary_search (a: array int) (v: int) : int
 requires ∀i j. 0 ≤ i ≤ j < length(a) ⟶ a[i] ≤ a[j]
 ensures  0 ≤ result < length(a) ∧ a[result] = v
 raises   Not_found ∧ ∀i. 0 ≤ i < length(a) ⟶ a[i] ≠ v
= let ref l = 0 in
  let ref u = length a - 1 in
  while l <= u do
    invariant 0 ≤ l ∧ u < length(a)
    invariant ∀i. 0 ≤ i < length(a) ∧ a[i] = v ⟶ l ≤ i ≤ u
    variant u − l
    let m = l + div (u - l) 2 in
    if a[m] < v then
      l := m + 1
    else if a[m] > v then
      u := m - 1
    else
      assert a[m] = v
      return m
  done;
  raise Not_found
```

$$
\begin{array}{l}
(\forall i\, j.\ 0 \le i \le j < \text{length}\,a \longrightarrow a[i] \le a[j]) \longrightarrow\\
\text{let } o_1 = \text{length}\,a - 1 \text{ in}\\
(0 \le 0 \land o_1 < \text{length}\,a)\\
\land\ (\forall i.\ 0 \le i < \text{length}\,a \longrightarrow a[i] = v \longrightarrow 0 \le i \le o_1)\\
\land\ (\forall u\, l.\\
\quad\quad (0 \le l \land u < \text{length}\,a)\\
\quad \land (\forall i.\ 0 \le i < \text{length}\,a \longrightarrow a[i] = v \longrightarrow l \le i \le u)\\
\quad \longrightarrow \text{if } l \le u\\
\quad \text{then let } m = l + (u-l)/2 \text{ in}\\
\quad\quad (0 \le m \land m < \text{length}\,a) \land\\
\quad\quad (\ \text{if } a[m] < v\\
\quad\quad\quad \text{then } (0 \le u-l \land u-(m+1) < u-l)\\
\quad\quad\quad\quad \land (0 \le m+1 \land u < \text{length}\,a)\\
\quad\quad\quad\quad \land\ (\forall i.\ 0 \le i < \text{length}\,a \land a[i] = v\\
\quad\quad\quad\quad\quad\quad \longrightarrow m+1 \le i \land i \le u)\\
\quad\quad\quad \text{else } (0 \le m \land m < \text{length}\,a)\\
\quad\quad\quad\quad \land\ (\text{if } v < a[m]\\
\quad\quad\quad\quad\quad \text{then } (0 \le u-l \land m-1-l < u-l)\\
\quad\quad\quad\quad\quad\quad \land (0 \le l \land m-1 < \text{length}\,a)\\
\quad\quad\quad\quad\quad\quad \land (\forall i.\ 0 \le i \,\text{length}\,a \land a[i] = v\\
\quad\quad\quad\quad\quad\quad\quad \longrightarrow l \le i \land i \le m-1)\\
\quad\quad\quad\quad\quad \text{else } (0 \le m < \text{length}\,a) \land a[m] = v))\\
\quad \text{else } (\forall i.\ 0 \le i < \text{length}\,a \longrightarrow a[i] \ne v))
\end{array}
$$

Figure 6: The original program and the original VC of Fig. 5, without simplification

satisfies the expectation as stated in the `ensures` clause, by replacing the *result* variable in the `ensures` clause with m. At last, the last line in the VC corresponds to line 18, where the VC checks value v does not appear in array a.

Finally, we must emphasize that this VC is simplified for better readability. The original VC is much more complicated (as shown in Fig. 6), where the invariant $\mathcal{I}$ is not defined as a term, duplicated terms abound, and $\land$-connected terms are disordered. This binary search is also one of the simplest cases in program verification, while other VCs can be much more complicated. This represents a gap between competition-style mathematical theorems and VCs: the former are concise but require sophisticated mathematical skills to construct paths towards proofs, whereas VCs require less 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.

C Licensing and Rules of Engagement

Since the VCs in the benchmark are generated from existing projects, the license of our benchmark must be at least the supremum of all their licenses, which is GPL v2, and we indeed choose it.

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 ∼800/790/770 lines of YAML, ∼930/780/970 lines of Python, for Isabelle, Lean, Rocq, respectively).
6. ITP libraries that map Why3 notions into the ITPs' native builtins (in ∼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 ($\texttt{contant}\ arg_1 \cdots arg_n$) 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., `(Why3.length, 1, [Int.int, [Isabelle.length,` arg_0`]])` rewrites (Why3.length l) into Int.int (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 repeate 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 repeate 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 repeate any context nor the statement.

The templates specifically for Goedel-Prover and DeepSeek-Prover are 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.

G Failure Cases

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

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) 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

```
lemma goal10 (a : Memory.addr) (t_1 : Memory.addr -> Z) (t_4 : Memory.addr -> Memory.addr) (t : Z -> Z)
(t_3 : Memory.addr -> Z) (t_2 : Memory.addr -> Z) :
  let a_1 : Memory.addr := Memory.shift a (1 : Z);
  let x : Z := t_1 a_1;
  let a_2 : Memory.addr := Memory.shift a (0 : Z);
  let x_1 : Z := t_1 a_2;
  let x_2 : Z := x * x_1;
  let a_3 : Memory.addr := Memory.shift a (2 : Z);
  let a_4 : Memory.addr := Memory.shift a (3 : Z);
  let a_5 : Memory.addr := t_4 a_3;
  Memory.region (Memory.addr.base a) <= (0 : Z) -> -(2147483648 : Z) <= x_2 -> x_2 <= (2147483647 : Z) ->
  Memory.framed t_4 -> Memory.linked t -> Memory.sconst t_3 -> Cint.is_uint16 x -> Cint.is_uint16 x_1 ->
  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) ->
  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 ->
  A_MemSet.l_memset (Memory.havoc t_2 t_3 (Memory.shift a_5 (0 : Z)) x) a_5 (0 : Z) x = true ->
  Axiomatic12.p_valid_or_empty t (t_4 a_4) (Cint.to_uint32 x_2)
:=
  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
  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
  h_memset
  have h_main : Axiomatic12.p_valid_or_empty t (t_4 a_4) (Cint.to_uint32 x_2) := by
    have h1 : Axiomatic12.p_valid_or_empty t (t_4 a_4) (Cint.to_uint32 x_2) := by
      -- Use the assumption `h_framed` to deduce the required property
      -- This step is hypothetical and depends on the actual definition of `Axiomatic12.p_valid_or_empty`
      -- and how it interacts with `Memory.framed`.
      -- In practice, we might need more specific information or lemmas to make this step rigorous.
      -- For the sake of this example, we suppose that `h_framed` directly implies the conclusion.
      exfalso
      have h2 := h_framed
      have h3 := t_4_linked
      have h4 := t_3_sconst
      have h5 := h_p_valid_memb
      have h6 := h_valid_rd_a3
      have h7 := h_valid_rd_a4
      have h8 := h_valid_rd_a1
      have h9 := h_valid_rd_a2
      have h10 := h_p_valid_or_empty_a5
      have h11 := h_memset
      simp_all [Axiomatic12.p_valid_or_empty, Axiomatic13.p_valid_memb, Memory.framed,
                Memory.linked, Memory.sconst]
      <;>
      (try contradiction) <;>
      (try norm_num at *) <;>
      (try aesop)
      <;>
      (try
        {
          simp_all [Cint.is_uint16]
          <;>
          norm_num at *
          <;>
          omega
        })
    exact h1
  exact h_main
```

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

extraneous closing parenthesis, the term can be parsed. However, it will result in a term in the form of "$\cdots \wedge (0 \leq (1 + \mathit{expr}) - \mathtt{hgt(m1\ \ res1)} \wedge \cdots$ ", which is syntactically valid but semantically incorrect (the height of `res1` is being conjoined with another inequality). What one would expect is instead "$\cdots \wedge (0 \leq (1 + \mathit{expr}) - \mathtt{hgt(m1\ \ res1)}) \wedge \cdots)$", 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-o4-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).
- *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 strong complementarity: a significant number of 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.

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	2	4	4
Data Structure	5	6	3
Calculation	6	7	2
Engineering	6	1	14
Competition	0	3	1
Function	6	12	3
Memory	7	8	4
Loop	4	14	2
Invalid Arg.	6	10	1

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.