Quokka: Accelerating Program Verification with LLMs via Invariant Synthesis

Anjiang Wei [1] **Tianran Sun** [2] **Tarun Suresh** [3] **Haoze Wu** [4] **Ke Wang** [5] **Alex Aiken** [1]

Abstract

Program verification relies on loop invariants, yet automatically discovering strong invariants remains a long-standing challenge. We investigate whether large language models (LLMs) can accelerate program verification by generating useful loop invariants. We introduce Quokka, an evaluation-oriented framework for LLM-based invariant synthesis that provides sound evaluation and achieves state-of-the-art performance. Unlike prior work that treats LLM outputs as noisy symbolic material requiring substantial post-processing, Quokka adopts a simpler and evaluation-centric design that directly validates whether each LLM-generated invariant helps prove the target assertion. We construct a benchmark of 866 instances derived from SV-COMP and evaluate 9 state-of-the-art LLMs across multiple model families. We demonstrate that supervised fine-tuning and Best-of-N sampling yield measurable improvements, and we show that Quokka consistently outperforms prior LLM-based verifiers.

1. Introduction

Program verification aims to provide formal guarantees that software behaves as intended, with applications in many safety-critical domains (Fan et al., 2017; Luckcuck et al., 2019). A long-standing challenge in this area, studied for more than four decades, is the automatic discovery of loop invariants. In this work, we investigate whether large language models (LLMs) can accelerate program verification by generating useful loop invariants.

Loop invariants are conditions that hold before and after each loop iteration, and they are central to deductive program verification. To accelerate program verification, loop invariants must not only be correct but also sufficiently strong to prove the assertions. Generating correct invariants is relatively easy, since any universally true condition qualifies. However, only strong invariants can reduce verification effort and lead to a speedup. For example, in Figure 1, the invariant $x > 0$ is correct but not strong enough to prove the final assertion $x \neq 700$, whereas $x \equiv 3 \pmod 7$ is both correct and sufficiently strong.

Discovering such invariants is difficult and undecidable in general, which has motivated a long line of research. Traditional approaches include constraint solving (Colon et al., 2003; Gupta et al., 2009), dynamic analysis (Le et al., 2019), etc. Since invariant discovery is challenging, researchers have tried a variety of learning-based methods (Li et al., 2017; Ezudheen et al., 2018). Building on this progression, the strong capabilities of LLMs in code generation and program reasoning (Austin et al., 2021; Chen et al., 2021; Wei et al., 2025b) naturally motivate a systematic evaluation of their potential for invariant discovery.

The first work (Pei et al., 2023) to evaluate LLMs for invariant generation has two key limitations. One limitation is that it only checks correctness without assessing whether invariants are strong enough to accelerate verification. Another limitation is that correctness is determined by directly comparing against Daikon (Ernst et al., 2007), a dynamic analysis tool that infers invariants from test executions rather than formal verification. This approach is unsound: Daikon's invariants may themselves be incorrect. Moreover, semantically equivalent invariants (e.g., $a > 0$ vs. $a \geq 1$ for integers) can be incorrectly rejected as wrong. Consequently, prior work cannot reliably evaluate either the correctness or the practical utility of LLM-generated invariants.

A series of follow-up works have proposed LLM-based verifiers. Instead of evaluating LLMs in isolation, these efforts develop verification frameworks powered by LLMs. This line of work predominantly adopts complex algorithms to handle LLM-generated invariants, treating LLM output as noisy symbolic material that must be algorithmically repaired or reassembled. For instance, LaM4Inv (Wu et al., 2024a) uses a "query-filter-reassemble" strategy that filters and combines model-generated predicates through complex logical operations (conjunctions or disjunctions) to

[1]Stanford University [2]Shanghai Jiao Tong University [3]University of Illinois Urbana-Champaign [4]Amherst College [5]Nanjing University. Correspondence to: Anjiang Wei <anjiang@cs.stanford.edu>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

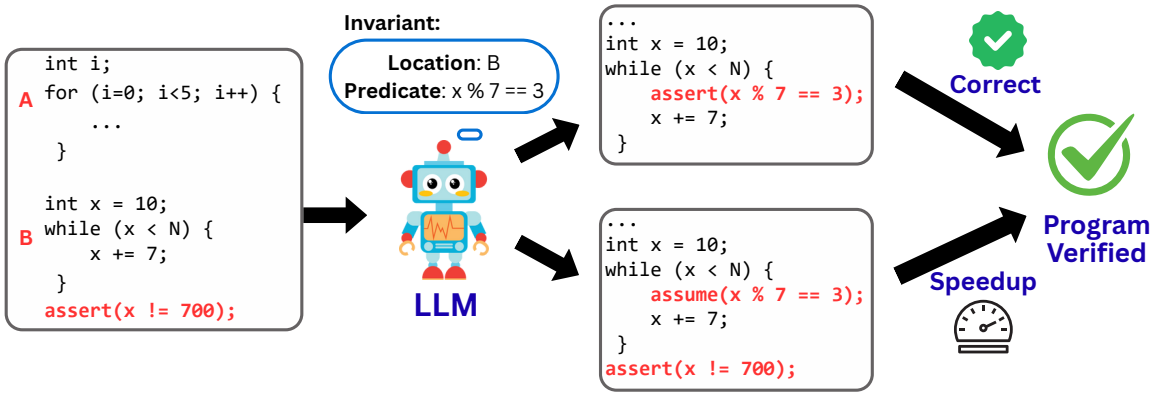

Figure 1. Illustration of Quokka's evaluation pipeline. The LLM proposes an invariant by specifying a program location and predicate (e.g., location B with `x % 7 == 3`). The verification algorithm then incorporates this invariant to prove the property `x != 700` using two verifier queries, and we measure the resulting speedup relative to a baseline without LLM assistance.

construct valid loop invariants, rather than using them directly. Clause2Inv (Cao et al., 2025) extends this idea further. LOOPY (Kamath et al., 2024) employs a customized Houdini algorithm with iterative refinement to filter candidate invariants. LEMUR (Wu et al., 2024b) develops a recursive and backtracking algorithm for fixing invariants. While these approaches demonstrate the potential of LLM-based verification, their complexity raises a fundamental question: as models become increasingly capable, do we really need such elaborate, harness-heavy designs for building LLM-based verifiers? What if we stop rescuing LLM-generated invariants and instead measure directly whether a candidate invariant helps the solver?

In this work, we introduce Quokka, an evaluation-oriented framework for LLM-based invariant synthesis that accelerates program verification. Prior LLM-based verification systems treat LLM outputs as imperfect intermediate artifacts requiring substantial symbolic post-processing—predicate filtering, clause combination, Houdini pruning, repair, or backtracking. In contrast, Quokka adopts a simpler, evaluation-centric design: it directly evaluates LLM-generated invariants by checking whether each invariant is valid and whether it helps prove the target assertion. This design eliminates most customized post-processing logic, enables parallel verification queries, and shifts the objective from merely finding correct invariants to finding invariants that can accelerate verification. Despite its simplicity, Quokka achieves state-of-the-art performance compared to all prior LLM-based verifiers.

To support systematic comparison across solvers and LLMs, we construct a dataset of 866 instances derived from the most recent edition of the software verification competition SV-COMP (Beyer & Strejček, 2025). To the best of our knowledge, this is the largest evaluation dataset for LLM-based verifiers to date. After evaluating 9 LLMs, we find that Quokka proposes a meaningful and challenging setup for LLMs, where model capabilities can be effectively distinguished and there remains significant room for improvement on challenging cases.

Beyond model evaluation, we investigate techniques to enhance invariant generation capabilities. We construct a training dataset of 3589 instances using a verifier-based filtering approach that ensures data quality. Our experiments demonstrate that both supervised fine-tuning and Best-of-N sampling yield measurable improvements in accelerating verification.

In summary, our contributions are as follows:

- We introduce a method for LLM-based invariant synthesis that provides sound evaluation and achieves the best performance results compared with prior LLM-based verifiers.
- We construct a benchmark of 866 instances, and conduct an evaluation of 9 LLMs across multiple model families.
- We construct training data and demonstrate that both supervised fine-tuning and Best-of-N sampling yield measurable improvements in accelerating verification.

2. Related Work

Traditional Methods for Program Invariant Generation. A long line of research has explored invariant synthesis using traditional techniques without machine learning, including model checking (Flanagan & Qadeer, 2002; Lahiri & Bryant, 2007; Hojjat & Rümmer, 2018; Vediramana Krishnan et al., 2024), abstract interpretation (Karr, 1976; Cousot & Cousot, 1977; Cousot & Halbwachs, 1978; Cousot & Cousot, 1979), constraint solving (Gulwani et al., 2009; Gupta et al., 2009), Craig interpolation (Jhala & McMillan, 2006; McMillan, 2010), and syntax-guided synthe-

sis (Fedyukovich & Bodík, 2018). Prior work evaluating LLM-generated invariants (Pei et al., 2023) has relied on Daikon (Ernst et al., 2007), a tool for dynamic invariant detection (Echenim et al., 2019; Le et al., 2019). Daikon executes the program, observes runtime values, and reports properties that consistently hold over the observed executions. However, such invariants may fail to generalize to all possible executions, thereby compromising soundness. Our approach instead employs a verifier-based algorithm relying on UAutomizer (Schüssele et al., 2024) that ensures soundness.

Learning-Based Method for Invariant Generation. Machine learning based techniques have been widely adopted in invariant synthesis, including decision tree (Garg et al., 2014; 2016; Ezudheen et al., 2018; Riley & Fedyukovich, 2022; Xu et al., 2020), support vector machine (Li et al., 2017; Sharma et al., 2012), reinforcement learning (Si et al., 2018; Yu et al., 2023), and others (Sharma et al., 2013; Ryan et al., 2019; Yao et al., 2020). More recently, large language models have demonstrated strong capabilities in reasoning about code and logic (Wei et al., 2025a;b;c), giving rise to a series of work that explore using LLMs for finding invariants. (Pei et al., 2023) is the first pioneering work that evaluates LLMs' capabilities in finding invariants, but it is not a sound evaluation. Various techniques have been proposed to couple LLMs with symbolic solvers, including ranking LLM-generated invariants (Chakraborty et al., 2023), the "query-filter-reassemble" strategy of LaM4Inv (Wu et al., 2024a), the back-tracking algorithm in LEMUR (Wu et al., 2024b), and Loopy's integration of the classic Houdini algorithm (Kamath et al., 2024). On the dataset side, (Liu et al., 2024) introduces a rule-based method for constructing a fine-tuning corpus, which differs from our verifier-based approach. In contrast, our work provides a simple and sound evaluation algorithm for assessing LLM-generated invariants and investigates how both fine-tuning and Best-of-N sampling can enhance LLM performance in invariant synthesis.

3. Method

3.1. Preliminary

We formalize the task of loop invariant synthesis using standard Hoare logic (Hoare, 1969). A Hoare triple $\{P\}\, S\, \{Q\}$ specifies that if the precondition P holds before executing a statement S, then the postcondition Q will hold after its execution. In the context of loops, an invariant I is a logical proposition that summarizes the state of the program at each iteration, and it is the key to proving the validity of Hoare triples involving loops. For a loop of the form `while` B `do` S, the goal of invariant synthesis is to identify a loop invariant I that satisfies the following inference rule:

$$\frac{P \Rightarrow I \quad \{I \wedge B\}\, S\, \{I\} \quad I \wedge \neg B \Rightarrow Q}{\{P\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{Q\}}$$

Here, P is the precondition, Q is the postcondition, B is the loop condition, and S is the loop body. Intuitively, the inference rule requires that the invariant I holds at the beginning of the loop ($P \Rightarrow I$), is preserved across every iteration of the loop body ($\{I \wedge B\}\, S\, \{I\}$), and upon termination ensures the postcondition ($I \wedge \neg B \Rightarrow Q$).

Invariant synthesis amounts to generating a logical summary I that is both *correct*, meaning it can be verified, and *strong*, meaning it enables verification of the final assertion. Weak but correct invariants contribute little, leaving most of the reasoning to the verifier, whereas strong invariants narrow the search space of program states, reduce solver effort, and yield substantial speedups.

3.2. Verifier-Based Approach

We formalize our verifier-based algorithm for assessing candidate invariants. Let P denote a program. A *property* is written as $p = \langle \varphi, \ell \rangle$, where φ is a state predicate and ℓ is a program location. For a finite set A of properties, let $\mathrm{Asm}(P, A)$ be the program obtained from P by inserting `assume` statements for all elements of A. An execution of $\mathrm{Asm}(P, A)$ that reaches a location where an assumption is violated terminates immediately. We write $P \models_A p$ to indicate that all executions of $\mathrm{Asm}(P, A)$ satisfy the assertion p. The notation $P \models p$ abbreviates $P \models_\varnothing p$. Since assumptions restrict behaviors, if $P \not\models_A p$ for some A, then necessarily $P \not\models p$.

We assume access to a verifier

$$V(P, A, p) \in \{\mathbf{T}, \mathbf{F}, \mathbf{U}\},$$

which returns either **T** (proved), **F** (refuted), or **U** (inconclusive). The verifier is required to be sound on conclusive outcomes:

$$V(P, A, p) = \mathbf{T} \ \Rightarrow\ P \models_A p,$$

$$V(P, A, p) = \mathbf{F} \ \Rightarrow\ P \not\models_A p.$$

No completeness is assumed for **U**, which may arise from timeouts or incompleteness of the underlying verifier.

The verification task specifies a target property $p^\star = \langle \varphi^\star, \ell^\star \rangle$. Given P and $p^\star$, a large language model proposes a *candidate invariant* $q = \langle \psi, \ell \rangle$, typically at a loop header. To evaluate the utility of q, the algorithm issues two verifier queries:

$$d_a := V(P, \varnothing, q)$$

$$d_b := V(P, \{q\}, p^\star)$$

The query d_a verifies whether q is a correct invariant, while the query d_b determines whether the target property holds under the assumption that q is true. An illustration of these two verifier queries is provided in Figure 1: d_a corresponds to `assert(x % 7 == 3)` (upper subfigure), and d_b corresponds to verifying the final assertion with the assume statement inserted (lower subfigure). The outcome of the algorithm is expressed as a judgment

$$P \;\Rightarrow\; \langle p^\star, q\rangle \;\Downarrow\; d \qquad \text{with} \quad d \in \{\mathbf{T}, \mathbf{F}, \mathbf{U}\}.$$

The interpretation is as follows: if the judgment yields $\mathbf{T}$, then $p^\star$ is established on P; if it yields $\mathbf{F}$, then $p^\star$ is refuted; and if it yields $\mathbf{U}$, the attempt is inconclusive.

The inference rules defining this judgment are given below. Each rule specifies one possible derivation of the outcome, depending only on the responses of the verifier.

$$\frac{V(P, \{q\}, p^\star) = \mathbf{F}}{P \;\Rightarrow\; \langle p^\star, q\rangle \;\Downarrow\; \mathbf{F}} \text{ (DEC-FALSE)}$$

$$\frac{\begin{array}{c} V(P, \varnothing, q) = \mathbf{T} \\ V(P, \{q\}, p^\star) = d \qquad d \neq \mathbf{F} \end{array}}{P \;\Rightarrow\; \langle p^\star, q\rangle \;\Downarrow\; d} \text{ (DEC-PROP)}$$

$$\frac{V(P, \varnothing, q) \neq \mathbf{T} \qquad V(P, \{q\}, p^\star) \neq \mathbf{F}}{P \;\Rightarrow\; \langle p^\star, q\rangle \;\Downarrow\; \mathbf{U}} \text{ (DEC-U)}$$

Rule DEC-FALSE captures short-circuit refutation: if the goal fails even in the restricted program $\mathrm{Asm}(P, \{q\})$, then it is false on the original program P. Rule DEC-PROP implements the prove-then-use strategy: once the candidate invariant q is established, the outcome is exactly the verifier's answer on the goal under the assumption q, restricted to $d \in \{\mathbf{T}, \mathbf{U}\}$ so as not to overlap with DEC-FALSE. Rule DEC-U gives explicit conditions for inconclusiveness: the goal is not refuted under q and q is not established as an invariant.

Theorem (Decision Soundness). If $P \Rightarrow \langle p^\star, q\rangle \Downarrow \mathbf{T}$ is derivable, then $P \models p^\star$. If $P \Rightarrow \langle p^\star, q\rangle \Downarrow \mathbf{F}$ is derivable, then $P \not\models p^\star$.

Proof. For the $\mathbf{T}$ case, the final rule must be DEC-PROP with $d = \mathbf{T}$. The premises yield $V(P, \varnothing, q) = \mathbf{T}$ and $V(P, \{q\}, p^\star) = \mathbf{T}$. By verifier soundness, $P \models q$ and $P \models_{\{q\}} p^\star$. Since q holds on all executions of P, introducing the assumption $\{q\}$ does not remove executions relevant to $p^\star$; thus $P \models p^\star$. For the $\mathbf{F}$ case, the final rule must be DEC-FALSE. Its premise $V(P, \{q\}, p^\star) = \mathbf{F}$ implies $P \not\models_{\{q\}} p^\star$ by soundness. Assumptions restrict behaviors; hence, a violation under assumptions entails a violation without them, yielding $P \not\models p^\star$. □

This theorem establishes that whenever the calculus derives a conclusive outcome, that outcome is correct. The inconclusive case $\mathbf{U}$ is deliberately conservative: it makes no claim about the truth or falsity of the property and safely captures verifier incompleteness or timeouts.

3.3. Implementation

We describe the implementation of our verifier-based evaluation framework. Given a program P and target property $p^\star$, the system evaluates the generated invariants according to the approach. Each invariant $q = \langle \psi, \ell\rangle$ consists of a program location ℓ and predicate ψ.

Syntactic Validation. Before invoking the verifier, we apply syntactic checks to the generated predicate ψ. These checks ensure that ψ can be safely interpreted as a state predicate and that its insertion as an assumption does not alter the program state. For instance, expressions that update variables (e.g., `a += 1`) are rejected. Only Boolean conditions over the program state are accepted.

Parallel Verifier Queries. For each candidate q, the algorithm issues two verifier queries, namely $d_a = V(P, \varnothing, q)$ to check whether q is an invariant and $d_b = V(P, \{q\}, p^\star)$ to check whether the target holds under the assumption q. These queries are executed in parallel in our implementation, which reduces latency and enables speedup when the proposed invariant is useful for verification. The final outcome is then derived exactly according to the calculus.

3.4. Supervised Fine-Tuning and Best-of-N Sampling

Synthetic Dataset Generation. We prompt GPT-4o (template in Appendix A.1) with three seed programs to synthesize new C programs containing loops and assertions. Seed programs are randomly drawn from SV-COMP (Beyer & Strejček, 2025), disjoint from our evaluation set to prevent data leakage. Generated programs may fail to compile, contain invalid assertions, or include multiple assertions. We split multi-assertion programs into separate instances, each with a single assertion and all loops. We then run UAutomizer on each program and discard non-compilable instances or those with invalid assertions, yielding 3589 synthetic programs.

Extract Invariants Generated from UAutomizer. We extract loop invariants from UAutomizer's output when proving assertions in the synthetic programs. Each invariant includes its program location and predicate. Since programs may contain multiple loops, we pair each program with each of its loop invariants to form our training dataset. An example is shown in Appendix A.3. We perform supervised fine-tuning using LoRA (Hu et al., 2022) on these ground-truth invariants.

Dataset	Avg. #Lines	#Instances	Multi-Loops	Multi-Functions	Arrays/Pointers
LaM4Inv (Wu et al., 2024a)	18	316	✗	✗	✗
Clause2Inv (Cao et al., 2025)	18	366	✗	✗	✗
LEMUR (Wu et al., 2024b)	20	180	✓	✗	✓
Loopy (Kamath et al., 2024)	25	638	✓	✓	✓
Quokka (Ours)	52	866	✓	✓	✓

Table 1. Dataset statistics.

Best-of-N Sampling. Best-of-N sampling generates multiple candidates and selects the most effective one (Ehrlich et al., 2025). We choose the invariant yielding the largest speedup. Since each candidate requires two parallel verifier queries (Section 3.2), Best-of-N evaluates 2N queries concurrently.

4. Experimental Setup

Dataset. We construct our evaluation benchmark from SV-COMP (Beyer & Strejček, 2025), a standard software verification competition, focusing on problems requiring loop invariant synthesis. By filtering instances containing loop keywords (`while` or `for`), we obtain 866 instances. To the best of our knowledge, this is the largest dataset used in prior LLM-based verifier evaluations. As shown in Table 1, our programs are significantly longer than prior work. Manual inspection reveals our dataset contains multiple loops, functions, arrays, and pointers (features largely absent from prior datasets), making it a more challenging benchmark that better distinguishes solver performance. For training data, we construct 3589 instances with synthetic data generation filtered by the solvers, as described in Section 3.4.

Metrics. We evaluate LLMs on two dimensions: (1) correctness of generated invariants and (2) performance improvements. Correctness is verified by the solver with a timeout of $1.2\times$ the original solving time, providing slack to reduce inconclusive outcomes. We report the number of additionally solved instances (unsolvable by the base solver), total solved instances, and average solving time under varying timeout budgets. All measurements represent end-to-end execution time, including LLM inference overhead.

Choice of Solvers. We use UAutomizer (Schüssele et al., 2024), the state-of-the-art non-LLM-based solver, as our base solver. Our methodology requires that the solver can verify programs without externally-provided invariants, enabling speedup measurement relative to a functional baseline. We select UAutomizer over alternatives like Frama-C (Cuoq et al., 2012) and ESBMC (Gadelha et al., 2018) because it represents the current state-of-the-art (outperforming ESBMC on SV-COMP benchmarks) and, unlike Frama-C, has built-in invariant synthesis capabilities for autonomous verification. To demonstrate generality, we also present ESBMC experiments in Appendix A.6.

Models. We benchmark Claude models from Anthropic, GPT models from OpenAI, and models from the Qwen and LLaMA families (Hui et al., 2024; Yang et al., 2025; Touvron et al., 2023).

Hardware and OS. Experiments were conducted on a node with Intel E5-2640 v4 CPUs, 256G main memory, running Ubuntu 20.04 LTS.

5. Results

5.1. Results of LLMs

Table 2 presents the performance of different LLMs when using Quokka. Overall, the table demonstrates substantial variation in both correctness and performance across different models, with larger and more recent models generally achieving better results.

Quokka proposes a meaningful and challenging problem setup for LLMs. The results in Table 2 reveal an interesting performance pattern across timeout budgets. Under the 60s timeout, LLMs solve much more instances beyond the UAutomizer baseline (ranging from 5 additional instances for smaller models to 21 for gpt-5.2), demonstrating their ability to accelerate verification on moderately difficult problems. However, under the 500s timeout, the incremental benefit diminishes significantly, indicating that LLMs struggle to provide meaningful invariants for the hardest problems that require more solving time. This pattern suggests that Quokka establishes a well-calibrated benchmark: LLMs already provide meaningful acceleration on a portion of instances, yet significant room remains for improvement on challenging cases. The substantial variation in correct invariants (342 for Llama-3.1-8B to 710 for gpt-5.2) and average solving times (41.9s to 22.2s under 60s timeout) further demonstrates that the benchmark effectively distinguishes model capabilities. While alternative verification

Model	# Correct Invariants	60s			500s		
		Δ	# Solved	$\overline{T}$ (s)	Δ	# Solved	$\overline{T}$ (s)
Llama-3.1-8B	342	5	321	41.9	0	342	309.9
Qwen2.5-7B	419	5	402	37.6	0	417	267.5
claude-opus-4.1	487	13	466	34.2	1	487	229.4
Qwen2.5-72B	500	11	474	33.8	0	500	221.7
o3	550	17	534	35.9	1	549	198.3
claude-sonnet-4	620	16	591	27.8	1	620	156.0
claude-opus-4.5	689	17	659	24.2	1	689	118.8
gpt-5.1	694	15	661	23.3	1	694	113.8
gpt-5.2	710	21	681	22.2	1	710	105.1

Table 2. Performance comparison of LLMs on invariant generation: number of correct invariants, additional instances solved beyond UAutomizer baseline (Δ), total solved instances, and average solving time ($\overline{T}$) under 60s and 500s timeouts.

Model	# Correct Invariants	60s			500s		
		# Δ	# Solved	$\overline{T}$	# Δ	# Solved	$\overline{T}$
Qwen2.5-7B (base)	419	5	402	37.6	0	417	267.5
Qwen2.5-7B (finetuned)	431	7	414	36.8	0	429	260.8

Table 3. Performance comparison of base and fine-tuned Qwen2.5-7B. Fine-tuning leads to modest improvements in both correctness and speedup metrics.

algorithms might yield different absolute performance numbers, the superiority and simplicity of Quokka over other LLM-based verifiers (discussed in Section 5.4) establishes it as an appropriate framework for evaluating LLMs' invariant generation capabilities.

We provide further analysis of model inference time in Appendix A.5 and evaluate alternative prompting strategies (few-shot and chain-of-thought prompting) in Appendix A.7.

5.2. Failure Mode Analysis

To understand why many invariants fail to accelerate verification in Quokka, we analyze the failure modes of the top three models, with detailed results provided in Appendix A.9. The analysis shows that although some failures are caused by incorrect invariants, the majority are due to timeouts.

We randomly sampled 50 timeout cases and manually analyzed the invariants. Our analysis reveals several key patterns. First, only 7 cases (14%) involved genuinely incorrect invariants, indicating reasonable LLM correctness. Second, 19 cases (38%) generated invariants identical to the original assertion, providing no decomposition benefit, with 4 additional cases nearly identical. Third, 9 cases (18%) generated trivial invariants (e.g., simple bounds) that failed to capture essential loop behavior. Fourth, only 5 cases (10%) produced strong, non-trivial invariants that still timed out, suggesting inherently difficult verification queries. Finally, 6 cases are incorrect original assertions where the LLM generated correct invariants. We provide case studies in Appendix A.10.

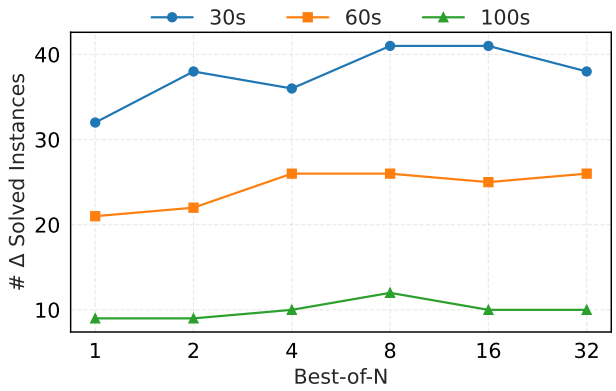

Figure 2. Effect of Best-of-N sampling on the number of Δ instances solved over Uautomizer under different timeout settings.

5.3. Results of Fine-Tuning and Best-of-N Sampling

Modest Improvement from Supervised Fine-Tuning. We perform supervised fine-tuning using LoRA (Hu et al., 2022) on Qwen2.5-7B for 3 epochs. As shown in Table 3,

Method	30s		60s		250s		350s		500s	
	# Δ	$\overline{T}$(s)	# Δ	$\overline{T}$(s)	# Δ	$\overline{T}$(s)	# Δ	$\overline{T}$(s)	# Δ	$\overline{T}$(s)
LaM4Inv	1	30.0	1	59.9	1	249.3	0	348.9	0	498.4
Clause2Inv-original	14	25.8	9	49.9	4	199.2	2	277.4	0	394.4
Clause2Inv-gpt-5.2	20	25.1	13	48.2	6	190.7	3	265.2	1	377.0
Loopy	0	30.0	0	60.0	5	240.0	3	299.5	2	360.4
LEMUR-original	7	28.6	7	50.4	3	165.3	1	222.3	1	306.8
LEMUR-gpt-5.2	15	27.9	13	47.9	7	153.9	4	205.9	2	282.8
Quokka-gpt-5.2 (best-of-8)	41	17.5	25	26.6	7	78.5	4	104.8	3	143.8
Quokka-gpt-5.2 (2-shot)	33	14.3	20	19.4	6	45.2	2	57.6	2	76.0

Table 4. Extra solved instances over UAutomizer and average solving time at different timeout thresholds.

Method	30s		60s		250s		350s		500s	
	# Solved	$\overline{T}$(s)	# Solved	$\overline{T}$(s)	# Solved	$\overline{T}$(s)	# Solved	$\overline{T}$(s)	# Solved	$\overline{T}$(s)
LEMUR-gpt-5.2	4	27.9	9	34.0	18	75.9	19	88.1	20	106.6
Quokka-gpt-5.2	19	9.7	20	11.6	24	28.8	24	28.8	24	28.8

Table 5. Performance comparison on the benchmark used in LEMUR. Since UAutomizer solves 0 instances within ten minutes on this dataset, # Solved equals # Δ (additional instances solved beyond the baseline). Results shown for different timeout thresholds.

fine-tuning yields modest improvements: correct invariants increase from 419 to 431, additionally solved instances increase from 5 to 7 under 60s timeout. These results suggest that domain-specific fine-tuning can improve both correctness and speedup, though gains remain moderate.

Benefits of Best-of-N Sampling. We evaluate Best-of-N sampling by generating multiple invariant candidates from gpt-5.2 at temperature 0.7 and selecting the one yielding the fastest verification time. As shown in Figure 2, increasing N from 1 to 4 generally improves performance. The benefits continue to grow up to N=8, which achieves the best results. Beyond N=8, performance plateaus as resource contention and memory constraints begin to offset the advantages of additional sampling. These results demonstrate that Best-of-N sampling is an effective strategy for improving verification performance, with optimal gains observed at N=8 in our experimental setup.

5.4. Comparison with Prior LLM-based Verifiers

Section 5.1 compares Quokka against prior LLM-based verifiers, reporting additional instances solved beyond UAutomizer and average solving time under different timeout budgets. Figure 3 shows cumulative solved instances over time, demonstrating Quokka's superior performance across all timeout thresholds.

Quokka Outperforms Prior LLM-based Verifiers. Quokka with gpt-5.2 and best-of-8 sampling achieves state-of-the-art results across all speedup metrics, while Quokka with gpt-5.2 and 2-shot prompting also achieves leading performance under most timeout thresholds. Among prior work, LEMUR (Wu et al., 2024b) and Clause2Inv (Cao et al., 2025) represent the strongest baselines. For fair comparison, we upgrade both to gpt-5.2 (from gpt-4 and gpt-4o-mini respectively), creating LEMUR-gpt-5.2 and Clause2Inv-gpt-5.2. Quokka-gpt-5.2 outperforms both upgraded baselines. LaM4Inv (Wu et al., 2024a) and LOOPY (Kamath et al., 2024) show minimal gains over UAutomizer. LaM4Inv's performance differs from its original paper due to dataset differences and its requirement for manual preprocessing (confirmed by the authors), for which no automated script exists.

To further quantify the performance gap at larger timeout, we evaluate on LEMUR's benchmark (Section 5.1)—challenging instances where UAutomizer fails within 10 minutes. Quokka consistently outperforms LEMUR-gpt-5.2 across all timeouts, solving substantially more instances with lower average solving times.

Analysis. The baselines differ in both base solver choices and algorithm design. We analyze base verifier differences in Appendix A.8. Regarding algorithm design, Quokka adopts a distinctly "evaluation-oriented" approach: rather than implementing a handcrafted invariant search algorithm, it provides a clean interface between an LLM and a verifier. This represents a conceptual shift from prior work, which introduces complex algorithms to transform imperfect LLM

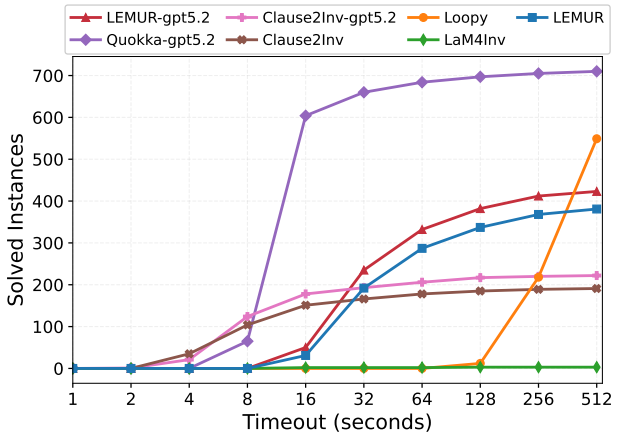

Figure 3. Number of instances solved by different methods over varying timeout budgets.

outputs into usable invariants. Prior approaches treat LLM output as noisy symbolic material requiring algorithmic repair, whereas Quokka treats it as a direct proof candidate to be validated by the verifier. For instance, LaM4Inv requests complete invariants but then filters predicates with model checker and reassembles them, indicating distrust in the raw output. Clause2Inv takes this further. However, as models become increasingly capable, simpler evaluation-centric designs become preferable. We additionally provide a detailed comparison between Quokka and LEMUR in Appendix A.8.

6. Conclusion

This work introduced Quokka, an evaluation-oriented framework for LLM-based invariant synthesis that accelerates program verification. Unlike prior work that treats LLM outputs as noisy symbolic material requiring substantial post-processing, Quokka adopts a simpler and evaluation-centric design that directly validates whether each LLM-generated invariant helps prove the target assertion. We constructed a benchmark of 866 instances derived from SV-COMP and evaluated 9 state-of-the-art LLMs across multiple model families. Our results demonstrate that Quokka consistently outperforms prior LLM-based verifiers. We further showed that supervised fine-tuning and Best-of-N sampling yield measurable improvements in accelerating verification. Our work establishes a new state-of-the-art for LLM-based program verification and provides a principled foundation for future research in this direction.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Beyer, D. and Strejček, J. Improvements in software verification and witness validation: Sv-comp 2025. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 151–186. Springer, 2025.

Cao, W., Wu, G., Xu, T., Yao, Y., Wei, H., Chen, T., and Ma, X. Clause2inv: A generate-combine-check framework for loop invariant inference. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1009–1030, 2025.

Chakraborty, S., Lahiri, S. K., Fakhoury, S., Musuvathi, M., Lal, A., Rastogi, A., Senthilnathan, A., Sharma, R., and Swamy, N. Ranking llm-generated loop invariants for program verification. *arXiv preprint arXiv:2310.09342*, 2023.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Colon, M., Sankaranarayanan, S., and Sipma, H. Linear invariant generation using non-linear constraint solving. *Computer-aided Verification: Proceedings*, pp. 420, 2003.

Cousot, P. and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, 1977.

Cousot, P. and Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 269–282, 1979.

Cousot, P. and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 84–96, 1978.

Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. Frama-c: A software analysis perspective. In *International conference on software engineering and formal methods*, pp. 233–247. Springer, 2012.

Echenim, M., Peltier, N., and Sellami, Y. Ilinva: Using abduction to generate loop invariants. In *Frontiers of Combining Systems: 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings 12*, pp. 77–93. Springer, 2019.

Ehrlich, R., Brown, B., Juravsky, J., Clark, R., Ré, C., and Mirhoseini, A. Codemonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723*, 2025.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

Ezudheen, P., Neider, D., D'Souza, D., Garg, P., and Madhusudan, P. Horn-ice learning for synthesizing invariants and contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

Fan, C., Qi, B., Mitra, S., and Viswanathan, M. Dryvr: Data-driven verification and compositional reasoning for automotive systems. In *International Conference on Computer Aided Verification*, pp. 441–461. Springer, 2017.

Fedyukovich, G. and Bodík, R. Accelerating syntax-guided invariant synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24*, pp. 251–269. Springer, 2018.

Flanagan, C. and Qadeer, S. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 191–202, 2002.

Gadelha, M. R., Monteiro, F. R., Morse, J., Cordeiro, L. C., Fischer, B., and Nicole, D. A. Esbmc 5.0: an industrial-strength c model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 888–891, 2018.

Garg, P., Löding, C., Madhusudan, P., and Neider, D. Ice: A robust framework for learning invariants. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pp. 69–87. Springer, 2014.

Garg, P., Neider, D., Madhusudan, P., and Roth, D. Learning invariants using decision trees and implication counterexamples. *ACM Sigplan Notices*, 51(1):499–512, 2016.

Gulwani, S., Srivastava, S., and Venkatesan, R. Constraint-based invariant inference over predicate abstraction. In *Verification, Model Checking, and Abstract Interpretation: 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings 10*, pp. 120–135. Springer, 2009.

Gupta, A., Majumdar, R., and Rybalchenko, A. From tests to proofs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 262–276. Springer, 2009.

Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Hojjat, H. and Rümmer, P. The eldarica horn solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–7. IEEE, 2018.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Lu, K., et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Jhala, R. and McMillan, K. L. A practical and complete approach to predicate refinement. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 459–473. Springer, 2006.

Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S. K., Lal, A., Rastogi, A., Roy, S., and Sharma, R. Finding inductive loop invariants using large language models. *arXiv preprint arXiv:2311.07948*, 2023.

Kamath, A., Mohammed, N., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S. K., Lal, A., Rastogi, A., Roy, S., and Sharma, R. Leveraging llms for program verification. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 107–118. IEEE, 2024.

Karr, M. Affine relationships among variables of a program. *Acta informatica*, 6(2):133–151, 1976.

Lahiri, S. K. and Bryant, R. E. Predicate abstraction with indexed predicates. *ACM Transactions on Computational Logic (TOCL)*, 9(1):4–es, 2007.

Le, T. C., Zheng, G., and Nguyen, T. Sling: using dynamic analysis to infer program invariants in separation logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 788–801, 2019.

Li, J., Sun, J., Li, L., Le, Q. L., and Lin, S.-W. Automatic loop-invariant generation and refinement through selective sampling. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 782–792. IEEE, 2017.

Liu, C., Wu, X., Feng, Y., Cao, Q., and Yan, J. Towards general loop invariant generation: a benchmark of programs with memory manipulation. *Advances in Neural Information Processing Systems*, 37:129120–129145, 2024.

Luckcuck, M., Farrell, M., Dennis, L. A., Dixon, C., and Fisher, M. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, 52(5):1–41, 2019.

McMillan, K. L. Lazy annotation for program testing and verification. In *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, pp. 104–118. Springer, 2010.

Pei, K., Bieber, D., Shi, K., Sutton, C., and Yin, P. Can large language models reason about program invariants? In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 27496–27520. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/pei23a.html.

Riley, D. and Fedyukovich, G. Multi-phase invariant synthesis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 607–619, 2022.

Ryan, G., Wong, J., Yao, J., Gu, R., and Jana, S. Cln2inv: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542*, 2019.

Schüssele, F., Bentele, M., Dietsch, D., Heizmann, M., Jiang, X., Klumpp, D., and Podelski, A. Ultimate automizer and the abstraction of bitwise operations: (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 418–423. Springer, 2024.

Sharma, R., Nori, A. V., and Aiken, A. Interpolants as classifiers. In *International Conference on Computer Aided Verification*, pp. 71–87. Springer, 2012.

Sharma, R., Gupta, S., Hariharan, B., Aiken, A., and Nori, A. V. Verification as learning geometric concepts. In *International Static Analysis Symposium*, pp. 388–411. Springer, 2013.

Si, X., Dai, H., Raghothaman, M., Naik, M., and Song, L. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems*, 31, 2018.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vediramana Krishnan, H. G., Chen, Y., Shoham, S., and Gurfinkel, A. Global guidance for local generalization in model checking. *Formal Methods in System Design*, 63 (1):81–109, 2024.

Wei, A., Cao, J., Li, R., Chen, H., Zhang, Y., Wang, Z., Sun, Y., Liu, Y., Teixeira, T. S., Yang, D., et al. Equibench: Benchmarking code reasoning capabilities of large language models via equivalence checking. *arXiv e-prints*, pp. arXiv–2502, 2025a.

Wei, A., Suresh, T., Cao, J., Kannan, N., Wu, Y., Yan, K., Teixeira, T. S. F. X., Wang, K., and Aiken, A. CodeARC: Benchmarking reasoning capabilities of LLM agents for inductive program synthesis. In *Second Conference on Language Modeling*, 2025b. URL https://openreview.net/forum?id=Q5pVZCrrKr.

Wei, A., Wu, Y., Wan, Y., Suresh, T., Tan, H., Zhou, Z., Koyejo, S., Wang, K., and Aiken, A. Satbench: Benchmarking llms' logical reasoning via automated puzzle generation from sat formulas. *arXiv preprint arXiv:2505.14615*, 2025c.

Wu, G., Cao, W., Yao, Y., Wei, H., Chen, T., and Ma, X. Llm meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 406–417, 2024a.

Wu, H., Barrett, C., and Narodytska, N. Lemur: Integrating large language models in automated program verification. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=Q3YaCghZNt.

Xu, R., He, F., and Wang, B.-Y. Interval counterexamples for loop invariant learning. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 111–122, 2020.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yao, J., Ryan, G., Wong, J., Jana, S., and Gu, R. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pp. 106–120, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385986. URL https://doi.org/10.1145/3385412.3385986.

Yu, S., Wang, T., and Wang, J. Loop invariant inference through smt solving enhanced reinforcement learning. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 175–187, 2023.

A. Appendix

A.1. Prompt Template Used for Training Dataset Generation

Table A1 shows the prompt template used for synthesizing training programs from seed programs.

You will be shown 3 example C programs. Please gain inspiration from the following programs to create a new high-quality C program. Do not simply copy from any of them.

Requirements for the generated program:
1. The program MUST contain non-trivial loops (for or while).
2. The program MUST contain assertions.
3. The program MUST be compilable, self-contained, and reasonably complex (not trivial or overly short).
4. Only output the new C program.

Example snippets:
Program 1:
{SEED_PROGRAM_1}
Program 2:
{SEED_PROGRAM_2}
Program 3:
{SEED_PROGRAM_3}

Output format: The generated program must be wrapped strictly in the following format:
```c
<NEW_C_PROGRAM>
```

Table A1. Prompt template for synthetic data generation.

A.2. Prompt Template Used for Invariant Generation

Table A2 shows the prompt template used to prompt the LLM to generate a loop invariant for a given C program.

A.3. An Example from the Fine-Tuning Dataset

Figure A1 shows an example from the fine-tuning dataset with the program to the UAutomizer and the generated loop invariant.

The loop invariant holds at Line 6 (the beginning of the loop). It is a disjunction of two clauses, and can be written as $I \equiv P \vee Q = \big((i+1) \bmod 2 = 0 \wedge x < 2+i+y \wedge x < 2+y+2i \wedge x < N+y+1 \wedge 1 \leq N\big) \vee \big(i \bmod 2 = 0 \wedge x < 2+i+y \wedge x < 2+y \wedge 1 \leq N\big)$.

By inspecting the loop, we can derive an exact relationship between x, y, and i at the beginning of each iteration. Since x accumulates all even numbers less than i and y accumulates all odd numbers less than i, we obtain:

$$\text{if } i \text{ is even:} \quad x - y = -\frac{i}{2}, \qquad \text{if } i \text{ is odd:} \quad x - y = \frac{i-1}{2}.$$

Using this relationship, it is straightforward to verify that the invariant I produced by UAutomizer is correct. Moreover, I is strong enough to prove the final assertion $x - y \leq N$.

To show that the assertion holds at loop termination, we can check the following verification condition

$$I \;\wedge\; (i = N) \;\Rightarrow\; (x - y \leq N),$$

where $i = N$ denotes the loop's exit condition.

System prompt:

```
Given a C program with line numbers, you need to identify the loop invariant
to be added at one of the provided points in the program.  A loop invariant
is an expression that is true before, during, and after the execution of a
loop.

You will be provided the set of points in the program where loop invariants
can be added.  Each point will be specified as after line X.

Choose one of the points to insert a loop invariant.  Respond with exactly
this format:
"After line X, insert assume(condition);"

Where:
- X refers to the provided line number where the loop invariant needs to be
added
- condition is the loop invariant condition you want to insert.  The loop
invariant should not modify any variable (no ++, --, +=, -=, =).
- only output one loop invariant
```

User prompt:

```
Program:
{PROGRAM}

Choose one of the points to insert a correct loop invariant at in the
program:
{POINTS}

Please analyze the provided C program and output only where to insert a loop
invariant in the specified format.
```

Table A2. Prompt template used to generate a loop invariant for a given program. The placeholder {PROGRAM} denotes the input C program with line numbers, and {POINTS} denotes the candidate locations where the invariant may be inserted.

Case 1: N is odd. Instantiating the invariant with $i = N$ activates the P disjunct of I, from which we obtain

$$x - y < N + 1.$$

Since $x - y$ is an integer, this directly implies $x - y \leq N$.

Case 2: N is even. In this case, the Q disjunct applies. From the clause $x < 2 + y$ contained in Q, we derive

$$x - y < 2.$$

Because $x - y$ is an integer and $N \geq 1$ and even (hence $N \geq 2$), we conclude

$$x - y \leq 1 \leq N,$$

establishing the desired post-condition.

Thus, the invariant I indeed suffices to prove the final assertion.

A.4. Results of UAutomizer on Prior Datasets

As shown in Table A3, UAutomizer, the state-of-the-art non-LLM-based verifier, consistently solves more instances than LaM4Inv (Wu et al., 2024a) and Loopy (Kamath et al., 2023), two representative LLM-based tools, on their respective

Original Program (Input to UAutomizer)

```
1  int main() {
2    int N = __VERIFIER_nondet_int();
3    assume_abort_if_not(N >= 1 && N < 100);
4    int x = 0, y = 0;
5    for (int i = 0; i < N; i++) {
6        if (i % 2 == 0) {
7            x += i;
8        } else {
9            y += i;
10       }
11   }
12   int diff = x - y;
13   __VERIFIER_assert(diff <= N);
14   return 0;
15 }
```

Loop Invariant Generated by UAutomizer:

```
Line Number: 6
Predicate:
(
(i + 1) % 2 == 0 &&
x < 2 + i + y &&
x < 2 + y + 2 * i &&
x < N + y + 1 &&
1 <= N
)
||
(
i % 2 == 0 &&
x < 2 + i + y &&
x < 2 + y &&
1 <= N
)
```

Figure A1. An example from the fine-tuning dataset: program and its loop invariant generated by UAutomizer.

Verifier	Total Instances	Solved Instances under Different Timeouts			
		10s	100s	300s	600s
LaM4Inv (Wu et al., 2024a)	316	144	286	295	299
UAutomizer	316	299	299	299	299
Loopy (Kamath et al., 2023)	469	0	133	353	403
UAutomizer	469	372	403	411	413
LEMUR (Wu et al., 2024b)	47	2	8	16	19
UAutomizer	47	0	0	0	0

Table A3. Comparison of prior LLM-based verifiers and UAutomizer on their own custom dataset under different timeout budgets.

custom datasets. This demonstrates the importance of base solver choice.

LEMUR (Wu et al., 2024b) is the only tool that surpasses UAutomizer on its own benchmark. This outcome is largely explained by LEMUR's dataset construction methodology: it specifically curates problems that UAutomizer cannot solve within 600 seconds. To assess whether LEMUR's advantage generalizes beyond this carefully selected benchmark, we conduct an additional analysis. We randomly selected 50 instances that UAutomizer failed to solve within 600 seconds and evaluated LEMUR's performance on them. LEMUR successfully solved 12 of these 50 instances within the same 600-second timeout, demonstrating that its gains are not merely an artifact of benchmark design but reflect genuine capability

Model	Avg Generation Time (s)
Llama-3.1-8B	0.300
Qwen2.5-7B	0.355
Qwen2.5-72B	0.578
gpt-5.2	1.076
gpt-5.1	1.373
claude-opus-4.1	1.626
claude-opus-4.5	2.538
claude-sonnet-4	2.746
o3	8.075

Table A4. Average inference time per instance for different LLMs when generating invariants.

Model	# Correct Invariants	60s			500s		
		Δ	# Solved	$\overline{T}$	Δ	# Solved	$\overline{T}$
gpt-5.1	399	31	394	34.8	1	399	272.9
gpt-5.2	455	30	450	31.2	1	455	240.9

Table A5. Performance of different LLMs when using ESBMC as the base verifier.

Model	# Correct Invariants	60s			500s		
		# Δ	# Solved	$\overline{T}$	# Δ	# Solved	$\overline{T}$
gpt-5.2 (CoT)	639	18	609	33.4	2	638	153.5
gpt-5.2	710	21	681	22.2	1	710	105.1
gpt-5.2 (4-shot)	756	20	721	20.1	0	754	81.1
gpt-5.2 (2-shot)	764	20	730	19.4	2	763	76.0

Table A6. Comparison of different prompting strategies for gpt-5.2.

on challenging problems. Nevertheless, for our work, we focus on problems that UAutomizer can solve within the time budget, as our primary objective is to quantify speedup.

A.5. Inference Time of Different Models

Table A4 presents the average inference time per instance for different LLMs when generating invariants. As expected, smaller open-source models such as Llama-3.1-8B and Qwen2.5-7B exhibit the fastest generation times, averaging around 0.3–0.4 seconds per instance. Mid-sized models like Qwen2.5-72B and the GPT-5 series require approximately 0.6–1.4 seconds. The Claude models take longer at around 1.6–2.7 seconds per instance. The o3 model has the highest inference overhead, averaging over 8 seconds per instance. These results highlight the trade-off between model capability and inference cost, which must be considered when evaluating end-to-end verification performance.

A.6. Results for ESBMC as the Base Solver

To demonstrate the generality of Quokka's methodology, we conduct additional experiments using ESBMC (Gadelha et al., 2018) as the base solver instead of UAutomizer. As shown in Table A5, Quokka successfully accelerates ESBMC as well, with both gpt-5.1 and gpt-5.2 generating hundreds of correct invariants and solving additional instances. These results confirm that Quokka is not limited to a specific solver and can be applied to accelerate different verification tools.

A.7. Results for Prompting-Based Methods

We investigate the impact of different prompting strategies on invariant generation quality using gpt-5.2. As shown in Table A6, we compare chain-of-thought (CoT) prompting, few-shot prompting with 2 and 4 examples, and our default zero-shot approach. The results show that few-shot prompting can sometimes improve performance (e.g., 2-shot is generally better than 0-shot), but more examples may also introduce noise or distract the model from the task at hand (e.g., 4-shot

Model	# Incorrect	# Goal Timeout	# Invariant Timeout	# Both Timeout
gpt-5.1	78	96	245	110
claude-sonnet-4	110	160	209	145
gpt-5.2	56	139	230	121

Table A7. Failure mode breakdown for the top three models on Quokka. *Incorrect*: the invariant is refuted by the solver. *Goal Timeout*: the invariant is verified, but proving the assertion under it times out. *Invariant Timeout*: verifying the invariant itself times out. *Both Timeout*: both verification queries time out.

is worse than 2-shot). Chain-of-thought prompting yields the weakest results across all metrics, with only 639 correct invariants and the fewest solved instances. This suggests that explicit reasoning steps may increase generation time without proportional benefits to invariant quality, thereby reducing overall speedup. These findings indicate that few-shot prompting could improve the task of invariant synthesis.

A.8. Detailed Comparison against Prior Work

Comparison of Base Solver Different LLM-based verification frameworks are built on different base solvers: Quokka and LEMUR use UAutomizer, LOOPY uses Frama-C (Cuoq et al., 2012), and LaM4Inv uses ESBMC (Gadelha et al., 2018). The choice of base solver significantly impacts the performance, as stronger solvers provide a better foundation for speedup. Since UAutomizer represents the current state-of-the-art solver, we believe future work should prioritize developing LLM-based verifiers atop state-of-the-art solvers such as UAutomizer. For a direct comparison between UAutomizer and prior LLM-based solvers, we also refer readers to Appendix A.4, where we evaluate UAutomizer on the dataset released by prior works.

Detailed Comparison with LEMUR. To better understand the performance differences between LEMUR and Quokka, we analyze their implementation strategies and examine the instances where Quokka outperforms LEMUR. LEMUR allows LLMs to insert invariants at multiple locations (both before and at the beginning of loops) and generates multiple candidate invariants for each location. It then sequentially evaluates each candidate, verifying both the original assertion and the invariant itself. If verification times out within 30 seconds, LEMUR *recursively* invokes the LLM again to generate additional invariants. In contrast, Quokka adopts a much simpler strategy: it restricts invariant placement to the beginning of loops, asks the LLM to select a single location (when multiple loops exist), and generates only one invariant. The two verification queries (for the assertion and the invariant) are then executed in parallel without recursion or additional LLM invocations. Among the 353 instances that Quokka solves but LEMUR does not, 129 instances (36.5%) contain multiple loops. We hypothesize that for programs with multiple loops, LEMUR's exhaustive search strategy becomes less effective due to the combinatorially larger space of invariant placements and candidates it must explore. In contrast, Quokka's focused approach (i.e., requiring the LLM to commit to a single location and invariant) appears more effective in these complex scenarios. We identify three key advantages of our design. First, parallel execution of verification queries reduces wall-clock time by simultaneously checking both the assertion and the invariant. Second, constraining the LLM to select a single location and invariant forces more deliberate reasoning about which invariant will be most impactful, rather than relying on exhaustive search over many candidates. Third, eliminating recursion and repair loops reduces both implementation complexity and the risk of execution failures (which we observe in LEMUR's logs). Together, these design choices enable Quokka to achieve superior performance while maintaining a simpler and more robust implementation.

A.9. Failure Mode Analysis

As shown in Table A7, we categorize failures into four types: 1) *Incorrect Invariants*: the invariant is refuted by the solver; 2) *Goal Timeout*: the invariant is verified, but proving the assertion under it times out; 3) *Invariant Timeout*: verifying the invariant itself times out; 4) *Both Timeout*: both verification queries time out.

A.10. Case Studies: Verification Timeout in Quokka

To better understand the verification timeouts in Quokka, we manually examined 50 timeout cases and identified five recurring categories. In this section, we select two of these categories for closer analysis and present one representative program from each. The first category includes cases where the LLM proposes a trivial but correct invariant, yet the solver

```
1  int main() {
2      short k;
3      long long y, x, c;
4      k = __VERIFIER_nondet_short();
5      assume_abort_if_not(k >= 0 && k <= 5);
6      assume_abort_if_not(k <= 256);
7
8      y = 0;
9      x = 0;
10     c = 0;
11
12     while (1) {
13         __VERIFIER_assert(-2 * y * y * y * y * y * y - 6 * y * y * y * y * y - 5 * y * y
           ↪  * y * y + y * y + 12 * x == 0);
14
15         if (!(c < k)) {
16             break;
17         }
18         c = c + 1;
19         y = y + 1;
20         x = y * y * y * y * y + x;
21     }
22
23     return 0;
24 }
```

Figure A2. A timeout case where the proposed invariant `c == y` is semantically correct and trivial to prove, yet the solver still cannot establish it within the timeout budget.

still cannot complete verification within the timeout budget. The second category includes cases where the LLM proposes a correct invariant that is almost identical to the original assertion.

Case 1: Trivial and correct invariant. Figure A2 shows a benchmark where the LLM proposes the invariant `c == y` after line 12. This invariant is correct because `c` and `y` are both initialized to zero before the loop and are incremented by exactly one in every iteration, so the equality is preserved throughout the loop. Moreover, it is trivial to prove: its validity follows directly from the initialization and the loop updates, and can be established by a simple inductive argument. Nevertheless, in this case, the solver still cannot prove this invariant within the timeout budget. This example shows that verification timeout may occur even when the proposed invariant is both semantically correct and trivial to prove.

Case 2: Correct invariant as a trivial variant of the original assertion. A different phenomenon is shown in Figure A3, where the LLM proposes the invariant

$$x = \frac{y(y+1)}{2}.$$

This invariant is correct: it holds initially and is preserved by the loop updates. However, it is essentially just a trivial algebraic reformulation of the original assertion

$$y^2 - 2x + y = 0.$$

This possibly suggests a limitation of the LLM in decomposing the verification task into a genuinely simpler intermediate property, instead of merely restating the target assertion in an equivalent form.

```
int main() {
    int k;
    long long y, x, c;
    k = __VERIFIER_nondet_int();
    assume_abort_if_not(k >= 0 && k <= 10);

    y = 0;
    x = 0;
    c = 0;

    while (1) {

        if (!(c < k)) {
            break;
        }

        c = c + 1;
        y = y + 1;
        x = y + x;
    }
    __VERIFIER_assert((y * y) - 2 * x + y == 0);
    return 0;
}
```

Figure A3. A case where the proposed invariant is correct but is essentially a trivial algebraic variant of the original assertion.