LLMs versus the Halting Problem: Revisiting Program Termination Prediction

Anonymous ACL submission

Abstract

Determining whether a program terminates is a central problem in computer science. Turing's foundational result established the Halting Problem as undecidable, showing that no algorithm can universally determine termination for all programs and inputs. Consequently, automatic verification tools approximate termination, sometimes failing to prove or disprove; these tools rely on problem-specific architectures and abstractions, and are usually tied to particular programming languages. Recent success and progress in large language models (LLMs) raises the following question: *can LLMs reliably predict program termination?* In this work, we evaluate LLMs on a diverse set of *C* programs from the Termination category of the International Competition on Software Verification (SV-Comp) 2025. Our results suggest that LLMs perform remarkably well at predicting program termination, where GPT-5 and Claude Sonnet-4.5 would rank just behind the top-ranked tool (using test-time-scaling), and Code World Model (CWM) would place just behind the second-ranked tool. While LLMs are effective at predicting program termination, they often fail to provide a valid *witness* as a proof. Moreover, LLMs performance drops as program length increases. We hope these insights motivate further research into program termination and the broader potential of LLMs for reasoning about undecidable problems.

1 Introduction

Program termination is a central problem in computer science, demonstrated by the Halting Problem (Turing et al., 1936). The undecidability of the Halting Problem proves that no algorithm can universally determine whether a given program will halt. Ensuring program termination is important for software reliability and safety (Avizienis et al., 2004). Non-termination may present a critical software defect that can lead to unbounded resource consumption, loss of system responsiveness, and severe failures in real-world deployments.

Despite the undecidability of the Halting Problem, a variety of verification tools have been proposed over the years to approximate termination analysis. State-of-the-art tools, such as UAutomizer (Heizmann et al., 2023) and PROTON (Metta et al., 2024), use automata over program statements or leverage symbolic execution via CBMC (Kroening and Tautschnig, 2014). These tools often rely on complex, multi-stage problem- and language-specific architectures (for example, see PROTON architecture in Figure 1).

In recent years LLMs have demonstrated impressive capabilities in code generation, reasoning, and problem solving (Zheng et al., 2023; Zhao et al., 2023; Liu et al., 2024; Jiang et al., 2024). Unlike traditional tools, LLMs can be integrated directly into the code generation phase, reason beyond surface-level syntax, and generalize across programming languages. This raises a compelling question: *Can LLMs reliably predict program termination and enhance verification methods and systems?*

Determining whether a program terminates on all inputs is undecidable. Non-termination cannot always be inferred from a finite set of concrete executions, since the infinite behaviors might not be captured by those finite traces. Proving termination requires ruling out all infinite execution paths. Thus, we argue that assessing termination requires LLMs to reason about program structure and control flow beyond pattern matching.

In this work, we asses the reasoning capabilities of LLMs by evaluating their performance on the International Competition on Software Verification (SV-Comp) 2025 Termination category[1] (Beyer, 2012). We cover both open-weights models, i.e., CWM (Copet et al., 2025) and Qwen3-32B (Yang et al., 2025) together with proprietary models, i.e.,

[1] The dataset is publicly available, see Appendix A.

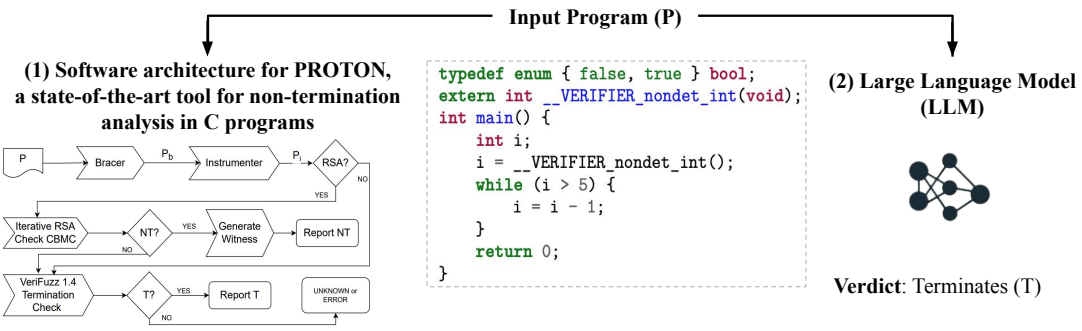

Figure 1: **Termination prediction task.** Given a program, the task is to predict whether the program halts for all inputs (instantiations of nondeterministic variables). **(1)** Traditional verification tools like PROTON use multi-component architectures for parsing, input augmentation, and tool-chain management (its architecture image reproduced from Metta et al. (2024) under Attribution 4.0 International License). **(2)** We investigate whether LLMs can match state-of-the-art verification tools, providing a simpler, language-agnostic solution, as illustrated in the example above where the LLM correctly predicts termination. For an example of non-termination see Figure 2.

GPT-5 (OpenAI, 2025), Claude Sonnet 4.5 (Anthropic, 2025), and GPT-4o (Hurst et al., 2024), which serves as a non-reasoning baseline model. For a reference, we report results for the top three SV-Comp 2025 verification tools: PROTON (Metta et al., 2024), UAutomizer (Heizmann et al., 2023), and AProVE (Emrich et al., 2023).

To further validate our findings, we qualitatively analyze a subset of programs by prompting the model to explicitly specify the input values that cause non-termination. We believe this is a simpler, more interpretable alternative to the SV-Comp witness format, that allows us to probe deeper into the reasoning process of the model.

Results indicate that GPT-5 and Claude Sonnet-4.5 reach performance levels corresponding to second- and third-place, respectively, trailing only the PROTON tool, which placed first. CWM ranks immediately behind UAutomizer, which finished second in SV-Comp. Error analysis indicates that while LLMs are highly effective at predicting program termination, they struggle to provide supporting evidence by constructing valid automaton graphs for infinite paths within the program. Additionally, their performance tends to degrade when handling longer and more complex code samples.

More broadly, termination is one instance of a large class of undecidable problems. By Rice's Theorem (Rice, 1953), every non-trivial semantic property of programs is undecidable, including central verification tasks such as unreachability, memory safety, and program equivalence checking. We hope that our results encourage further research into how large language models can reason about such semantic properties in practice.

2 Problem Formulation

We follow the format specification of the SV-Comp Termination category, in which the **input** is a *C* program with one main function and possibly auxiliary helper functions. The program may include multiple non-deterministic variables, which can be assigned random values during execution.

The **output** is a termination prediction. Specifically, we consider three possible predictions: **(1) T (Terminating)**, meaning the program terminates on all possible input values; **(2) NT (Non-Terminating)**, indicating the program diverges on some values and forms a non-termination loop. Here, the model is required to provide a *witness automaton*, which is a graph containing a cycle of valid states to demonstrate an infinite execution path within the program. Nodes correspond to program states; edges represent transitions.

Importantly, this path does not need to be fully explicit; the model is not required to specify the exact nondeterministic values chosen along the path. As a result, the model does not need to produce a concrete counterexample, but rather a valid infinite path through the program's state space. This allows standard SV-Comp validators to confirm correctness in under a second. See Appendix E for details about UAutomizer, the witness validator we used, and Figure 2 for an example of LLM witness prediction.

Lastly, **(3) UNK (Unknown)** is returned when the model cannot determine the termination behavior.

3 Dataset

We use SV-Comp 2025, which is arguably the largest public collection of formal verification tasks, span-

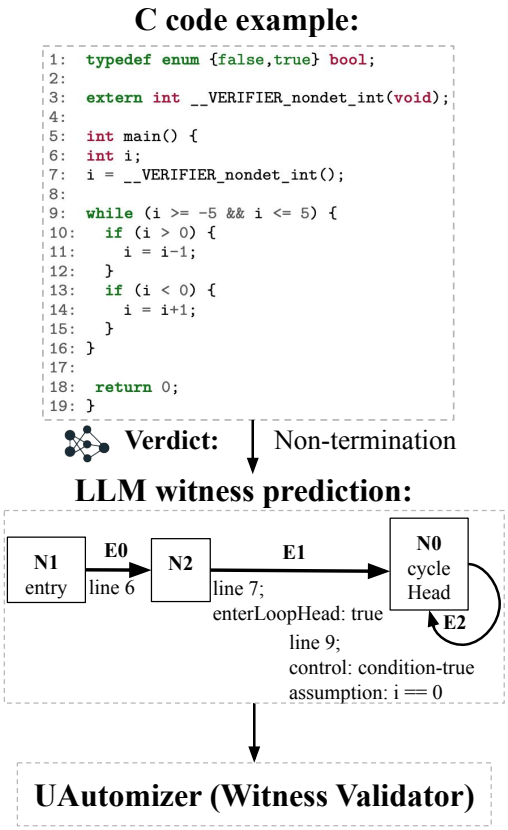

Figure 2: **LLM non-termination witness prediction.** Given a *C* program, an LLM that predicts non-termination must additionally output a **witness automaton** as a proof in JSON format (see Figure 8 in the Appendix). The witness automaton models a potentially infinite execution: nodes correspond to program states and edges to transitions. The predicted JSON is converted to GraphML and validated by a **witness validator** (e.g., UAutomizer). The example illustrates a loop where i is initialized in $[-5, 5]$, eventually reaches 0 (see the *assumption* on edge E2), and executes indefinitely.

ning six key properties for tool evaluation. We focus on the `termination` category resulting in a total of 2, 328 tasks. These tasks are organized into four subcategories: BitVectors (bit-precise arithmetic), MainControlFlow (complex control flow such as loops and recursion), MainHeap (dynamic heap memory manipulation), and Other. Each subcategory presents distinct program analysis challenges. A detailed summary of these subcategories is provided in Table 7 in the Appendix.

Each task in the dataset provides a *C* code and its expected verification result, i.e., whether the code terminates or not. The benchmarks range from synthetic cases targeting specific algorithms to real-world code. Figure 2 illustrates a simple non-termination example from the SV-Comp dataset.

Dataset statistics. The dataset contains *C* input programs of diverse lengths. Using the Instruct Llama3 Tokenizer we find that 90% of samples include maximum of 20K tokens (the median is 6K tokens). See Appendix B for full statistics.

The dataset contains 2,328 samples, with 65.6% labeled as Termination (T) and 34.4% as Non-termination (NT). This distribution is skewed toward terminating programs, reflecting typical real-world software; non-terminating cases often arise from unintended behaviors or bugs, which are also deliberately included in SV-Comp.

4 Experimental Setup

We provide a detailed description of the evaluation setup, covering model inference, evaluation methodology, and task-specific evaluation metrics.

4.1 Model Inference

We evaluate CWM (Copet et al., 2025), Qwen3-32B (Yang et al., 2025), GPT-5 (OpenAI, 2025) and Claude Sonnet-4.5 (Anthropic, 2025), all in reasoning mode. We additionally evaluate GPT-4o (Hurst et al., 2024) as a strong baseline model with no reasoning capabilities.

Inference pipeline. Each model is prompted to generate a JSON object containing a `verdict` attribute: `true` for Termination (T), and `false` for Non-termination (NT). For NT cases, the model also outputs a `witness` attribute, which is a JSON graph (nodes are states in the program; edges the transitions between states) representing a potential infinite path. See Figure D in the Appendix for the complete prompt provided to the model.

For a false *verdict* (NT), the LLM outputs a witness automaton graph in the *witness* JSON attribute. The prediction is validated against a schema: the graph must have unique `nodes` and `edges` ids, it also must contain the required attributes (`id`, `source`, `target`, `line`[2], `sourcecode`). Duplicates or missing fields treated as errors. See Figure 8 in Appendix for an example of predicted witness.

Finally, we convert the validated JSON object into a GraphML representation, which is then processed by a witness validator. Specifically, we use the UAutomizer tool (Heizmann et al., 2023), which symbolically analyzes the candidate infinite path by accumulating constraints and using an SMT solver

[2]To facilitate accurate prediction of the `line` attribute, we added line numbers to each line in the input code.

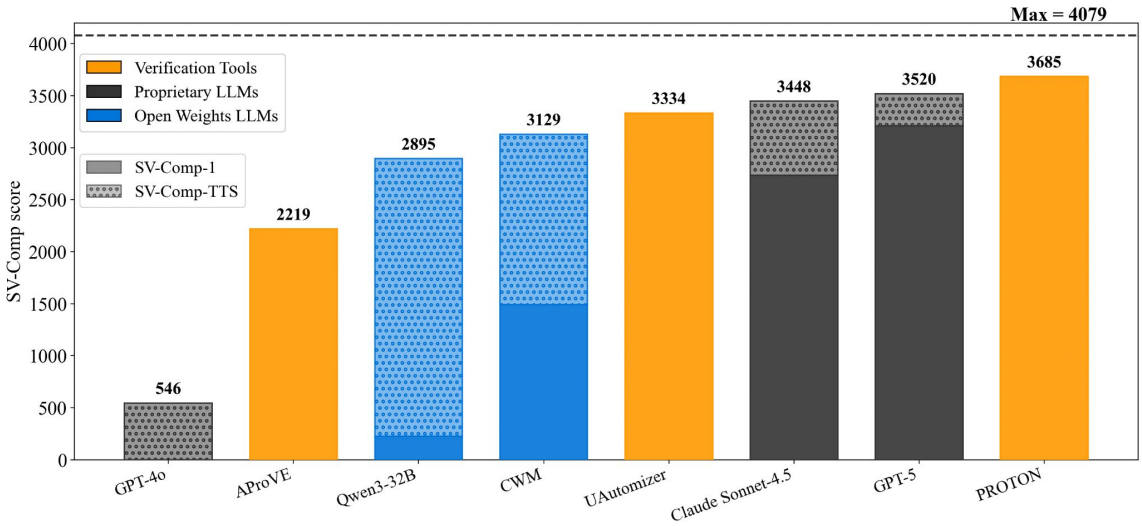

Figure 3: **SV-Comp main results.** Mean SV-Comp scores for LLMs (across 100 bootstraps), top SV-Comp 2025 verification tools, and the max possible scores (**minimum score is** −50,064). GPT-5 (TTS) and Claude Sonnet-4.5 (TTS) would place 2nd and 3rd (3,520 and 3,448), behind the gold medalist PROTON. CWM ranks just below UAutomizer. GPT-4o scores significantly lower: 546 with TTS, and a negative score without TTS (−5, 145).

to validate the feasibility of an infinite execution. See Appendix E for more details on UAutomizer.

4.2 SV-Comp Score

As discussed in Section 3, the SV-Comp Termination category is divided into four categories. The SV-Comp score provides an average measure of performance across all categories, assigning equal weight to each. The score is computed as,

$$\text{Score} = \frac{1}{k}\sum_{i=1}^{k}\left(\frac{s_i}{n_i}\right)\cdot\left(\sum_{i=1}^{k} n_i\right) \tag{1}$$

where s_i is the total score for category i, n_i is the number of samples in category i, k is the total number of categories.

Scoring per sample. We use the asymmetric scoring approach, following the metric used in SV-Comp, with the non-termination (NT) class treated as positive. See the scoring criteria in Table 1.

Correct T predictions (**true negatives (TN)**) receive +2 points. Correct NT predictions with a valid witness (**TP_valid_witness**) receive +1 point, while correct NT predictions with an invalid witness (**TP_invalid_witness**) or **unknown** receive 0 points. Incorrect NT predictions (**false positives (FP)**) are penalized with −16 points, and incorrect T predictions (**false negatives (FN)**) with −32 points.

This scheme prioritizes safety in verification. False negatives (misclassify diverging programs as terminating) can lead to deploying software with infinite loops, risking system failures and resource exhaustion. False positives, by contrast, only prompt unnecessary caution.

Prediction type	Score
Correct Termination (TN)	+2
Correct Non-termination (TP_valid_witness)	+1
Unknown	0
Correct Non-termination (TP_invalid_witness)	0
Incorrect Non-termination (FP)	-16
Incorrect Termination (FN)	-32

Table 1: **SV-Comp scoring per sample.** Non-termination (NT) is the positive class. For NT predictions, scores differ based on witness validity.

4.3 Evaluation

Bootstrap sampling. LLM outputs are inherently stochastic, so repeated runs may yield different predictions. To ensure a robust evaluation, we employ bootstrap sampling. For each instance, we aggregate results across 100 bootstrap runs.

SV-Comp-1. We generate 20 predictions per instance and randomly select one prediction from the 20 generated for each bootstrap iteration.

SV-Comp-TTS. To improve reliability and uncertainty estimates, we apply Test-Time Scaling (TTS) with Consensus Voting. For each instance, we randomly select $n = 10$ out of 20 predictions. If all sampled votes agree (considering `T` and `NT`, while ignoring `unknown`), we adopt the unanimous prediction. If there is any disagreement, we output `unknown` to reflect model uncertainty, reducing the

Model	F1 (T) ↑	F1 (NT) ↑
GPT-5 (TTS)	**0.98**	0.96
GPT-5	**0.98**	**0.97**
Claude S-4.5 (TTS)	0.94	0.89
Claude S-4.5	**0.98**	0.96
CWM (TTS)	0.84	0.63
CWM	0.91	0.80
Qwen3-32B (TTS)	0.83	0.59
Qwen3-32B	0.90	0.78
GPT-4o (TTS)	0.73	0.50
GPT-4o	0.73	0.49

Table 2: **LLMs SV-Comp F1 scores.** F1 (T) and F1 (NT) are averaged over 100 bootstrap runs. F1 decreases under TTS due to increased `unknown` predictions (counted as errors in F1). Rankings align with SV-Comp score.

Model	P ↑	R ↑	V ↑
GPT-5	**40.1%**	**40.7%**	**41.4%**
Claude Sonnet-4.5	38.5%	39.9%	40.7%
CWM	28.5%	25.9%	30.6%
Qwen3-32B	22.6%	17.9%	24.8%
GPT-4o	20.8%	22.7%	27.3%

Table 3: **LLMs witness prediction (validated by UAutomizer) success rates**. Metrics: **P** (Precision): (%) of validated witness NT predictions out of all NT predictions; **R** (Recall): (%) of validated witness NT predictions out of all NT samples; **V** (Validity): (%) of validated witness NT predictions out of all `correct` NT predictions. GPT-5 and Claude lead in prediction rates, while CWM, GPT-4o, and Qwen3-32B perform lower, highlighting the challenge of generating a valid witness.

risk of penalties from incorrect predictions.

We found $n = 10$ as a sweet spot for TTS, representing an optimal trade-off. Lower values of n make consensus easier and reduce `unknown` outputs, while higher values increase the likelihood of disagreement and thus more `unknown` predictions, which can lower the overall score.

Note that requiring agreement between all trajectories is a strict criterion that may lead to excessive `unknown` predictions and reduced overall score. Future work could explore alternative agreement thresholds to optimize performance.

F1 score. Beyond the SV-Comp score, we report per-class F1 scores to enable a more fine-grained analysis of model performance. Unknowns predictions considered as a mistake.

Model hyper-parameters. We configured each model using its official recommended settings. Top-p was set to 0.95 for all models. For CWM and Claude Sonnet 4.5, temperature is set to 1.0 while for Qwen temperature is set to 0.6, and for GPT-4o it is set to 0.7. For GPT-5, temperature control is not supported; we used the official default configuration with `medium` reasoning effort.

5 Results

5.1 Main Results

Results are presented in Figure 3. Under the TTS setup, all reasoning models show impressive SV-Comp scores. Among the LLMs, GPT-5 achieves the highest LLM SV-Comp score (3,520), outperforming UAutomizer and placing second after PROTON. Claude Sonnet-4.5 also surpasses UAutomizer, ranking third (3, 448). CWM (3,129) and Qwen-32B (2,895) follow, and do not exceed UAutomizer. GPT-4o, serving as a non-reasoning baseline, shows substantially lower performance.

While all models benefit from TTS, this benefit is more pronounced in CWM, Qwen-32B, and GPT4o (e.g., Qwen SV-Comp-1 is 225 while its SV-Comp-TTS is 2,895). Interestingly, the gap is the smallest for GPT-5, implying better model calibration and robustness across predictions.

Table 2 presents LLMs performance in terms of F1 score. The F1 ranking aligns with the SV-Comp ranking (e.g., GPT-5 ranks best under both). As expected, the F1 scores for both classes T and NT decrease under TTS, as increasing `unknown` predictions are neutral for SV-Comp scoring but counted as errors in F1 calculation.

F1 score's standard deviations are consistently low (< 0.1), indicating stable classification performance across models, especially top LLMs. SV-Comp score variability decreases with TTS, remaining low for top models and demonstrating robust results. See Appendix H for details.

5.2 Analysis

Next, we analyze models performance across three main axis: (i) ability to predict a valid witness automaton graph; (ii) the frequency of `unknown` predictions, comparing their occurrence in SV-Comp-1 and TTS settings; and (iii) model performance as a function of input code length.

Witness automaton prediction. Table 3 summarizes the results of witness automaton prediction, as validated by the UAutomizer witness validator. Performance is evaluated using three metrics, which correspond to the prediction types defined in Table 1. The first metric, **Validity (V)**, measures the

proportion of correct NT predictions that have a valid witness, relative to all *correct* NT predictions:

$$Validity = \frac{TP_{\text{valid_witness}}}{TP_{\text{valid_witness}} + TP_{\text{invalid_witness}}} \quad (2)$$

Similarly, **Precision (P)** and **Recall (R)** are defined as the proportion of correct NT predictions with a valid witness (out of all NT predictions in precision, and out of all NT samples in recall).

Note that Precision and Recall also account for NT label prediction errors: false positives (FP) in Precision and false negatives (FN) in Recall, which are independent of whether the witness is valid.

GPT-5 achieves the highest witness automaton validated predictions, with Claude performing similarly. CWM, GPT-4o, and Qwen3-32B exhibit lower performance, with CWM providing the best performance among the three.

One might wonder whether models mistakes are derived from wrong prediction content or wrong format generation. To validate that, we compare witness formatting errors against witness validator errors, and find that most errors are content-related ones. GPT-5 and GPT-4o produced no formatting errors, while Qwen and Claude have formatting issues in about 5% of the cases. Interestingly, CWM exhibits formatting errors in roughly 25% of the cases, indicating that improved formatting could enhance its witness prediction performance.

Overall, we conclude that predicting a valid witness automaton is a challenging task for LLMs.

Unknown prediction. Table 4 presents the results for unknown prediction considering both SV-Comp-1 and TTS. GPT-5 and Claude predict unknown least often, 0.5%, 0.3% respectively, followed by Qwen3-32B (2%) and CWM (6%). CWM's higher unknown rate improves its SV-Comp-1 score compared to Qwen. GPT-4o predicts unknown much more frequently (26%), reflecting its difficulty with the program termination classification task.

Under the TTS setup, GPT-5 and Claude have the lowest *unknown* rates (3% and 8%). These results are due to better consensus among model predictions, which implies better and more consistent model predictions. CWM, Qwen3-32B, and GPT-4o show less agreement, resulting in higher *unknown* rates, 22%, 23%, 30%, respectively.

Code inputs length. We hypothesize that model performance is affected by code complexity, which often correlates with input length (longer code inputs are likely to present greater challenge).

Model	Unk ↓	TTS-Unk ↓
GPT-5	0.5%	**3%**
Claude Sonnet-4.5	**0.3%**	8%
CWM	6%	22%
Qwen3-32B	2%	23%
GPT-4o	26%	30%

Table 4: **LLMs unknown prediction distribution. Unk**: (%) of unknown predictions, aggregated over 20 generations per entry. **TTS-Unk**: (%) of cases without unanimous agreement among 10 model predictions, resulting in an unknown. As we can see, GPT-5 and Claude Sonnet-4.5 rarely predict unknown, while GPT-4o does so most often. For TTS-Unk, GPT-5 and Claude show high prediction unanimity, whereas CWM, Qwen3-32B, and GPT-4o have more disagreement – improving SV-Comp scores but lowers F1 scores.

Figure 4 illustrates the mean SV-Comp score as a function of code input length (measured in tokens using Instruct Llama3 Tokenizer), with examples grouped into three `equal size` bins. The results show a clear trend: as code length increases, the mean SV-Comp score decreases.

Among the models, GPT-5 consistently achieves the highest scores, followed by Claude and then CWM. In the medium and long bins, GPT-4o outperforms Qwen3 and matches CWM's performance, although its scores are notably lower in the smallest bin. Interestingly, GPT-4o predicts a relatively high number of unknown outcomes, which helps reduce penalties for longer code inputs.

Next, we examine how code length varies with prediction score. Our analysis reveals that incorrect predictions (FN, FP), as well as *unknown* cases and NT predictions with invalid witnesses, are more prevalent on longer code inputs, while correct predictions (TP, TN) tend to occur on shorter ones. See Appendix F for details. These observations align with the trends discussed above.

5.3 Qualitative Analysis

Since the SV-Comp witness format is rather complex, we propose an interpretable alternative in which models are asked to directly output a logical expression over non-deterministic variables characterizing conditions for non-termination (e.g., $x < 0 \wedge y = 0$ indicates that the program diverges only when x is negative and y equals zero.)

To assess models' ability to predict logical expressions representing non-termination (NT), we manually annotated 40 NT programs from the SV-Comp dataset. Each program is labeled with a

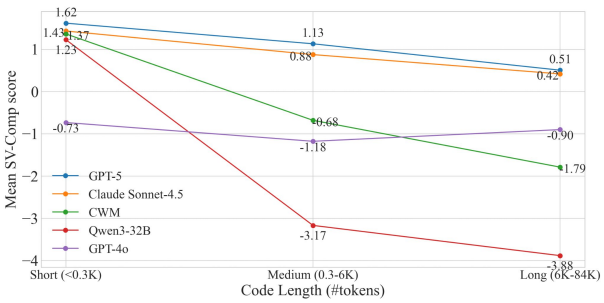

Figure 4: **LLMs mean SV-Comp score vs. code length**. Dataset examples are grouped into three *equal size* bins by code length (measured in tokens using Instruct Llama3 Tokenizer) (**x** axis). The mean SV-Comp score (**y** axis) per bin is shown for all models (20 predictions per sample). Scores decrease with code length; GPT-5 leads, followed by Claude and CWM.

Model	Domain W. ↑		SV-Comp W. ↑	
	P@1	P@3	P@1	P@3
Claude S-4.5	**0.87**	**0.96**	**0.80**	**0.83**
CWM	0.82	**0.96**	0.75	**0.83**
Qwen3-32B	0.85	0.94	0.77	**0.83**

Table 5: **LLMs performance on two types of witness formats.** We report Mean Pass@1 and Pass@3 accuracy for correct witness predictions on 40 SV-Comp NT samples (10 generations each) on these relatively short code samples. All models perform well, with higher accuracy for Domain W.

logical expression for the NT domain.

One annotator labeled all samples, while four others annotated a disjoint subset. Annotators were instructed to *"Read the given C program and specify a logical expression over the non-deterministic variables that characterizes the inputs leading to non-termination."*. All annotators exhibited perfect agreement with the first annotator.

Note that we selected short code examples with at most two non-deterministic variables to keep the annotation process manageable and reliable for human annotators. We observe that the resulting dataset contains simple logical expressions, and acknowledge that more complex expressions could arise in general (e.g., involving equations with higher-order terms or non-linear constraints).

Since model predictions for NT domains may use different (syntactically) but equivalent logical expressions (e.g., $x < 10 \wedge y > -10$ versus $x \leq 9 \wedge y \geq -9$), we use the Z3 Theorem Prover (De Moura and Bjørner, 2008) to automatically verify equivalence between predicted and ground-truth expressions.

Table 5 presents the mean Pass@1 and Pass@3 (Chen, 2021) accuracy for the 40 annotated samples, each evaluated with 10 model predictions. "Domain W." column reports results for our proposed witness format, while "SV-Comp W." corresponds to the graph-based format. See Appendix G for the domain witness prompt.

Both witness approaches demonstrate high accuracy on these relatively short and straightforward code samples. Domain W. consistently achieves higher Pass@1 and Pass@3 scores than SV-Comp W., indicating that it facilitates the generation of valid witnesses. Note that Domain W. is easier to interpret and requires fewer output tokens, making it worth to consider in future work.

Examples of model witness outputs. We find that certain NT samples yield a valid domain witness format but not a valid graph-based format, and vice versa. This suggests that the two formats are complementary: errors in one format do not necessarily occur in the other. See Appendix I for examples comparing both witness formats, with one example also includes the model's reasoning process prior to its final answer.

6 Related Work

LLMs vs. classic reasoners. Kambhampati (2024) examine the capacity of LLMs for reasoning and planning, showing that they underperform classical planners on benchmarks from the International Planning Competition (Valmeekam et al., 2024). They argue that while LLMs may complement traditional planners through their language capabilities, they fall short in both accuracy and efficiency (Kambhampati et al., 2024). Similarly, Hazra et al. (2025) report that reasoning-oriented models yield improvements on SAT tasks but still do not reach the performance of classical solvers.

In the context of theorem proving, prior work mostly integrates LLMs with symbolic reasoning systems such as Lean, SMT solvers, and first-order logic provers-where LLMs generate candidate proofs that are verified by symbolic tools, improving performance and reliability (Sultan et al., 2025; Trinh et al., 2024; Mirzadeh et al., 2024; Régin et al., 2024; Szeider, 2024; Olausson et al., 2023). In contrast, we evaluate the standalone ability of LLMs to reason about program termination, while noting that practical systems should ultimately combine LLMs with symbolic proof checkers.

Termination analysis using LLMs. Most prior

work has focused on predicting program inputs and outputs (Gu et al., 2024; Xu et al., 2025). Recent studies show that jointly training models to predict code and execution traces improves code generation (Armengol-Estapé et al., 2025), and that learning code world models further extends this capability (Copet et al., 2025). Termination prediction goes beyond execution, requiring reasoning about the existence of non-terminating behaviors.

Another line of work has explored using LLMs to synthesize termination arguments, such as ranking functions (Kamath et al., 2024), but, to the best of our knowledge, has not addressed non-termination witnesses. Alon and David (2022) applies GNNs to termination reasoning, using timeouts as a heuristic proxy for non-termination. More recently, Copet et al. (2025) evaluate CWM and Qwen3 on a simplified subset of translated Python functions from SV-Comp and TPDB, [3] showing strong performance.

Termination analysis tools. In SV-Comp 2025, the leading systems are PROTON (Metta et al., 2024) and UAutomizer (Heizmann et al., 2023). PROTON uses CBMC (Kroening and Tautschnig, 2014) to soundly detect non-termination via recurrent loop states and generate witnesses, falling back on high-confidence termination checks from VeriFuzz (Metta et al., 2023). UAutomizer instead encodes programs as automata and applies SMT-based refinement to eliminate infeasible paths.

SV-Comp tools typically target small, self-contained programs. In contrast, Raad et al. (2024) propose a repository-level analysis over millions of lines of code that seeks to establish non-termination, without proving termination. Their approach identifies recurrent sets (Gupta et al., 2008) as non-termination witnesses and leverages bi-abduction (Calcagno et al., 2011) and under-approximation (O'Hearn, 2020) to achieve soundness at scale.

7 Discussion

Decidable vs. Undecidable problems. As discussed in Section 6, prior studies report that LLMs underperform classical solvers on decidable tasks such as SAT and propositional planning. Interestingly, our results suggest that LLMs achieve comparable performance to classical tools on the halting problem, a task that is theoretically more challenging due to its undecidability.

This discrepancy may be partly attributable to differences in benchmark construction: SAT and planning competition benchmarks are largely synthetically generated, whereas SV-Comp has evolved through community contributions informed by the practical experience of verification teams. As a result, SV-Comp more closely reflects naturally occurring verification problems rather than being explicitly designed to be adversarial. Moreover, unlike SAT or propositional planning, the halting problem has not been fully resolved by classical tools, hence leaves more room for improvement.

Additionally, as undecidable problems cannot be solved exhaustively and instead depend on heuristic methods, they may constitute a more natural application domain for LLMs than problems for which classical techniques are already highly effective.

Beyond SV-Comp. While SV-Comp constitutes an important and valuable benchmark, making LLMs useful for program termination requires extending evaluation to more challenging settings. We believe that the ultimate assessment of the practical utility of LLMs in termination analysis should derive from deployment on real-world codebases, for example within continuous integration pipelines. In practical deployments, automatic termination reasoning must handle a continual stream of previously unseen problems and meet the accuracy requirements of real-world engineering, rather than those defined by researcher-curated benchmarks.

8 Conclusions and Future Work

We tackled the foundational problem of program termination, examining whether LLMs can reason about termination and achieve performance comparable to state-of-the-art symbolic verification tools on the SV-Comp termination benchmark.

We empirically show strong performance of LLMs on program termination prediction. GPT-5 and Claude S-4.5 (with TTS) performs just below the top-ranked tool (PROTON), while the Code World Model (CWM) performs just below the second-ranked tool (UAutomizer). Across all models, performance degrades as code length increases, and witness generation remains a notable challenge.

Future work will focus on constructing challenging, real-world benchmarks to advance research on termination reasoning and, more broadly, the role of LLMs in addressing undecidable problems. Another promising direction is exploring neuro-symbolic approaches that combine LLMs with symbolic verification tools.

[3] https://github.com/TermCOMP/TPDB

Limitations

- **Programming Language.** Our experiments were conducted on the SV-Comp dataset, which consists exclusively of C programs. We acknowledge that results may vary for other programming languages.
- **Proprietary LLM (OpenAI GPT-5).** GPT-5 achieves the best performance across models. While this model delivers strong performance, its architecture, training data, and training methodology are proprietary.
- **LLMs are sensitive to prompt phrasing.** LLMs are sometimes sensitive to small changes to the prompts. We acknowledge that results may vary with different prompts.
- **Lab versus competition conditions.** The LLM results reported on in this work were not from official entry to the SV-Comp competition, and exact competition conditions cannot be guaranteed, and this could affect comparison to the scores of SV-Comp entrants. We expect that our LLM results are conservative, as we use one witness validator rather than many. This work was completed after the deadline for SV-Comp 2026; perhaps LLM entries could participate in SV-Comp 2027.

Ethical Considerations

AI Assistants. We used AI assistants for coding support, writing, and rephrasing. All AI-generated outputs were carefully reviewed and edited to ensure alignment with our design goals and to preserve the original intent of our work.

References

Yoav Alon and Cristina David. 2022. Using graph neural networks for program termination. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 910–921. ACM.

Anthropic. 2025. Claude sonnet 4.5.

Jordi Armengol-Estapé, Quentin Carbonneaux, Tianjun Zhang, Aram H Markosyan, Volker Seeker, Chris Cummins, Melanie Kambadur, Michael FP O'Boyle, Sida Wang, Gabriel Synnaeve, et al. 2025. What i cannot execute, i do not understand: Training and evaluating llms on program execution traces. *arXiv preprint arXiv:2503.05703*.

Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

Dirk Beyer. 2012. Competition on software verification: (sv-comp). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 504–524. Springer.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. 2025. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*.

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

Frank Emrich, Jera Hensel, and Jürgen Giesl. 2023. Aprove: Modular termination analysis of memory-manipulating c programs. *arXiv preprint arXiv:2302.02382*.

Alex Gu, Baptiste Rozière, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving non-termination. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 147–158. ACM.

Rishi Hazra, Gabriele Venturato, Pedro Zuidberg Dos Martires, and Luc De Raedt. 2025. Have large language models learned to reason? A characterization via 3-sat phase transition. *CoRR*, abs/2504.03930.

Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele, and Andreas Podelski. 2023. Ultimate automizer and the commuhash normal form: (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 577–581. Springer.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

Adharsh Kamath, Nausheen Mohammed, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2024. Leveraging llms for program verification. In *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, pages 107–118. IEEE.

Subbarao Kambhampati. 2024. Can large language models reason and plan? *Annals of the New York Academy of Sciences*, 1534(1):15–18.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B Murthy. 2024. Position: Llms can't plan, but can help planning in llm-modulo frameworks. In *Forty-first International Conference on Machine Learning*.

Daniel Kroening and Michael Tautschnig. 2014. Cbmc–c bounded model checker: (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Ravindra Metta, Hrishikesh Karmarkar, Kumar Madhukar, R Venkatesh, and Supratik Chakraborty. 2024. Proton: probes for termination or not (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–398. Springer.

Ravindra Metta, Prasanth Yeduru, Hrishikesh Karmarkar, and Raveendra Kumar Medicherla. 2023. Verifuzz 1.4: Checking for (non-) termination (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 594–599. Springer.

Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. 2024. Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models. *arXiv preprint arXiv:2410.05229*.

Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32.

Theo X Olausson, Alex Gu, Benjamin Lipkin, Cedegao E Zhang, Armando Solar-Lezama, Joshua B Tenenbaum, and Roger Levy. 2023. Linc: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. *arXiv preprint arXiv:2310.15164*.

OpenAI. 2025. Gpt-5.

Azalea Raad, Julien Vanegue, and Peter W. O'Hearn. 2024. Non-termination proving at scale. *Proc. ACM Program. Lang.*, 8(OOPSLA2):246–274.

Florian Régin, Elisabetta De Maria, and Alexandre Bonlarron. 2024. Combining constraint programming reasoning with large language model predictions. *arXiv preprint arXiv:2407.13490*.

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366.

Oren Sultan, Eitan Stern, and Dafna Shahaf. 2025. Towards reliable proof generation with llms: A neuro-symbolic approach. *arXiv preprint arXiv:2505.14479*.

Stefan Szeider. 2024. Mcp-solver: Integrating language models with constraint programming systems. *arXiv preprint arXiv:2501.00539*.

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. 2024. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482.

Alan Mathison Turing et al. 1936. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5.

Karthik Valmeekam, Kaya Stechly, and Subbarao Kambhampati. 2024. Llms still can't plan; can lrms? A preliminary evaluation of openai's o1 on planbench. *CoRR*, abs/2409.13373.

Ruiyang Xu et al. 2025. Cruxeval-x: A benchmark for multilingual code reasoning, understanding and execution. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23762–23779.

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

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2).

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372*.

A Reproducibility

The SV-Comp dataset we used is available at: https://gitlab.com/sosy-lab/sv-comp/bench-defs. To obtain the benchmarks, clone the repository. Then, checkout the `svcomp25` tag and initialize submodules (including `sv-benchmarks`). Repository versions used in the competition are listed in Table 4 of the report: https://doi.org/10.1007/978-3-031-90660-2_9. The `benchmark-defs` directory contains definitions for all verifiers. Benchmarks are organized by property (e.g., `Termination`), with sub-categories. We followed the official instructions and excluded 11 invalid tasks (9 from "MainControlFlow", 2 from "Other"), resulting in 2,328 samples.

B Dataset Statistics

The details of these subcategories are summarized in Table 7.

Length distribution. The dataset exhibits significant variability in code length: token counts range from 45 to 96,827, with a mean of μ = 6,080 and a standard deviation of σ = 10,898. See Figure 5 for the code length histogram.

C SV-Comp Witness

According to the SV-Comp specification, when a non-termination is predicted, a `witness` must be provided to encode the proof as a graph structure. In this graph, nodes represent distinct program states, each assigned a unique `id`. Nodes may also include special attributes such as `entry` (marking the entry point) or `cyclehead` (indicating the head of a cycle). Edges denote transitions between states, corresponding to lines of code. Each edge includes its own `id`, the relevant source code line, the `source` and `target` node IDs, the actual `sourcecode` line, and any special properties (e.g., `enterLoopHead`, `control`, `assumption`).

To generate this witness, we prompt the LLM to produce a JSON object that encodes the graph. This JSON is subsequently converted to a GraphML file and validated using a dedicated verifier tool. For a visual illustration, refer to Figure 8.

D SV-Comp Prompt

See Figure 6 for a detailed description of the prompt provided to the LLM, including the termination analysis task instructions and the requirements for witness proofs in SV-Comp format for non-termination predictions. See Figure 7 for few-shot examples given to the LLM. These include one example demonstrating non-termination and another demonstrating termination.

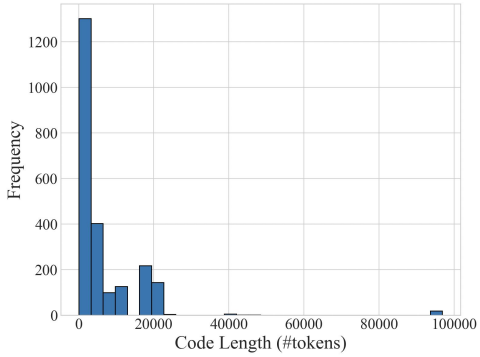

Figure 5: Histogram of input code length in SV-Comp 2025. **X axis:** code length in tokens. **Y axis:** frequency in the data. Almost all code samples have at most $20K$ tokens ($\mu = 6,080$, $\sigma = 10,898$).

E UAutomizer Witness Validator

Ultimate Automizer (UAutomizer) is a powerful verification tool capable of both producing and validating witnesses for program verification tasks. As a witness validator, UAutomizer checks whether a given witness automaton (encoded in GraphML) correctly describes a feasible error path (violation witness) or a proof of correctness (correctness witness) for a program and its specification.

Validation workflow. UAutomizer requires the program source file (e.g., `test.c`), the specification file (in our task, `../properties/termination.prp`), the witness automaton in GraphML format (e.g., `witness.graphml`), and the architecture (32 or 64 bit).

Command-line invocation.

```
./Ultimate.py \
  --architecture 32bit \
  --spec PropertyUnreachCall.prp \
  --file test.c \
  --validate witness.graphml
```

Validation mechanism. UAutomizer parses the witness automaton (GraphML) and interprets its nodes and edges according to the witness specification. It simulates program execution, guided by the automaton. It checks for a feasible path in the program matching the error path(s) described by the automaton, leading to a violation of the specification.

Result interpretation. If UAutomizer finds a matching error path, it outputs `FALSE`, meaning

Method	SV-Comp-1	F1 (T)	F1 (NT)	SV-Comp-TTS@10	F1 (T)	F1 (NT)
GPT-5	**3,211** (± 185)	**.98** (± .001)	**.97** (± .002)	**3,520** (± 47)	**.98** (± .001)	**.96** (± .002)
Claude Sonnet-4.5	2,737 (± 184)	**.98** (± .002)	.96 (± .003)	3,448 (± 49)	.94 (± .003)	.89 (± .005)
CWM	1,491 (± 324)	.91 (± .004)	.80 (± .011)	3,129 (± 80)	.84 (± .003)	.63 (± .008)
Qwen3-32B	225 (± 214)	.90 (± .004)	.78 (± .010)	2,895 (± 84)	.83 (± .003)	.59 (± .005)
GPT-4o	-5,145 (± 750)	.73 (± .004)	.49 (± .006)	546 (± 146)	.73 (± .004)	.50 (± .006)
PROTON	**3,685**	–	–	–	–	–
UAutomizer	3,334	–	–	–	–	–
AProVE	2,219	–	–	–	–	–
Max	4,079	1.0	1.0	4,079	1.0	1.0
Min	-50,064	0.0	0.0	-50,064	0.0	0.0

Table 6: SV-Comp-1, SV-Comp-TTS@10, and the F1 scores are reported as the mean ± standard deviation across 100 bootstrap runs. In F1 score `unknown` predictions treated as errors. We evaluate both proprietary and open-weights state-of-the-art LLMs with `reasoning` mode, and GPT-4o as a baseline without `reasoning`. Additionally, the table lists the top three state-of-the-art verifier participants from the SV-Comp 2025 Termination category, as well as the maximum and minimum SV-Comp score possible. As we can see, GPT-5 and Claude Sonnet-4.5 would score second and third (SV-Comp-TTS@10 of 3,520 and 3,448 respectively), behind the gold medal PROTON verification tool, while the Code World Model (CWM) would place just below UAutomizer, the silver medal winner at SV-Comp. GPT-4o, a non-reasoning baseline, shows much lower performance.

the witness is validated and the program violates the specification as described. If no such path exists, or the automaton does not match the program, it outputs `TRUE`, meaning the witness is rejected.

Interpreting the witness automaton. UAutomizer expects the witness automaton to be encoded in GraphML, with nodes representing control states and edges representing transitions. It reads `<data>` elements attached to nodes and edges, which encode information such as entry/sink/violation states, invariants, assumptions, source code locations, and thread information. The automaton must include specification and architecture information, allowing UAutomizer to match the witness to the correct verification context. See https://github.com/sosy-lab/sv-witnesses/blob/main/README-GraphML.md for reference.

F Code Length Error Analysis

One of the key questions in our error analysis is whether longer code snippets are associated with a higher rate of mistakes across models. Table 8 reports the mean code length for each prediction outcome. Our results show that incorrect predictions (false negatives and false positives) are indeed linked to longer inputs, while correct predictions (true positives and true negatives) are more common for shorter code snippets. Additionally, unknowns and unconfirmed witnesses are also associated with longer code, suggesting that models are more likely to abstain from answering when confronted with complex or lengthy programs.

G Domain Witness Prompt

See Figure 9 for a detailed description of the prompt provided to the LLM for the domain prediction witness settings.

H Main Results

See Table 6 for the main results of LLM evaluation in SV-Comp 2025. The table reports mean scores and standard deviations across 100 bootstrap runs. Notably, the standard deviation for F1 scores across models is consistently low, which indicates stable classification performance, particularly for top-performing LLMs. In contrast, SV-Comp score standard deviations span a much wider range, from 47 up to 750. While leading models demonstrate robust and consistent SV-Comp results with lower variability, mid-tier and baseline models exhibit greater fluctuations and less reliable verification performance.

Notice, in SV-Comp 2025, multiple witness validators are used, and a prediction is accepted if any of them validates it. We consider the UAutomizer validator only, uautomizer, making the LLM's evaluation *stricter* and possibly yielding a lower scores compared to the official competition score.

I Model Witness Responses Examples

In Section 5.3, we sampled 40 short code snippet examples from the `svcomp` dataset and evaluated several LLMs, generating 10 predictions per sample. Our results indicate that LLMs demonstrate

Subcategory	Description	Samples
BitVectors	Programs involving bit-precise arithmetic operations.	34
MainControlFlow	Programs with complex control flow (e.g., loops, conditionals, recursion).	281
MainHeap	Programs that utilize dynamic heap memory allocation and manipulation.	202
Other	Miscellaneous benchmarks related to termination verification.	1, 822
Total	Union over subcategories. We follow SV-Comp 2025 and exclude 11 invalid tasks (9 from "MainControlFlow" and 2 from "Other")	2, 328

Table 7: Distribution of termination verification tasks in SV-Comp 2025 across four subcategories.

strong performance in both the graph-based witness format and the domain witness format. This is expected, as this subset of code examples consists of relatively easy cases. Notably, LLMs consistently achieve higher accuracy in the domain witness format. Interestingly, we observed instances where predictions were correct in the domain witness format but incorrect in the automaton (graph) witness format. These findings suggest that the domain witness and automaton (graph) witness formats are complementary; a mistake in one format does not necessarily imply a mistake in the other. We present examples of CWM predictions. Both domain and graph-based witness predictions are correct (see Figure 10). Domain witness prediction is incorrect, but graph-based prediction is correct (see Figure 11). Domain witness prediction is correct, but graph-based prediction is incorrect (see Figure 12). Lastly, we include another example of correct domain witness prediction, with the goal of showing how the natural language reasoning is align and reflect the final answer (Figure 13).

Program Termination Analysis Prompt

You are an expert code analyzer specializing in program termination analysis. You will be given a C program to analyse. Every line of the program will begin with a line number (e.g., `13: x = x + 1` for line 13), which is for tracking and does not affect compilation. The program may either terminate for all inputs or diverge for some inputs. A function called `__VERIFIER_nondet_X()` can return an arbitrary value of type `X`.

Determine whether the program terminates on all inputs. Your goal is to earn the maximum number of points by answering correctly.

- If the program always terminates, respond with:

```
{
"verdict": true
}
```

  If you are correct, you will earn points.
- If the program can diverge (i.e., may not terminate for some inputs), respond with:

```
{
"verdict": false,
"witness": { ... }
}
```

  and provide a witness demonstrating the control flow graph of the non-termination loop. If you are correct and the witness is valid, you will earn points.
- If your answer is incorrect, you will lose points. The most severe error is predicting termination (`"verdict": true`) when the correct answer is non-termination (`"verdict": false`), which will result in the largest point deduction.
- If you cannot confidently determine whether the program always terminates, it is recommended to respond with:

```
{
"verdict": null
}
```

  This will not earn points but also incurs no penalty.

Witness Generation for Non-Terminating Programs

When the verdict is false (non-terminating program), you MUST include a witness object that represents the control flow graph focused on the infinite loop structure.

JSON Structure (SV-COMP Witness Format):

- **Nodes:** `id`, `entry`, `cyclehead`
- **Edges:** `id`, `source`, `target`, `line`, `sourcecode`, `control`, `assumption`, `enterLoopHead`, `enterFunction`, `returnFrom`

Critical Constraints:

1. Unique edge IDs and line numbers
2. Loop head entries marked with `enterLoopHead: true`
3. Node conventions: N1 is entry, N0 is cyclehead
4. Cycle structure: last edge targets N0
5. Focus on loop, no exit paths
6. Graph flow: N1 → setup → N0 → loop body → N0 (cycle)

Graph Flow Structure:

1. Start at N1 (entry node)
2. Progress through necessary setup
3. Enter the loop head N0
4. Execute loop body
5. Final edge targets N0, creating the infinite cycle

Figure 6: Task requirements for program termination analysis in SV-Comp format.

Model	FN	FP	Unknown	TP_Unconfirmed_Witness	TP	TN
GPT-5	13,492	6,607	6,658	10,356	4,874	4,422
Claude Sonnet-4.5	6,802	6,769	16,144	10,561	4,602	4,367
CWM	11,672	11,534	6,504	9,449	4,717	4,184
Qwen3-32B	7,040	6,451	6,543	10,221	4,387	4,529
GPT-4o	3,887	1,595	11,684	13,160	3,924	1,334
Average	8,579	6,591	9,507	10,749	4,501	3,767

Table 8: Mean input code length (tokens) per score type, aggregated across all models. Incorrect predictions (FN, FP) and unknowns are associated with longer code inputs, while correct predictions (TP, TN) occur on shorter inputs. Unconfirmed witness cases show the highest average code length, suggesting models often abstain or struggle with validation on complex code.

Examples: Example 1: Non-Terminating Program

```
1: extern int __VERIFIER_nondet_int(void);
2: int main() {
3:    int x;
4:    x = __VERIFIER_nondet_int();
5:
6:    while (x % 2 == 0) {
7:       x = x + 2;
8:   }
9:    return 0;
10: }
```

Expected answer:

```
{
"verdict": false,
"witness": {
"nodes": [
{ "id": "N1", "entry": "true" },
{ "id": "N2" },
{ "id": "N0", "cyclehead": "true" },
{ "id": "N3" }
],
"edges": [
{ "id": "E0", "source": "N1", "target": "N2", "line": 3, "sourcecode": "int x;" },
{ "id": "E1", "source": "N2", "target": "N0", "line": 4, "enterLoopHead": true, "sourcecode":
"x = __VERIFIER_nondet_int()" },
{ "id": "E2", "source": "N0", "target": "N3", "line": 6, "control": "condition-true",
"assumption": "x % 2 == 0", "sourcecode": "while (x % 2 == 0) {" },
{ "id": "E3", "source": "N3", "target": "N0", "line": 7, "enterLoopHead": true, "sourcecode":
"x = x + 2" }
]
}
}
```

Example 2: Terminating Program

```
1: extern int __VERIFIER_nondet_int(void);
2: int main() {
3:    int i, j;
4:    j = 0;
5:    for (i = 0; i < 10; i++) {
6:       j = j + 1;
7:   }
8:    return 0;
9: }
```

Expected answer:

```
{
"verdict": true
}
```

Figure 7: Few-shot examples for program termination and non-termination for the SV-Comp prompt.

C code example:

```c
1:  typedef enum {false,true} bool;
2:
3:  extern int __VERIFIER_nondet_int(void);
4:
5:  int main() {
6:  int i;
7:  i = __VERIFIER_nondet_int();
8:
9:  while (i >= -5 && i <= 5) {
10:   if (i > 0) {
11:     i = i-1;
12:   }
13:   if (i < 0) {
14:     i = i+1;
15:   }
16: }
17:
18:  return 0;
19: }
```

LLM witness prediction

```json
{
  "verdict": false,
  "witness": {
    "nodes": [
      { "id": "N1", "entry": "true"},
      { "id": "N2"},
      { "id": "N0", "cyclehead": "true"}],
    "edges": [
      {
        "id": "E0", "source": "N1", "target": "N2",
        "line": 6,
        "sourcecode": "int i;"
      },
      {
        "id": "E1", "source": "N2", "target": "N0",
        "line": 7,
        "enterLoopHead": true,
        "sourcecode": "i = __VERIFIER_nondet_int()"
      },
      {
        "id": "E2", "source": "N0", "target": "N0",
        "line": 9,
        "control": "condition-true",
        "assumption": "i == 0",
        "enterLoopHead": true,
        "sourcecode": "while (i >= -5 && i <= 5) "
      }
    ]
  }
}
```

Figure 8: Example of an LLM-generated witness automaton (graph) in json format.

Domain Witness Prompt
You are given a main function in C programming language that contains a loop or recursion and does not terminate for all inputs.

Note: __VERIFIER_nondet_ defines a non-deterministic value that can be used to generate a random input.

Your task: Predict for which non-deterministic inputs it will not terminate (diverge).

To achieve this: Simulate the execution of the main function step-by-step.

During simulation, detect early signs of divergence based on the logic of the loop.

For example: if a variable used in the loop condition or update becomes positive while another grows unboundedly, or if the loop body increases
a value that keeps the loop condition true indefinitely, this may indicate divergence.

Important:
Do not add arbitrary iteration limits unless strictly necessary for safety – the detection should be logical and correct.

Assume the function uses only basic arithmetic and comparisons; no external state or side effects.

In your answer: Write an expression involving the non-deterministic variables, which describes the conditions under which the function main is guaranteed to diverge.

Please write an expression involving the non-deterministic variables and the logic gates ("and", "or", "not", and so on) for the range of input values causing the program to diverge.

An example would be as follows:

```
typedef enum {false,true} bool;
extern int __VERIFIER_nondet_int(void);
int main() {
int i;
i = __VERIFIER_nondet_int();
while (i != 0) {
if (i < 0) {
i = i+2;
if (i < 0) {
if (i <= -2147483648) return 0;
i = i*(-1);
}
}
} else {
i = i-2;
if (i > 0) {
i = i*(-1);
}
}
}
return 0;
}
```

Expected answer:
(i % 2 != 0) and (i >= -2147483649)

Figure 9: A one-shot example prompt given to the model for the domain witness prompt.

Input *C* code:

```
1: //
2: //
3: //
4:
5: /*
6: *
7: *
8: */
9:
10: extern int __VERIFIER_nondet_int(void);
11:
12: int main() {
13:    int a, x, y;
14:    x = __VERIFIER_nondet_int();
15:    y = __VERIFIER_nondet_int();
16:    a = 0;
17:    while (y > 0) {
18:       if (x < 0) {
19:         y = x̃;
20:       } else {
21:         y = y - 1;
22:       }
23:    }
24:    return 0;
25: }
```

A correct domain witness:

```
x < -1 and y > 0
```

A correct witness automaton prediction:


```
{
"verdict": false,
"witness": {
"nodes": [
{ "id": "N1", "entry": "true" },
{ "id": "N2" },
{ "id": "N3" },
{ "id": "N4" },
{ "id": "N5" },
{ "id": "N0", "cyclehead": "true" },
{ "id": "N6" }
],
"edges": [
{ "id": "E0", "source": "N1", "target": "N2", "line": 13, "sourcecode": "int a, x, y;" },
{ "id": "E1", "source": "N2", "target": "N3", "line": 14, "sourcecode": "x =
__VERIFIER_nondet_int()" },
{ "id": "E2", "source": "N3", "target": "N4", "line": 15, "sourcecode": "y =
__VERIFIER_nondet_int()" },
{ "id": "E3", "source": "N4", "target": "N5", "line": 16, "sourcecode": "a = 0" },
{ "id": "E4", "source": "N5", "target": "N0", "line": 17, "enterLoopHead": true,
"sourcecode": "while (y > 0)" },
{ "id": "E5", "source": "N0", "target": "N6", "line": 18, "control": "condition-true",
"assumption": "x < 0", "sourcecode": "if (x < 0 )" },
{ "id": "E6", "source": "N6", "target": "N0", "line": 19, "enterLoopHead": true,
"sourcecode": "y = x̃" }
]
}
}
```


Figure 10: A sample where CWM produces both a **correct domain witness prediction** and a **correct graph-based prediction** (SV-Comp). The model's output includes natural language reasoning (omitted for brevity) and a JSON answer: the verdict is `false` for NT and `true` for T, with a witness provided for NT cases.

Input *C* code:

```
1: typedef enum {false,true} bool;
2:
3: extern int __VERIFIER_nondet_int(void);
4:
5: int main() {
6:     int i;
7:     i = __VERIFIER_nondet_int();
8:
9:     while (i >= -5 && i <= 5) {
10:         if (i > 0) {
11:             i = i-1;
12:         }
13:         if (i < 0) {
14:             i = i+1;
15:         }
16:     }
17:
18:     return 0;
19: }
```

A wrong domain witness:

```
i = 0
```

The correct domain witness:

```
i >= -5 and i <= 5
```

A correct witness automaton prediction:

```
{
"verdict": false,
"witness": {
"nodes": [
{ "id": "N1", "entry": "true" },
{ "id": "N2" },
{ "id": "N0", "cyclehead": "true" }
],
"edges": [
{ "id": "E0", "source": "N1", "target": "N2", "line": 6, "sourcecode": "int i;" },
{ "id": "E1", "source": "N2", "target": "N0", "line": 7, "enterLoopHead": true, "sourcecode":
"i = __VERIFIER_nondet_int()" },
{ "id": "E2", "source": "N0", "target": "N0", "line": 9, "control": "condition-true",
"assumption": "i == 0", "enterLoopHead": true, "sourcecode": "while (i >= -5 && i <= 5) " }
]
}
}
```

Figure 11: A sample where CWM produces a **wrong domain witness prediction** but a **correct graph-based prediction** (SV-Comp). The model's output includes natural language reasoning (omitted for brevity) and a JSON answer: the verdict is `false` for NT and `true` for T, with a witness provided for NT cases. The witness automaton graph demonstrates a non-terminating loop: the variable i may start at any value from -5 to 5, and after several iterations, i eventually reaches 0 (as indicated by the assumption property in edge E2), causing the loop to continue indefinitely. So the correct domain is $i >= -5$ and $i <= 5$.

Input *C* code:

```
1: typedef enum {false,true} bool;
2:
3: extern int __VERIFIER_nondet_int(void);
4:
5: int main() {
6:     int i;
7:     i = __VERIFIER_nondet_int();
8:
9:     while (i < 0) {
10:         if (i != -5) {
11:             i = i+1;
12:         }
13:     }
14:
15:     return 0;
16: }
```

A correct domain witness prediction:

```
i <= -5
```

A wrong witness automaton prediction:


```
{
"verdict": false,
"witness": {
"nodes": [
{"id": "N1", "entry": "true"},
{"id": "N2"},
{"id": "N0", "cyclehead": "true"},
{"id": "N3"}
],
"edges": [
{"id": "E0", "source": "N1", "target": "N2", "line": 6, "sourcecode": "int i;"},
{"id": "E1", "source": "N2", "target": "N0", "line": 7, "enterLoopHead": true, "sourcecode":
"i = __VERIFIER_nondet_int()"},
{"id": "E2", "source": "N0", "target": "N3", "line": 9, "control": "condition-true",
"assumption": "i < 0", "sourcecode": "while (i < 0) "},
{"id": "E3", "source": "N3", "target": "N0", "line": 11, "enterLoopHead": true, "sourcecode":
"i = i+1"}
]
}
}
```


A correct witness automaton prediction:


```
{
"verdict": false,
"witness": {
"nodes": [
{"id": "N1", "entry": "true"},
{"id": "N2"},
{"id": "N0", "cyclehead": "true"},
{"id": "N3"}
],
"edges": [
{"id": "E0", "source": "N1", "target": "N2", "line": 6, "sourcecode": "int i;"},
{"id": "E1", "source": "N2", "target": "N0", "line": 7, "enterLoopHead": true, "sourcecode":
"i = __VERIFIER_nondet_int()"},
{"id": "E2", "source": "N0", "target": "N3", "line": 9, "control": "condition-true",
"assumption": "i < 0", "sourcecode": "while (i < 0) "},
{"id": "E3", "source": "N3", "target": "N0", "line": 13, "enterLoopHead": true, "sourcecode":
""}
]
}
}
```


Figure 12: A sample where CWM produces a **correct domain witness prediction** but a **incorrect graph-based prediction** (SV-Comp). The model's output includes natural language reasoning (omitted for brevity) and a JSON verdict (true for T, false for NT), with a witness provided for NT cases. Among 10 predictions, a correct witness automaton prediction is also found and shown below the incorrect one.

Input *C* code:

```
1: typedef enum {false,true} bool;
2:
3: extern int __VERIFIER_nondet_int(void);
4:
5: int main() {
6:     int i;
7:     i = __VERIFIER_nondet_int();
8:
9:     while (i < 0) {
10:         if (i != -5) {
11:             i = i+1;
12:         }
13:     }
14:
15:     return 0;
16: }
```

Reasoning and answer: Okay, so I need to figure out for which values of the non-deterministic variable ‘i‘ the given C program will not terminate. Let me look at the code again.The program starts by assigning ‘i‘ a non-deterministic integer value. Then it enters a while loop that continues as long as ‘i‘ is less than 0. Inside the loop, there’s an if statement that checks if ‘i‘ is not equal to -5. If that’s true, it increments ‘i‘ by 1. If ‘i‘ is -5, the loop body does nothing, and the loop condition is checked again.Hmm, so let’s think about what happens in different scenarios.First, if ‘i‘ starts at a value less than -5, say -10. Then, in each iteration, since ‘i‘ is not -5, it gets incremented by 1. So it would go from -10 to -9, then -8, and so on, until it reaches -5. At that point, when ‘i‘ is -5, the if condition ‘i != -5‘ becomes false, so the increment doesn’t happen. The loop condition ‘i < 0‘ is still true because -5 is less than 0, so the loop continues. But now, since ‘i‘ is -5, the if statement doesn’t execute, so ‘i‘ remains -5. This means the loop will keep checking the condition ‘i < 0‘, which is true, and since ‘i‘ doesn’t change, it will loop indefinitely. So any starting value of ‘i‘ that is less than or equal to -5 will cause the program to diverge.Wait, but what if ‘i‘ starts at a value greater than or equal to 0? Then the loop condition ‘i < 0‘ is false from the start, so the loop doesn’t execute at all, and the program terminates immediately. That’s not a problem.What about if ‘i‘ starts between -5 and 0, like -3? Then, in the first iteration, since ‘i‘ is not -5, it gets incremented by 1, becoming -2. Then the loop condition is checked again. Since -2 is still less than 0, the loop continues. Again, ‘i‘ is not -5, so it increments to -1. Then the loop condition is checked again. -1 is less than 0, so loop again. Increment to 0. Now, the loop condition ‘i < 0‘ is false, so the loop exits, and the program terminates. So starting values between -4 and -1 (inclusive) will eventually reach 0 and terminate.But wait, what if ‘i‘ starts at -5? Then, in the first iteration, the if condition ‘i != -5‘ is false, so ‘i‘ doesn’t change. The loop condition ‘i < 0‘ is true, so the loop continues. But since ‘i‘ remains -5, this will loop forever. So starting at -5 is a problem.Similarly, if ‘i‘ starts at any value less than -5, like -6, -7, etc., it will increment by 1 each time until it reaches -5. Once it reaches -5, the increment stops, and the loop continues indefinitely because ‘i‘ is still less than 0. So any starting value less than or equal to -5 will cause divergence.Wait, but let’s test this with an example. Suppose ‘i‘ starts at -6. First iteration: ‘i‘ is -6 (not -5), so increment to -5. Now, ‘i‘ is -5. Next iteration: ‘i‘ is -5, so the if condition is false, no increment. Loop condition ‘i < 0‘ is true, so loop again. ‘i‘ remains -5. This repeats forever. So yes, starting at -6 leads to divergence.Another example: ‘i‘ starts at -5. First iteration: ‘i‘ is -5, so no increment. Loop condition is true, loop again. ‘i‘ remains -5. Diverges.If ‘i‘ starts at -4: first iteration, ‘i‘ is -4 (not -5), increment to -3. Loop condition true. Next iteration, ‘i‘ is -3, increment to -2. Loop condition true. Next, ‘i‘ is -2, increment to -1. Loop condition true. Next, ‘i‘ is -1, increment to 0. Loop condition false, exit. Terminates.So the divergence occurs when the initial value of ‘i‘ is less than or equal to -5. Because in those cases, the loop will eventually reach -5 (if starting below -5) or start at -5, and then get stuck in an infinite loop.Wait, but what if ‘i‘ starts at a value less than -5, but when it increments, it skips over -5? No, because it increments by 1 each time. So if ‘i‘ starts at -6, it goes to -5, then stops incrementing. If it starts at -1000, it will increment 995 times to reach -5, then stop. So any starting value less than or equal to -5 will eventually reach -5 and then loop forever.Therefore, the condition for divergence is ‘i <= -5‘. But wait, let me check the code again. The loop condition is ‘i < 0‘. So when ‘i‘ is -5, the loop continues. But when ‘i‘ is -5, the if statement ‘i != -5‘ is false, so ‘i‘ doesn’t change. So the loop will run indefinitely.Therefore, the program diverges if and only if the initial value of ‘i‘ is less than or equal to -5.But wait, what if ‘i‘ is exactly -5? Then the loop runs once, checks ‘i < 0‘ (true), enters the loop body. Since ‘i‘ is -5, the if condition is false, so ‘i‘ remains -5. Then the loop condition is checked again, still true, and this repeats forever. So yes, ‘i = -5‘ is a diverging case.Therefore, the divergence condition is ‘i <= -5‘.But let me think again. Suppose ‘i‘ is -5. The loop runs, but ‘i‘ doesn’t change. So it’s an infinite loop. If ‘i‘ is less than -5, like -6, then it increments to -5, then gets stuck. So any ‘i <= -5‘ will diverge.Therefore, the answer should be ‘i <= -5‘.Wait, but the problem says to write an expression involving the non-deterministic variables and logic gates. So in this case, the non-deterministic variable is ‘i‘, and the condition is ‘i <= -5‘.But let me check if there’s any other possibility. For example, what if ‘i‘ is a very large negative number, like -2147483648 (assuming 32-bit int). Then, incrementing by 1 each time until it reaches -5. But since integers in C can overflow, but in this case, since we’re dealing with negative numbers and incrementing, it should be okay. Wait, but if ‘i‘ is -2147483648 (the minimum value for a 32-bit signed int), then incrementing it by 1 would result in -2147483647, and so on, until it reaches -5. So even in that case, it would eventually reach -5 and then loop forever. So the condition ‘i <= -5‘ holds.Therefore, the program diverges if and only if the initial value of ‘i‘ is less than or equal to -5.So the answer should be ‘i <= -5‘.Wait, but let me make sure there aren’t any other edge cases. For example, what if ‘i‘ is 0? Then the loop doesn’t run, program terminates. If ‘i‘ is positive, same thing. If ‘i‘ is between -4 and -1, it increments until it reaches 0 and exits. So yes, only ‘i <= -5‘ causes divergence. Therefore, the condition is ‘i <= -5‘.</think>**<answer>i <= -5</answer>**

Figure 13: A sample with a **correct domain witness prediction** by CWM. The output includes a reasoning in natural language, and then the answer, which in this witness format is a logical expression for the domain of non-termination. As we can see, the model can simulate the program execution and infer it will diverge for any $i <= -5$.