Computer Modeling in Engineering & Sciences

Tech Science Press

https://doi.org/10.32604/cmes.2026.083873

ARTICLE

AutoINF: Path-Sensitive Invariant Inference for Multipath Loops

Abeer S. Hadad[1], **Fahman Saeed**[2], **Adeeb A. Ahmed**[3,*] **and Jiangbin Zheng**[1]

[1]School of Software, Northwestern Polytechnical University, Xi'an, China
[2]College of Computer and Information Sciences, Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Saudi Arabia
[3]School of Electro-Mechanical Engineering, Xidian University, Xi'an, China

*Corresponding Author: Adeeb A. Ahmed. Email: adeebalgoidi@gmail.com

Received: 12 April 2026; Accepted: 05 June 2026; Published: 27 July 2026

ABSTRACT: Loop invariant inference is fundamental to program verification, yet it remains particularly challenging for multipath loops, where different execution paths may exhibit incompatible behaviors across feasible executions. In such settings, invariants that are both sound and sufficiently precise often require disjunctive forms, whose automatic inference remains difficult. This paper presents an efficient, path-sensitive, counterexample-guided framework for automated loop invariant inference. Our approach leverages a Path Dependency Automaton (PDA) to systematically decompose the semantics of multipath loops by modeling feasible execution paths independently. Building on this decomposition, we introduce a localized, path-guided Counterexample-Guided Invariant Refinement (CEGIR) process that validates candidate invariants against individual path semantics and uses targeted counterexamples to refine disjunctive and nonlinear polynomial invariant templates only where violations occur. We implement the proposed framework in AutoINF, an automated invariant inference tool for C programs containing multipath loops. The experimental evaluation on a diverse benchmark suite, including programs with complex control flow and nonlinear arithmetic, demonstrates that AutoINF infers sound and expressive invariants for challenging multipath loops.

KEYWORDS: Numerical loop invariant; multipath loops; counterexample-guided refinement; program analysis and verification; software reliability; disjunctive invariants

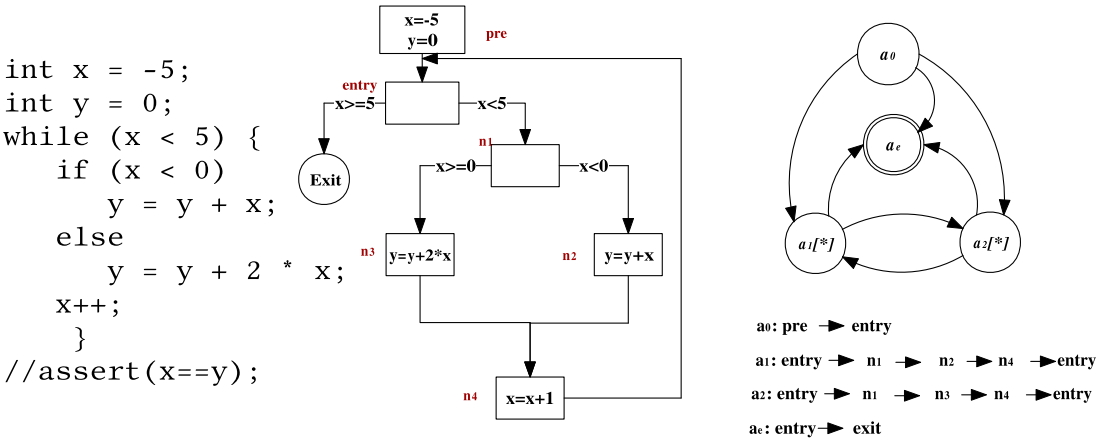
1 Introduction

In safety-critical domains such as aviation, the efficient and consistent verification of program correctness is essential [1,2]. A key aspect of program verification is the loop invariant, a logical property that holds before and after each iteration of the loop. These invariants provide the foundation for reasoning about program behavior, allowing verification frameworks to prove that loops function as intended [3,4]. Automating the inference of such properties further strengthens verification tools by minimizing reliance on manual annotations, which are both time-consuming and prone to human error. This automation supports the scalability of formal verification, making it more applicable to the verification of large and complex systems [5].

Numerical loop invariants can be inferred using either static or dynamic analysis techniques. Static methods [6–9] typically rely on symbolic reasoning and abstract interpretation to derive invariants that hold for all possible program executions. Dynamic analysis approaches, such as Daikon [10–12], infer invariants by analyzing concrete loop executions obtained from a diverse set of test inputs and generalizing patterns among observed variable values. However, these methods derive invariants solely from observed data, which makes them susceptible to violations in unobserved executions and can result in spurious invariants.

Several invariant inference approaches [13–17] adopt a hybrid strategy that combines dynamic invariant inference with formal verification within a Counterexample-Guided Invariant Refinement (CEGIR) loop. Candidate invariants are inferred from execution traces, verified over all inputs, and refined using counterexamples when verification fails. Despite their effectiveness for single-path loops, these CEGIR-based approaches exhibit fundamental limitations in multipath settings. In particular, refinement is typically driven by counterexamples generated from aggregated loop behaviors, without ensuring that candidate invariants are validated and refined with respect to all feasible execution paths. As a result, refinement may focus on a subset of paths while leaving others insufficiently constrained, leading to invariants that are either overly restrictive, fail to converge, or are invalid for certain paths. This lack of path-complete and path-sensitive refinement significantly limits the ability of existing methods to infer disjunctive invariants and accurately capture path-dependent loop semantics. For example, the loop in Fig. 1 requires a disjunctive invariant of the form $(y = x^2 - x - 15) \vee (2y = x^2 - x - 30)$, as different execution paths impose incompatible variable update relations. Existing state-of-the-art invariant inference techniques often struggle to automatically synthesize such disjunctive invariants. Similarly, for the loop in Fig. 2, prior trace-based and CEGIR-style approaches typically infer the invariant $b = 2a$, which holds for only one execution path and fails to characterize the behavior of the alternative path. In both cases, the absence of explicit path-sensitive reasoning prevents existing methods from validating invariants across all feasible execution paths and from inferring the disjunctive forms required to capture multipath behaviors. These examples highlight a fundamental limitation of state-of-the-art approaches in handling multipath loop semantics during invariant inference.

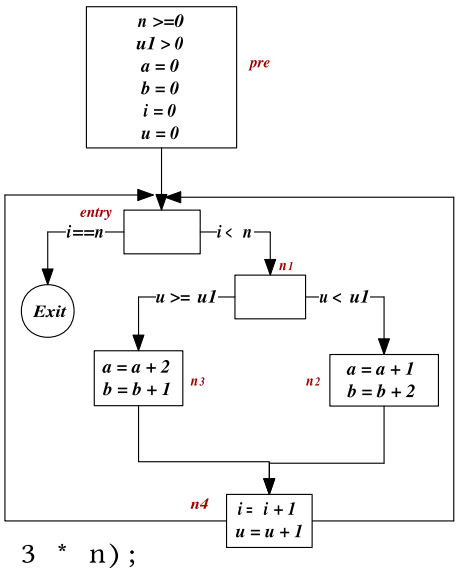
We propose a path-sensitive loop modeling framework for accurately capturing multipath loop semantics and enabling the automated inference of expressive disjunctive invariants. Our approach builds upon the Path Dependency Automaton (PDA) architecture, originally introduced for loop summarization [18,19] and later extended to multipath invariant generation [20]. While Ref. [20] demonstrates the utility of PDAs for separating execution paths, their method remains a forward, speculative invariant synthesis procedure based on a constraint propagation operator (spk) and heuristic mutations. As a result, it lacks a systematic mechanism for revising or refining candidate invariants when verification fails.

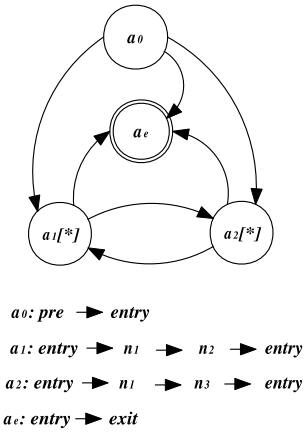


(**a**) Corresponding C Code (**b**) Control Flow Graph and PDA.

Figure 1: Example of a multipath loop with a disjunctive invariant.

```
//assert(n >= 0&& u1 > 0);
int a = 0;
int b = 0;
int i = 0;
int u = 0;
while (i < n) {
 if (u < u1) {
   a = a + 1;
   b = b + 2;
 }
 else {
    a = a + 2;
   b = b + 1;
 }
 i = i + 1;
 u = u + 1;
    }
//assert(a + b == 3 * n);
```





(**a**) Corresponding C Code (**b**) Control Flow Graph and PDA.

Figure 2: Example of a multipath loop with a non-disjunctive invariant.

In this work, we significantly extend this foundational PDA framework by embedding it within a counterexample-guided architecture. We present AutoINF, which integrates a path-sensitive PDA model into a CEGIR loop. During synthesis, path-specific execution behaviors are isolated through individual PDA states, enabling precise instantiation of polynomial invariant templates. During verification, AutoINF constructs path-specific verification conditions and discharges them using a Satisfiability Modulo Theories (SMT) solver. When a candidate invariant is determined to be unsound, counterexamples produced by the solver are mapped back to the corresponding violating PDA states, enabling localized and iterative refinement. This integration allows our framework to systematically reason about complex, nonlinear and disjunctive invariants that are difficult to capture using purely forward or heuristic approaches.

The main contributions of this paper are as follows:

- We introduce a path-sensitive loop modeling framework based on PDA that explicitly represents each execution path and its associated control-flow and variable update semantics.
- We develop AutoINF, a CEGIR inference framework that uses a unified path-sensitive representation for both invariant synthesis and formal verification.
- The proposed approach enables the automated inference of expressive numerical invariants, including nonlinear and disjunctive forms, by generating and reasoning over path-specific traces and constraints.
- We ensure soundness by validating inferred invariants against all feasible execution paths using SMT-based path-sensitive verification, thereby avoiding spurious or path-incomplete invariants.

The remainder of this paper is organized as follows. Section 2 review the related studies. Section 3 provides background on loop fundamentals, including the definition of invariants and key loop features. Section 4 presents an overview of our path-sensitive loop model. Section 5 details the design and construction of AutoINF. The experimental setup and benchmark descriptions are provided in Section 6. Section 7 analyzes and discusses the experimental results. Threats to validity are examined in Section 8. Finally, we conclude and summarize the study in Section 9.

2 Related Work

Several studies have been proposed using different levels of automation to infer loop invariants. We categorize these works into distinct groups based on the approach employed:

2.1 Static Analysis

Abstract Interpretation: Based on the traditional abstract interpretation [6,7], several works have proposed a general approach that has computed the approximations of program semantics under certain abstract domain [8,9]. Instead of an invariant inference problem, Refs. [21–25] have thought through the constraint of the program that has defined a template that generalizes the invariants. However, abstract interpretation-based approaches face several key limitations. Their precision is largely constrained by the choice of abstract domain, which determines the expressiveness of the inferred invariants. Simple domains such as intervals or octagons yield coarse approximations, while more expressive domains like polyhedra or nonlinear constraints are computationally expensive and often impractical for large or complex programs. Moreover, most abstract interpretation frameworks are path-insensitive, meaning that multiple control-flow paths are merged into a single abstract state. This causes the loss of important path-specific relationships among variables, making it difficult to reason about multipath loops or infer disjunctive and nonlinear invariants.

Symbolic Execution: Symbolic execution is a powerful approach commonly used to infer loop invariants [17,26–29] by systematically exploring all loop execution paths using symbolic values. In the symbolic execution-based approach, the loop's structure has been analyzed by symbolically tracking the variables' behavior and determining how these symbols evolve through each iteration and conditional branch. Then, solve the constraints derived from the symbolic execution to identify invariants. While effective for single-path or small loops, symbolic execution faces significant challenges when dealing with multipath loops. First, the path explosion problem becomes critical, as the number of execution paths grows exponentially with nested conditionals and branching, making it infeasible to explore all paths. Second, constraints derived from multiple paths may interact in complex ways, especially when disjunctive or path-dependent invariants are required, which standard symbolic execution struggles to infer. Third, verifying that an invariant holds across all paths can be computationally expensive, leading to timeouts or incomplete analysis in practice. These limitations make traditional symbolic execution insufficient for scalable and precise invariant inference in multipath loops.

2.2 Dynamic Analysis

Daikon-Based: Early work, such as Daikon [11], demonstrated that program specifications can be inferred by observing concrete execution states. Daikon uses a template-driven approach, where a predefined set of invariant patterns is matched against observed traces. To maintain tractability, the template set is relatively limited, which constrains the ability to infer more expressive invariants. One key limitation is that Daikon relies solely on observed executions, so unexecuted paths can lead to spurious invariants, particularly in loops with multiple execution paths. Additionally, Daikon is not equipped to infer disjunctive invariants or handle path-specific variable interactions, which are common in multipath loops. Subsequent approaches, such as iDiscovery [30], extend Daikon templates and use symbolic execution to validate candidate invariants. However, these methods inherit the same limitations and remain unable to fully capture nonlinear or disjunctive properties across all loop paths.

DIG: The DIG framework [31–33] focuses on numerical invariant inference and extends beyond Daikon by supporting richer template classes, such as nonlinear polynomial equalities and octagonal constraints. This allows DIG to infer more expressive numerical relationships. However, DIG still relies exclusively on observed execution traces, which can lead to spurious invariants when some loop paths remain unexecuted. In the context of multipath loops, this limitation is particularly pronounced, as the interactions between different paths may require disjunctive reasoning or path-sensitive analysis capabilities that DIG does not support. Consequently, its inferred invariants may fail to capture all path-dependent behaviors, reducing soundness in complex loops.

Data-Driven: The data-driven approaches have been presented based on the analysis of the concrete execution data collected. This runtime information uses machine learning [13,14], techniques or statistical analysis [15,16,34,35] to discover patterns and relationships among them. The extracted patterns are then generalized to form potential loop invariants and then leverage empirical evidence from program executions, ensuring that the inferred invariants are satisfied with the collected data, though their accuracy depends heavily on the quality and coverage of the test cases used.

The data-driven approach, although effective for inferring loop invariants in simple cases, faces significant limitations when applied to multipath loops. Its accuracy heavily depends on the quality and coverage of collected concrete states; unobserved execution paths may result in invariants that miss important behaviors. Furthermore, the approach often struggles with scalability in large or complex programs, as gathering and analyzing extensive execution data can be computationally expensive. A key limitation is that traditional data-driven methods primarily focus on concrete values from observed executions without explicitly accounting for variations among multiple loop paths. This omission can lead to incorrect or incomplete invariants that fail to capture the full set of path-dependent behaviors and are incapable of representing necessary disjunctive invariants.

Recent work has investigated using large language models (LLMs) like GPT-4 to infer the loop invariant by prompting them with code and natural language descriptions [36–40]. While LLMs can generate sufficient invariants for simple cases, they have key limitations: they lack formal guarantees, often produce incorrect assertions, struggle with complex loops, and fail to cover all execution paths. Crucially, they cannot provide formal proofs, limiting their use in rigorous program verification where mathematical soundness is essential.

3 Background

In this section, we first provide an overview of loop invariants and then present the concepts of control flow graphs and loop paths.

3.1 Loop Invariant

Loop invariants express conditions that hold throughout the execution of a loop, regardless of the number of iterations. They serve as a foundation for reasoning about loop correctness and enable verification of loop behavior. For an invariant to be useful in proving program correctness. Formally, the loop invariant is a mathematical condition that remains true before and after each loop iteration. A valid loop invariant requires satisfying two conditions: (1) it must be weak enough to be established from the loop's precondition, and (2) it must be strong enough to imply the postcondition at termination. These conditions correspond to the classical Hoare logic [5] formulation for establishing the partial correctness of a loop.

Given a loop with condition C, a precondition P and a postcondition Q, A loop invariant inv must satisfy the following conditions:

- **Initialization:** $P \Rightarrow inv$ (the invariant holds before the loop starts).
- **Preservation:** $\{inv \wedge C\}\ ST\ \{inv\}$, (if the invariant and the loop guard hold, executing the loop body re-establishes the invariant.)
- **Termination:** When the loop terminates, inv and the negation of the loop condition imply the postcondition ($\neg$ C $\wedge\ inv \Rightarrow Q$).

3.2 Loop Modeling Preliminaries

A loop can be represented by a control flow graph (CFG), as described in Definition 1. Throughout this paper, we assume that all loops terminate

Definition 1: *A Control Flow Graph (CFG) for a loop L is defined as a tuple* CFG = $(N, E, entry, exit, pre, Paths)$ *where:*

- N *is a finite set of nodes representing loop assignment statements*
- $E \subseteq N \times N$ *is a set of edges representing control flow between nodes*
- $entry \subset N$ *is the set of loop entry nodes (loop header)*
- $exit \subset N$ *is the set of exit nodes (loop exit)*
- $pre \in N$ *is the loop pre-header node*
- $Paths$ *denotes the set of loop paths representing all possible execution sequences of the loop. There are three types of paths: (1) the initial path, which starts at the (pre) and ends at the loop (entry); (2) the exit path, which starts at* $entry$ *and ends at the loop (exit); and (3) iterative paths, where each path starts at* $entry$*, executes a sequence of loop statements, and returns to* $entry$.

For an iterative path, we have $head(p) = tail(p)$, and such a path is executed repeatedly. In contrast, a one-time path is executed only once, after which control proceeds to one of the other paths.

Example 1: *Fig. 2b illustrates a C loop with six nodes:* $(pre, entry, exit, n_1, n_2, n_3)$. *The loop contains four paths:* $(p_1 : pre \rightarrow entry)$, $(p_2 : entry \rightarrow n_1 \rightarrow n_2 \rightarrow entry)$, $(p_3 : entry \rightarrow n_1 \rightarrow n_3 \rightarrow entry)$, *and* $(p_5 : entry \rightarrow exit)$. *The* p_1 *is the initial path,* p_2 *and* p_3 *are iterative paths (since* $head = tail$*), and* p_5 *is the exit path.*

3.3 Semantics of Variable Updates and Dependencies

Assignment statements in loops fundamentally determine how program variables evolve across iterations. Different assignment forms create distinct dependency patterns among program statements, and accurately modeling these dependencies is essential for generating sound and expressive loop invariants. In this subsection, we classify the primary categories of variable update dependencies commonly encountered in loop structures. To systematically capture these dataflow relationships within our automated analysis framework, we formally define statement-level loop dependencies as follows:

- **Independent assignment**, where a variable receives a constant value that does not depend on other variables. For example, $x = 5$.
- **Self-dependent assignment**, where a variable is updated based on its previous value, creates a linear relationship between the current and previous value of the variable. For example, $x = x + 1$ or $x = 2 * x$.

- **Variable-dependent statement**, where a variable is assigned based on other variables. For example, $x = y + 1$ or $x = x + y$. This type of dependency can be further classified into two categories, depending on whether the dependency occurs within the same loop iteration or across successive iterations. *Intra-iteration dependencies* occur when a variable relies on another variable updated earlier in the same iteration, whereas *inter-iteration dependencies* arise when a variable depends on the value of another variable from a previous iteration. We formally define these two types of dependencies as follows:

Definition 2: *Let L be a loop with body containing set of statements $ST = \{st_1, st_2, \ldots, st_n\}$ executed sequentially within each iteration, and let V be the set of loop variables. For $v_i, v_j \in V$, we say that v_j has a* variable dependency *on v_i at iteration k if there exist statements $st_p, st_q \in ST$ with $p < q$ such that:*

- **Intra-iteration dependency (IntraD):** $st_p : v_i^{(k)} = f(v_i^{(k-1)}, \ldots)$ *and* $st_q : v_j^{(k)} = g(v_i^{(k)}, \ldots)$, *i.e., v_j depends on the value of v_i updated earlier in the same iteration.*
- **Inter-iteration dependency (InterD):** $st_p : v_i^{(k)} = f(v_i^{(k-1)}, \ldots)$ *and* $st_q : v_j^{(k)} = g(v_i^{(k-1)}, \ldots)$, *i.e., v_j depends on the value of v_i from the previous iteration.*

Example 2: C*onsider two variables x and y within a loop. An IntraD dependency arises when y uses the newly updated value of x within the same iteration, as in $x = x + 1; y = y + x$. In contrast, an InterD dependency occurs when y in the current iteration relies on the value of x from the previous iteration, as in $y = y + x; x = x + 1$, where the updated value of x is carried over and used by y in subsequent iterations.*

3.4 Patterns of Variable Updates

For a given variable and iterative loop path, repeated executions of the path induce a sequence of values associated with that variable. The n-th term of the sequence is defined as the value of the variable after n consecutive executions of the corresponding path. To establish a well-defined operational scope for trace generation and closed-form reasoning, AutoINF restricts its analysis to execution paths composed exclusively of numerical variables undergoing linear updates. More specifically, the framework focuses on *linear recurrence relations with constant coefficients* and *stratified linear dependencies*. These restrictions avoid the inherent complexity of arbitrary nonlinear loop summarization and enable the resulting recurrence relations to be deterministically unfolded and compositionally solved in closed form along individual paths. Within this setting, we classify the generated sequences according to the linear dependency structures induced by variable update statements as follows:

- **Self-Dependent Sequences:** Occur when a variable is updated based on its own previous value through a linear *self-dependent assignment*. The n-th term can be derived from standard linear recurrence relations. For an additive update of the form $x = x \pm c$, where c is a constant, the value after n iterations is $x^{(n)} = x^{(0)} \pm c \cdot n$. For a multiplicative update $x = x \cdot c$, the value becomes $x^{(n)} = x^{(0)} \cdot c^n$.
- **Dependent Sequences:** Arise when a variable update depends linearly on other variables within the loop body. The n-th term of such sequences varies according to the dependency type. For *InterD* dependencies (e.g., $x = x + 1;\ y = y + x$), the terms evaluate to $x^{(n)} = x^{(0)} + n$ and $y^{(n)} = y^{(0)} + \sum_{i=0}^{n-1} x^{(i)}$. For *IntraD* dependencies (e.g., $y = y + x;\ x = x + 1$), the terms evaluate to $x^{(n)} = x^{(0)} + n$ and $y^{(n)} = y^{(0)} + \sum_{i=0}^{n-1} x^{(i+1)}$.

4 Path-Sensitive Loop Modeling

In this section, we present a path-sensitive loop model that enables precise reasoning about the behavior of loops with multiple execution paths. Our model is fundamentally constructed based on the PDA framework [19], which we adapt to systematically decouple complex control-flow structures into isolated execution paths. The modeling process is constructed in two main steps. First, leveraging the structural

properties of the PDA, we represent the loop as a set of states and transitions where each state corresponds to a distinct execution path. Second, for each extracted state, we encode the corresponding variables' updates and path conditions as deterministic mathematical constraints. These constraints are subsequently used to generate loop traces and to formulate verification conditions, allowing candidate invariants to be validated by solving the corresponding state constraints with an SMT solver.

4.1 Path Dependency Automaton

In our framework, we construct the baseline structural paths of the PDA and then explicitly extract the semantic properties associated with each state. Capturing these localized semantics is essential to ensuring that the automaton accurately represents the complete behavioral profile of every execution path. By embedding this path-level semantic context directly into the automaton states, the model provides the precise behavioral information required by the downstream inference engine to synthesize sound and expressive loop invariants. Definition 3 presents the adapted definition of the PDA.

Definition 3: *Given a loop L with its $CFG = (N, E, entry, exit, pre, Paths)$, the path dependency automaton(PDA) is a tuple $M = (A, a_0, a_e, TR)$ where:*

- *A is a finite set of states, with each state $a \in A$, representing a path of the loop. States are classified into two categories: iterative and non-iterative. An iterative state corresponds to an iterative path that both starts and ends at the loop's $entry$ node, while a non-iterative state represents a one-time path.*
- *$a_0 \in A$ is the initial state that corresponds to the initial path of the loop.*
- *$a_e \in A$ is the exit state that corresponds to the exit path of the loop.*
- *$TR \subseteq A \times A$ is the set of transitions that capture the control flow relationships between states. For each transition $tr_i = (a_i, a_j) \in TR$, the connectivity constraint $head(a_j) = tail(a_i)$ must hold. Each transition is further annotated with a guard derived from the path condition of the target state.*
- *Each state a_i is annotated with a $4-tuple$ $(VAR, \alpha, \Phi, \Omega^{(n)})$, where:*
  1. *$VAR = D \cup U$ is the set of path's variables, where D represents the defined variables (i.e., assigned) and U represents variables used within the loop's path statements.*
  2. *Φ: is an FIFO queue of the path's variables' update statements.*
  3. *α: represents the path conditions that must be held to execute the state.*
  4. *$\Omega^{(n)}$: This is defined only for iterative states and represents the n–th term expressions of the state's variables.*

Example 3: *Fig. 2b illustrates the loop's CFG together with its PDA model. The model consists of four states (a_0, a_1, a_2, a_e), where a_0 and a_e denote the initial and exit states, respectively, and a_1 and a_2 represent iterative states that correspond to loop executions. The symbol $*$ marks the iterative states. Edges between nodes correspond to transitions, with the transition set given by $\{(a_0, a_1), (a_0, a_2), (a_1, a_2), (a_2, a_1), (a_1, a_e), (a_2, a_e)\}$.*

Table 1 presents the detailed components of the loop states shown in Fig. 2. Each row corresponds to a state a_i and specifies its condition (α), the set of defined and used variables (VAR), the variables' update statements Φ, and the $\Omega^{(n)}$ that represent the n–th term expressions for iterative states. For non-iterative states (a_0 and a_e), the n–th term column is marked with "✗". The "-" indicates that the corresponding state does not include this component; in particular, the initial state a_0 takes the preconditions as its condition, and the exit state a_e terminates without variable updates.

4.2 SSA-Based Symbolic Encoding of Variable Updates

To generate precise SMT constraints for multipath loops, AutoINF translates each statement $\phi(v_i) \in a_i.\Phi$ into a symbolic constraint representation. A standard technique for exposing explicit and acyclic

def-use relationships in program analysis is the Static Single Assignment (SSA) form [41]. Because each feasible execution path is isolated into a distinct state a_i, the control flow within the state is linear by construction. Consequently, AutoINF applies a simplified, state-isolated SSA transformation independently to each state. The transformation is guided by the ordered statement FIFO queue $a_i.\Phi$, where symbolic variable versions are assigned according to the dependency category of each statement, including self-dependent, interdependent, and intradependent updates. This ordering determines whether a variable reference corresponds to its value from the previous iteration or its updated value within the current iteration, thereby preserving precise def-use semantics while maintaining acyclic SMT constraints. This design is semantically equivalent to conventional SSA renaming for linear execution paths and eliminates the need for global merge operators (i.e., ϕ-nodes), since execution paths are separated prior to constraint generation. The encoding process is implemented as follows:

Table 1: Components of Loop states in Fig. 2 (✗ = the state is *non-iterative*, - = not exist).

State	Condition (α)	VAR	Variables Update Statements (Φ)	Variables' n-th Terms	The Mapped SMT Constraints
a_0	$n >= 0 \wedge u1 > 0,$	$D = \{a, b, i, u\},$ $U = \{\}$	$a = 0, b = 0,$ $i = 0, u = 0$	✗	$a = 0, b = 0,$ $i = 0, u = 0$
a_1	$i < n \wedge u < u1$	$D = \{a, b\},$ $U = \{a, b\}$	$a = a + 1, b = b + 2$	$a = a_0 + n;$ $b = b_0 + 2 * n$	$a == a_{k-1} + 1,$ $b = b_{k-1} + 2$
a_2	$i < n \wedge u >= u1$	$D = \{a, b\},$ $U = \{a, b\}$	$a = a + 2, b = b + 1$	$a = a_0 + 2 * n;$ $b = b_0 + n$	$a == a_{k-1} + 2,$ $b = b_{k-1} + 1$
a_e	$i == n$	–	–	✗	–

Definition 4: *Given a loop state $a_i \in A$, each statement $\phi(v_i) \in a_i.\Phi$ of the form $v = \phi(\mathcal{V})$, where $\mathcal{V} \subseteq a_i.U$, is translated into an SMT constraint. The translation follows these rules:*

- **Self-dependent statement**: *where a variable depends on its previous value, having the form of $v = \phi(v, c)$, the statement is encoded as: $v == \phi(v_{k-1}, c)$, where v_{k-1} is the value from the previous iteration $k-1$, and c is a constant.*
- **InterD dependent statement:** *This occurs when a variable is updated as $v = \phi(v, x)$, where $x \in a_i.D$ that is, x is either defined within the scope of state a_i but appears after v in the Φ queue order, or it is not defined within the same state. The statement is encoded as: $v == \phi(v_{k-1}, x_{k-1})$, where both v_{k-1} and x_{k-1} denote the values of v and x from the previous iteration $(k-1)$.*
- **IntraD dependent statement:** *If a variable $v \in a_i.D$ is assigned as $v = \phi(v, x)$ and $x \in a_i.D$ (i.e., x is defined within the scope of state a_i and appears before v in the Φ queue order). The statement is encoded as: $v == \phi(v_{k-1}, x)$, where x represents the updated value of x within the same iteration.*

Example 4: *In the variable update statements of the loop states shown in Table 1, each $\phi \in a_i.\Phi$ is translated into SMT constraints, as listed in the column Mapped SMT Constraints. For example, in state $a_1.\Phi$, we have $x = \phi(x) = x - y$ and $y = \phi(y) = y + 2$. The update for x represents an InterD dependency, and is therefore encoded as $x_k = x_{k-1} - y_{k-1}$, indicating that the value of x in the current iteration depends on the value of y from the previous iteration.*

5 Approach Construction

In this section, we describe the construction of AutoINF. As illustrated in Fig. 3, the proposed approach takes a loop as input and produces a corresponding loop invariant. AutoINF consists of three main stages: *Path-Sensitive Loop Modeling*, *Loop Invariant Inference*, and *Loop Invariant Validation*. In the *path-sensitive loop modeling* stage, the input loop is transformed into a model in which each execution path is represented as a distinct state. Variable update statements associated with each state are encoded as SMT constraints. This representation supports both trace generation across all execution paths for invariant synthesis and constraint-based verification of candidate invariants. The *loop invariant inference* stage first attempts to infer a unified invariant by generating execution traces from all paths and solving a polynomial invariant template. Successful template instantiation yields a candidate invariant for subsequent validation. When unified invariant inference is unsuccessful, the approach infers path-specific invariants, which are then combined into a disjunctive invariant. In the *loop invariant validation* stage, candidate invariants are checked for correctness against all execution paths using SMT solving. Unified invariants that fail validation trigger a counterexample-guided refinement process, in which extracted counterexamples are used to generate targeted traces and refine the invariant candidate. This refinement process is repeated for at most MAX iterations. When refinement does not yield a verified unified invariant, the approach reverts to disjunctive invariant inference. Disjunctive invariants are validated by checking that the combined formula holds over all execution paths. Upon successful verification, the invariant is returned; otherwise, the inference process terminates without a valid invariant.

5.1 Path-Sensitive Model Construction

The path-sensitive model is constructed in two main steps. First, we adopt the process presented in [19] to build a PDA, which transforms the loop into a state-transition model. Second, the update statements of the variables within each state are translated into path-isolated SSA constraints and encoded as SMT formulas. These constraints are then solved to extract a set of execution traces, providing mathematical representations that are used to verify the candidate invariant, ensuring the generated counterexamples accurately reflect the actual loop behavior.

Algorithm 1 outlines the procedure for constructing the PDA. It takes the CFG of the loop as input and produces the corresponding PDA as output. Each $path \in Paths$ is mapped to a corresponding state, with two special states a_0 and a_e that are designated as the initial and exit states, respectively. The initial state begins at the loop's pre node, while the exit state terminates at the loop's $exit$ node. The path conditions α and the set of assignment statements Φ are extracted directly from the source code. The Φ is defined as a queue to preserve the dependency order among statements by following the (First-In, First-Out (FIFO)) principle. Then, the transitions between all possible pairs of states are computed. A state a_i can transition to another state a_j only if $head(a_j)$ is equal to $tail(a_i)$.

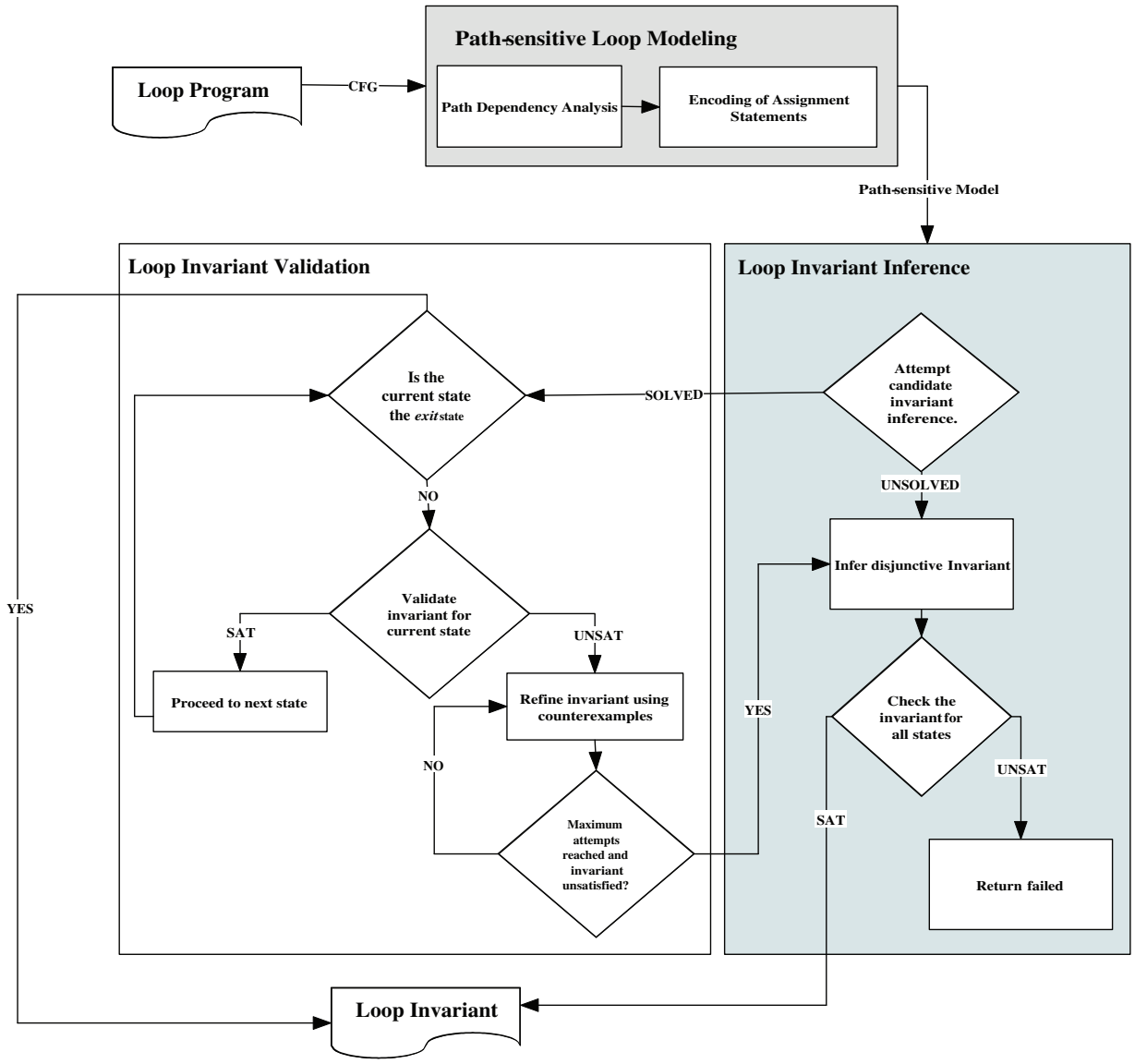


Figure 3: An overview of our approach autoINF.

Algorithm 1: PDA_Construction

Input: $CFG = (N, E, entry, exit, pre, Paths)$.
Output: constructed PDA M

$paths \leftarrow \varnothing$;
$A \leftarrow \varnothing$;
$TR \leftarrow \varnothing$
for *each* $path \in Paths$ **do**
  Map $path$ to state a_i;
  Extract $path$ condition α;
  Extract $path$ variables' assignment statement and enqueue it to Φ;
  Compute $n-th$ terms of variables if the $state$ is $iterative$ and add to $\Omega^{(n)}$;
  Add a_i with $(\alpha, \Phi, \Omega^{(n)})$ to states A; Mark a_i as initial state a_0 if $head(path) = L.pre$; Mark a_i as exit state a_e if $tail(path) = L.exit$;
for *each* $(a_i, a_j) \in \{(a_n, a_m) | a_n \in A \wedge a_m \in A \wedge tail(a_n) = head(a_m) \wedge n \neq m\}$ **do**
  Add $TR = TR \cup \{(a_i, a_j)\}$;
return $M = (A, a_0, a_e, TR)$;

Algorithm 2 executes the SSA-based symbolic encoding of the variable update statements within each state into SMT constraints. The algorithm receives the constructed PDA as input and outputs an updated PDA that formally represents the path-sensitive behavior of the loop. The process iterates through each state $a_i \in A$, and two variable sets are defined and initialized: the defined variables (D) and the used variables (U). Each variable statement ϕ in the queue Φ is then translated into an SSA-form SMT constraint. The dequeued statement ϕ is first parsed into an Abstract Syntax Tree (AST) using depth-first search (DFS), and subsequently traversed for encoding. The algorithm processes AST tree nodes recursively: the assignment node is transformed into equality SMT constraints with the node's target and value, and binary operations are mapped to their SMT equivalents. The variables are evaluated based on their context: variables under a *Store* context represent the newly defined SSA instance, which is added to the D set and maintains its original name. Variables under a *Load* context are converted into localized SSA version variables with the suffix $\_k-1$ if they represent previous values (not defined before this statement within the same state or matching the currently defined variable) expressed by the condition $node.id \notin D \vee node.id = currentVAR$. Then, the translated statement is added to $new_S T$. Before proceeding to the next state, the state's information is updated by assigning $a_i.\Phi = new_S T$ and recording the corresponding variable sets D and U.

Algorithm 2: `StatementToSMT`

Input: PDA $M = (A, a_0, a_e, TR)$.
Output: path-sensitive Model (updated PDA) $M = (A, a_0, a_e, TR)$

for *each* $a_i \in A$ **do**
  Define $a_i.D = \{\}$ and $a_i.U = \{\}$;
  Extract $variables$ in state condition $a_i.\alpha$ and add them to $a_i.U$;
  while $a_i.\Phi$ *not empty* **do**
    Dequeue ϕ from $a_i.\Phi$;
    $tree$ = AST Parse ϕ ;
    Traverse $tree.node$;
    $node$ type is Name: **if** $node.ctx$ *is Store:* **then**
      Create SMT integer $variable$ with the original name;
      Add $variable$ to D;
    if $node.ctx$ *is Load:* **then**
      $node.id \notin D \vee node.id = currentVAR$ create a SMT $new\_variable$ with "$\_i$" suffix (e.g., $x \to x_i$); create a SMT $new\_variable$ with same name of variable;
      Add $new\_variable$ to U;
    if $node.type$ *is BinOP:* **then**
      $left$ = Traverse$(node.left)$ and $right$ = Traverse$(node.right)$;
      Map operators to SMT operations with $left$ and $write$;
    if $node.type$ *is Assignment:* **then**
      $target = node.target[0]$;
      $currentVAR = target$;
      $value$=Traverse$(node.value)$;
      $new\_ST = new\_ST \cup \{target == value\}$;
    if $node.type$ *is constant:* **then**
      numeric value directly as SMT constant;
    $a_i.\Phi = new\_ST$;
    add D and U to a_i;
return path-sensitive Model (updated PDA) M

5.2 Loop Invariant Inference

AutoINF adopts a counterexample-guided invariant Refinement (CEGIR) strategy that dynamically selects between two distinct inference phases depending on the behavioral alignment of the loop paths. To provide a rigorous, mathematical basis for this selection process, we formally define the concept of path compatibility as follows:

Definition 5: *Let* $P = \{p_1, p_2, \ldots, p_k\}$ *be a subset of isolated execution paths extracted from the PDA of a loop* L*, and let* $\Phi(p_m)$ *denote the SMT-encoded symbolic transition relation for path* p_m*. The paths in* P *are defined as* compatible *if and only if there exists a single single candidate invariant* I *such that:* $\forall p_m \in P : (I \wedge \Phi(p_m) \implies I')$ *where* I' *represents the invariant evaluated on the updated post-state variables. Conversely, if no single formula can soundly satisfy the consecution condition across all paths in* P *simultaneously due to conflicting numerical data-flow behaviors, the paths are classified as* incompatible.

Following Definition 5, our CEGIR framework splits its search space into two operational phases:

1. **Unified Invariant Inference:** Executed when a cluster of execution paths is determined to be compatible. Their behaviors can be characterized by a single inductive invariant, allowing the engine to synthesize a compact, shared single property that handles multiple paths simultaneously.
2. **Disjunctive Invariant Inference:** Triggered when paths are found to be incompatible. Because their divergent assignment patterns cannot be soundly over-approximated by a single formula, AutoINF infers distinct invariant clauses tailored to each individual execution trace and combines them into a robust disjunction ($\bigvee I_m$).

The approach employs different verification strategies depending on the form of the inferred invariant. AutoINF extracts candidate nonlinear equality invariants by constructing a polynomial template that captures multiplicative combinations of loop variables up to a specified polynomial degree. The general form of the invariant template is defined as follows:

$$c_1 x_1 \cdot c_2 x_2 + c_3 x_3 + \cdots + c_n x_n = 0 \tag{1}$$

The template contains n unknown coefficients c_i and n unknown terms x_i, which are the polynomial terms constructed from the program variables.

5.2.1 Unified Invariant Inference

AutoINF first attempts to infer a unified invariant, that characterizes all paths. Execution traces are generated from all paths in the path-sensitive model and used to solve a polynomial template. The resulting candidate $invariant$ is then verified across all loop paths.

Derivation of Loop Traces: AutoINF generates execution traces from a path-sensitive model rather than relying solely on concrete program execution. Although dynamic execution produces traces, it cannot establish invariant soundness over all feasible paths. By combining model-based trace generation with path-sensitive invariant validation, AutoINF enables sound verification beyond observed executions. In addition, traces can be generated selectively for individual execution paths in response to counterexamples, supporting targeted refinement and the inference of path-specific invariants that can be combined into disjunctive forms when needed.

Algorithm 3 illustrates how the path-sensitive model is used to generate a diverse set of loop execution traces. The algorithm takes as input the path-sensitive loop model M, the required number of traces K_t, the set of variables whose values need to be collected ($varSET$) and the set of blocked constraints ($blocks$), which represent the initial values of variables that have previously been used. The procedure traverses the states sequentially, starting from the initial state a_0, where the loop variables are initialized. In each state, the satisfiability of the SMT-encoded constraints is verified using the solver, and the corresponding variable valuations are extracted from the solver's model. The execution of a state is performed only if its condition $currentST.\alpha$ is satisfied; otherwise, the process transitions to the reachable state according to the corresponding transition condition ($currentST = \{a_m | (currenST, a_m) \in TR \wedge \text{checkSMT}(a_m.\alpha, IN) = SAT\}$). The set IN contains all variables whose values are obtained from the solver's model. From this set, the values of the state's used variables are extracted and stored in $tempIN$. Also from the IN variables' set, a $trace$ is constructed by collecting the values of all variables in $varSET$, representing a complete execution instance of the loop state. If the process reaches the exit state a_e and the total number of collected traces is not equal to the required count k_t, the procedure invokes the algorithm to request the remaining K–i traces.

Algorithm 3: `generate_traces`

Input: Loop's path-sensitive model M, Number of required traces K_t, Variables of Interest $varSET$, blocked constraints $blocks$.

Output: set of loop traces $traces$, set of updated blocked constraints $blocks$.

$values$ = optimize($a_0.\alpha$, $\neg blocks$);
$blocks = blocks \cup \{\bigwedge_{v \in values} v = values[v]\}$;
checkSMT($a_0.\Phi$, $values$);
IN=getModel( ) $traces = \{(v = IN[v]) | \forall v \in varSET\}$;
$currentST = a_0$;
while $currentST \neq a_e \wedge i < K_t$ **do**
    $currentST = \{a_m | (currenST, a_m) \in TR \wedge$ checkSMT($a_m.\alpha, IN$)= $SAT\}$;
    while *checkSMT(currentST.α, IN)=SAT* **do**
        $tempIN = \{(v = IN[v]) | \forall v \in IN$ if $v \in currentST.U$ add "$\_k - 1$" suffix for variable name$\}$;
        checkSMT($currentST.\Phi$, $tempIN$);
        IN = getModel( );
        $traces = traces \cup \{(v = IN[v]) | \forall v \in varSET\}$;
    $i = i + 1$;
if $|traces| < K_t$ **then**
    $newTraces, newBlocks$=`generate_traces`(A, $(K - i)$, $varSET$, $blocks$);
    $blocks = blocks \cup newBlocks$;
    $traces = traces \cup newTraces$;
return $traces$, $blocks$

Counterexample-Guided Refinement: Algorithm 4 presents the core AutoINF strategy, including the conditions under which disjunctive invariants are required. Execution traces extracted from the path-sensitive model, as described in Algorithm 3, are used to instantiate the invariant template by substituting concrete trace values for program variables. This produces a system of linear equations over the unknown template coefficients, which is solved using standard linear equation–solving techniques. The resulting coefficient values, when available, are substituted back into the template to derive the corresponding equality invariant. If template solving fails, the algorithm proceeds to infer a disjunctive invariant. Similarly, if verification of a derived invariant fails, disjunctive invariant inference is invoked.

In Algorithm 5, AutoINF verifies the inferred invariant across all loop paths to eliminate spurious results. Verification begins by checking the candidate invariant against the initial loop state a_0, and then iteratively traverses all states in the path-sensitive model until reaching the exit state a_e. For each state, the invariant is checked against the conjunction of the state's condition $a_i.\alpha$ and its variable assignments $a_i.\Phi$. When the invariant fails to hold at a given state, a set of counterexamples is generated and used to refine the invariant. This refinement process iterates until either a valid invariant is obtained or the maximum number of attempts (MAX) is reached. If refinement fails after MAX attempts, the verification is considered unsuccessful, and the algorithm returns failure, which yields an inference disjunctive invariant. A key aspect of this approach is the generation of path-specific counterexamples, which indicate exactly which execution path violates the candidate invariant. By associating counterexamples with particular paths, AutoINF ensures that the refinement process is targeted and precise, avoiding over-generalization or spurious corrections that could arise from path-agnostic counterexamples.

Algorithm 4: `invariant_inference`

Input: Loop's path-sensitive model M, Variables of Interest $varSET$, Maximum Degree $maxDegree$
Output: $Invariant$

```
traces ← ∅;
IN ← ∅;
blocks ← ∅;
template, terms=generateTemplate(varSET, maxDegree);
traces, blocks = generate_traces(A, |terms|, varSET, ∅);
Invariant, result = solveTemplate(template,traces);
if result = FALSE then
    invariant=infer_disjunctive(varSET, M) ;
else
    proved = check( Invariant, M, blocks, traces);
    if proved = FALSE then
        invariant=infer_disjunctive(varSET, M)
return Invariant
```

5.2.2 Disjunctive Invariant Inference

If the invariant template cannot be solved using traces from all loop paths, or if a synthesized unified invariant is violated by some paths, the analysis proceeds with a disjunctive representation. The loop invariant is then expressed in the following form:

$$\mathcal{I} = \bigvee_{i=1}^{k} (Inv_i) \tag{2}$$

where Inv_i is the invariant specific to the path s. Unlike a unified invariant that must hold across all paths simultaneously, a disjunctive invariant states that at each point in the loop, at least one disjunct Inv_s is satisfied. Algorithm 6 describes the procedure for disjunctive invariant inference, which consists of two main steps: derivation of path-specific traces and counterexample-guided refinement. The algorithm iterates over each state in the path-sensitive model, extracts path-specific traces to instantiate the invariant template, and applies CEGIR refinement to validate and refine the resulting candidates.

Derivation of Path-Specific Traces: Unlike unified invariant inference, which relies on traces collected from all execution paths, disjunctive inference requires path-specific traces generated independently for each path. For each state a_i in the path-sensitive model, traces are obtained by solving the state's constraints $(a_i.\alpha \wedge a_i.\Phi)$ using an SMT solver, yielding valuations that satisfy only the conditions and updates of path a_i. This isolation is essential, since combining traces from incompatible paths would produce an inconsistent linear system and prevent template solving. By generating traces independently, AutoINF ensures that each path-specific invariant Inv_s captures the polynomial relationships unique to path a_i.

Algorithm 5: `check`

Input: invariant inv, path-sensitive model M, set of blocked constraints $blocks$.
Output: checking result $proved$.

$currentST = a_0$;
$values$=optimize($a_0.\alpha$,$\neg blocks$);
checkSMT($a_0.\Phi, values$);
IN=getModel();
$proved$=FALSE;
while $(currentST \neq a_e)$ **do**
  $constraint = \bigwedge_{\phi \in currentST.\Phi} \phi_i \wedge currentST.\alpha$;
  if $checkSMT(\neg inv, constraint, IN)$=$SAT$ **then**
    while $numTRY < MAX \wedge checkSMT(currentST.\alpha, CExamples)$=$SAT$ **do**
      $proved$ =FALSE;
      CX=getModel( );
      $traces = traces \cup \{(v = CX[v]) | \forall v \in varSET\}$;
      $Invariant$= solveTemplate($template, traces$);
      if $checkSMT(\neg inv, constraint, IN)$=$UNSAT$ **then**
        $proved$=TRUE;
        break;
    if $proved$=$FALSE$ **then**
      break;
  IN=Calculate the value of variables after n time using SMT with $currentST.\Omega^{(n)}$;
  $currentST$ ={$a_m | (currenST, a_m) \in TR$} $\wedge$ checkSMT($a_m.\alpha, IN$) == SAT;
  $proved$=FALSE;
return $proved$;

In Algorithm 6 (lines 9–16), the solver evaluates each state's transition and assignment constraints to extract variable valuations, which are accumulated into trace sets $traceST$ for invariant inference. If a state condition becomes unsatisfied before sufficient traces are collected, the procedure advances to successor states and, upon reaching the exit state a_e, restarts from the initial state a_0 with new inputs. To avoid unnecessary recomputation when traversing paths other than the one currently being analyzed, the nth iteration of the state variables is used to summarize the effect of executing those paths. This summarized form allows the algorithm to compute the resulting variable values without explicitly re-executing intermediate states, thereby reducing redundant exploration and improving efficiency. Once enough traces are obtained, they are substituted into the invariant template to construct a solvable system of linear equations, yielding the candidate path-specific invariant Inv_s. As shown in Table 2, our approach effectively extracts path-specific traces independently. The loop under analysis involves two variables of interest (x, y). By leveraging path-specific traces, AutoINF infers a disjunctive invariant of the form $(y = x^2 - x - 15 \vee 2y = x^2 - x - 30)$, where the invariant for state a_1 is $2y = x^2 - x - 30$, and for state a_2 is $y = x^2 - x - 15$.

Table 2: State-specific traces of example in Fig. 1.

State	x Values	y Values
a_0	$(x = -5)$	$(y == 0)$
a_1	$(x == -4), (x == -3)$ $(x == -2), (x == -1)$ $(x == 0)$	$(y == -5), (x == -9)$ $(y == -12), (y == -14)$ $(y == -15)$
a_2	$(x == 1)(x == 2)$ $(x == 3), (x == 4)$ $(x == 5)$	$(y == -15)(y == -13)$ $(y == -9), (y == -3)$ $(y == 5)$

Verification of Disjunctive Invariants A central issue in disjunctive invariant inference is that per-path validity does not guarantee the correctness of their disjunctive composition. A path-specific invariant Inv_s may hold for its associated path a_i, yet the disjunction of such invariants may fail to characterize the semantics of the entire loop if the path conditions do not form a sound partition of the reachable states. This motivates a two-level verification strategy. Per-path verification establishes the soundness of each Inv_s with respect to its corresponding path a_i via counterexample-guided refinement. Disjunctive verification then checks the combined invariant $\bigvee_i(Inv_s)$ over all states of the path-sensitive model, ensuring that the path guards correctly partition the reachable state space and that the disjunction covers all feasible executions, including transitions between paths as well as loop entry and exit.

(1) Counterexample-Guided Refinement: Algorithm 6 (lines 19–26) presents the CEGIR procedure. Once a candidate path-specific invariant Inv_s is derived from path-specific traces, it is validated exclusively against its corresponding path a_i. This is accomplished by checking the satisfiability of $\neg Inv_s \wedge a_i.\Phi \wedge a_i.\alpha$ using an SMT solver. If the formula is unsatisfiable, Inv_s is confirmed to be valid for path a_i and is accepted. Otherwise, the solver produces a counterexample consisting of variable valuations that satisfy the constraints of path a_i while violating Inv_s. These counterexamples are then used to generate additional path-specific traces for a_i, and the invariant template is re-instantiated and solved using the expanded trace set. This counterexample-guided refinement process continues for at most MAX iterations.

(2) Disjunctive Invariant Checking: In Algorithm 6 (lines 29–31), the combined disjunctive invariant is validated against all loop paths by iterating over each state a_i and checking the invariant under the state's path condition $a_i.\alpha$ and transition constraints $a_i.\Phi$. If the invariant does not hold for any state, the procedure returns FAIL.

Algorithm 6: `infer_disjunctive`

Input: invariant template $template$, path-sensitive model $M = \langle A, TR, a_0, a_e \rangle$, set of blocked constraints $blocks$.

Output: Disjunctive inductive invariant $\mathcal{I}$ or FAIL

$InvSet \leftarrow \varnothing$;
$currentST \leftarrow a_0$;
$init \leftarrow \text{optimize}(a_0.\alpha, \neg blocks)$;
checkSMT$(a_0.\Phi, init)$;
$IN \leftarrow$ getModel();
while $(currentST \neq a_e)$ **do**
  $traceST \leftarrow \varnothing$;
  $template, terms \leftarrow$ generateTemplate$(varSET, maxDegree)$;
  $traceST \leftarrow \{(v = IN[v]) \mid \forall v \in varSET\}$;
  while $(|traceST| < |terms|)$ **do**
    while $(checkSMT(currentST.\alpha, IN) == SAT)$ **do**
      $tempIN \leftarrow \{(v = IN[v]) \mid$ if $v \in a_i.U$ add "_i" suffix for variable name$\}$;
      checkSMT$(currentST.\Phi, tempIN)$;
      $IN \leftarrow$ getModel();
      $traceST \leftarrow traceST \cup \{(v = IN[v]) \mid \forall v \in varSET\}$;
    $nextST \leftarrow \{a_m \mid (currentST, a_m) \in TR \wedge$ checkSMT$(a_m.\alpha, IN) ==$ SAT$\}$;
    $IN \leftarrow$ Calculate values using SMT with $nextST.\Omega^{(n)}$;
  $inv_s \leftarrow$ solveTemplate$(template, traceST)$;
  $constraint \leftarrow \bigwedge_{\phi \in currentST.\Phi} \phi_i \wedge currentST.\alpha$;
  if $checkSMT(\neg inv_s, constraint, IN) == SAT$ **then**
    $CX \leftarrow$ getModel() ;
    while $numTRY < MAX \wedge checkSMT(currentST.\alpha, CX) == SAT$ **do**
      $traceST \leftarrow traceST \cup \{(v = CX[v]) \mid \forall v \in varSET\}$;
      $inv_s \leftarrow$ solveTemplate$(template, traceST)$;
      if $checkSMT(\neg inv_s, constraint, IN) == UNSAT$ **then**
        break;
      $numTRY \leftarrow numTRY + 1$;
  $InvSet \leftarrow InvSet \cup \{inv_s\}$;
  $currentST \leftarrow \{a_m \mid (currentST, a_m) \in TR \wedge$ checkSMT$(a_m.\alpha, IN) ==$ SAT$\}$;
$\mathcal{I} \leftarrow \bigvee_{inv_k \in InvSet} inv_k$;
for *each state* $a_i \in A$ **do**
  if $checkSMT(\mathcal{I} \wedge a_i.\alpha \wedge a_i.\Phi \wedge \neg \mathcal{I}') == SAT$ **then**
    return *FAIL*;
return $\mathcal{I}$

5.3 Formalization of Two-Level Verification

From the Definition 3 of our PDA model The set of path conditions forms a precise semantic partition of the loop domain, meaning the global monolithic loop body transition T is equivalent to the disjunction

of all individual path actions:

$$T(\mathbf{v},\mathbf{v}') \equiv \bigvee_{a_i \in A} (a_i.\alpha(\mathbf{v}) \wedge a_i.\Phi(\mathbf{v},\mathbf{v}')) \tag{3}$$

Our framework synthesizes a candidate local numerical predicate inv_i for each path state $a_i \in A$ as described in Eq. (2). Instead of passing a monolithic, highly disjunctive formula directly to an SMT solver, AutoINF decouples verification into a two-level strategy defined mathematically below:

Definition 6 (Level 1: Per-Path Local Refinement): *For each isolated path state $a_i \in A$, the local candidate predicate inv_i must be sound with respect to its specific execution trace segment. This level checks the validity of the implication:*

$$\forall a_i \in A: \quad \vDash (a_i.\alpha(\mathbf{v}) \wedge a_i.\Phi(\mathbf{v},\mathbf{v}')) \implies inv_i(\mathbf{v}') \tag{4}$$

Definition 7 (Level 2: Global Cross-Path Inductive Check): *To guarantee soundness across arbitrary path switching between iterations, the combined disjunction $\mathcal{I}(\mathbf{v})$ must behave inductively over every feasible pathway. This level requires checking that for every path state $a_i \in A$, starting within the global invariant boundary preserves the global invariant after taking that path:*

$$\forall a_i \in A: \quad \vDash (\mathcal{I}(\mathbf{v}) \wedge a_i.\alpha(\mathbf{v}) \wedge a_i.\Phi(\mathbf{v},\mathbf{v}')) \implies \mathcal{I}(\mathbf{v}') \tag{5}$$

The corresponding SMT safety query evaluates the cross-path violation space:

Theorem 1: *(Soundness of Two-Level Inductive Maintenance): If a disjunctive loop invariant candidate $\mathcal{I}(\mathbf{v}) = \bigvee_{k \in A} inv_k(\mathbf{v})$ inferred across all iterative paths satisfies both Level 1 (Eq. (4)) and Level 2 (Eq. (5)) verification conditions, then $\mathcal{I}(\mathbf{v})$ mathematically satisfies the classical monolithic inductive loop maintenance condition:*

$$\mathcal{I}(\mathbf{v}) \wedge T(\mathbf{v},\mathbf{v}') \implies \mathcal{I}(\mathbf{v}') \tag{6}$$

Proof: Assume that the Level 2 verification condition holds successfully. Thus, by Definition 7, the SMT solver has proven that for every individual path state $a_i \in A$, the following implication is valid:

$$\vDash (\mathcal{I}(\mathbf{v}) \wedge a_i.\alpha(\mathbf{v}) \wedge a_i.\Phi(\mathbf{v},\mathbf{v}')) \implies \mathcal{I}(\mathbf{v}') \tag{7}$$

Because this property holds independently for every distinct path state in the finite set A, we can safely take the disjunction of the antecedents across all paths $a_i \in A$. By the distributive law of propositional logic, taking a disjunction of implications sharing an identical consequent yields:

$$\left(\bigvee_{a_i \in A} (\mathcal{I}(\mathbf{v}) \wedge a_i.\alpha(\mathbf{v}) \wedge a_i.\Phi(\mathbf{v},\mathbf{v}')) \right) \implies \mathcal{I}(\mathbf{v}') \tag{8}$$

We apply the distributive law of conjunction over disjunction to pull the invariant term $\mathcal{I}(\mathbf{v})$ outside of the path-specific disjunction:

$$\left(\mathcal{I}(\mathbf{v}) \wedge \left(\bigvee_{a_i \in A} (a_i.\alpha(\mathbf{v}) \wedge a_i.\Phi(\mathbf{v},\mathbf{v}')) \right) \right) \implies \mathcal{I}(\mathbf{v}') \tag{9}$$

From our structural partition model established in Eq. (3), we know that the inner disjunction over all path conditions and transitions is identical to the global loop body transition relation $T(\mathbf{v}, \mathbf{v}'0)$:

$$T(\mathbf{v}, \mathbf{v}') \equiv \bigvee_{a_i \in A} (a_i.\alpha(\mathbf{v}) \wedge a_i.\Phi(\mathbf{v}, \mathbf{v}')) \tag{10}$$

Substituting this monolithic equivalence relation directly back into Eq. (9) yields:

$$\mathcal{I}(\mathbf{v}) \wedge T(\mathbf{v}, \mathbf{v}') \implies \mathcal{I}(\mathbf{v}') \tag{11}$$

This matches the exact definition of classical inductive loop maintenance.

Remark on Initialization and Termination: Because AutoINF is architected to perform localized path-sensitive synthesis specifically over the loop's iterative paths, the initialization condition ($P(\mathbf{v}) \implies \mathcal{I}(\mathbf{v})$) and the final post-condition verification ($\mathcal{I}(\mathbf{v}) \wedge \neg Loop\_Cond(\mathbf{v}) \implies R(\mathbf{v})$) are evaluated through standard verification conditions using the global disjunction. Theorem 1 establishes that once these base-case and terminal queries are discharged, our two-level strategy guarantees absolute soundness for the inductive step. □

6 Experimental Setting

6.1 Benchmark Description

To evaluate AutoINF, we use a publicly available benchmark suite previously adopted in [17], which contains loop programs along with documented target invariants. The benchmark provides implementations in both Java and C; in our evaluation, we focus exclusively on the C programs, as the current AutoINF tool supports invariant inference for C code. From this benchmark, we select 60 programs that cover a wide range of loop structures and invariant patterns. To further evaluate AutoINF's capability in handling disjunctive and nonlinear invariants, we extended our experimental study through data augmentation techniques applied to a base multipath program. Using this process, we systematically constructed 12 benchmark programs exhibiting increasingly complex multipath behaviors that require expressive disjunctive and nonlinear invariants. Specifically, the benchmarks were generated by modifying variable identifiers and scaling variable update statements to introduce polynomial relations of degrees up to three. The resulting benchmark suite was organized into multiple categories, enabling us to assess AutoINF not only on complex multipath scenarios, but also on single-path loops with diverse constraint distributions.

- **Single-Path Loop Benchmarks (SPL):** This category consists of 26 programs whose loops contain no internal branching; they follow a single execution path. The corresponding invariants range from simple linear relations to more complex nonlinear ones.
- **MultiPath Loop Benchmarks (MPL):** This category contains 46 programs with unnested loops that include conditional branches within the loop body, resulting in multiple execution paths. The invariants range from linear to nonlinear relations, with several cases further requiring disjunctive invariant forms.

6.2 Parameter Settings

We implemented AutoINF as a tool that combines static program analysis with symbolic constraint solving. The front end is built on Low Level Virtual Machine (LLVM), which is used solely to analyze program control flow and construct the PDA by extracting loop paths, state transitions, and variable update information. Once the PDA is generated, it is passed to Python, which encodes each state's update semantics as mathematical constraints and performs both invariant inference and verification. Using the PDA as a

formal loop model, AutoINF generates loop traces, infers candidate invariants, and validates them through constraint solving and counterexample-guided refinement.

The experiments were conducted on a system equipped with an Intel® Core™ i7 processor, an NVIDIA GeForce RTX 3050 GPU. The implementation was developed using Python version 3.11.8 and LLVM version 16.0.6. These hardware and software configurations were selected to ensure efficient execution and reliable results throughout the experimental evaluation.

6.2.1 Maximum Refinement Attempts (MAX)

The MAX parameter controls the counterexample-guided refinement process, which is central to AutoINF's invariant inference. When an invariant candidate fails verification, the SMT solver returns a counterexamples which are the concrete variables valuation that violates the invariant. AutoINF uses this counterexample to generate additional execution traces for the problematic path and re-solves the template with the augmented trace set. This refinement cycle can repeat multiple times, each iteration potentially producing a better candidate. The MAX parameter serves two key roles: (i) it guarantees termination by preventing unbounded refinement in cases where path behaviors are fundamentally incompatible, and (ii) it controls computational overhead by limiting the cost of repeated synthesis and verification.

We determined this value through empirical analysis of refinement convergence behavior across our 72 benchmarks. Fig. 4 shows the distribution of refinement iterations required for successful invariant inference. The distribution is heavily left-skewed, indicating that most invariants converge quickly: 42 benchmarks (58.3%) require only 1 refinement iteration, meaning the initial candidate inferred from traces is already valid. An additional 27 benchmarks (37.5%) converge within 2–5 iterations, bringing the cumulative success rate to 95.8% by iteration 5. Only 3 benchmarks (4.2%) require 6–8 iterations, and critically, no benchmark required more than 8 iterations. This suggests that if refinement has not succeeded by iteration 10, it indicates genuine path incompatibility rather than insufficient refinement attempts. We set $MAX = 10$ to provide a 2× safety margin beyond the 95th percentile (5 iterations), ensuring all convergent cases are captured while limiting wasted computation on incompatible paths. To validate this choice, we tested failed cases with $MAX \in \{20, 50, 100\}$ and found identical results—benchmarks failing at $MAX = 10$ also failed at higher limits, confirming they represent genuine incompatibility rather than premature termination.

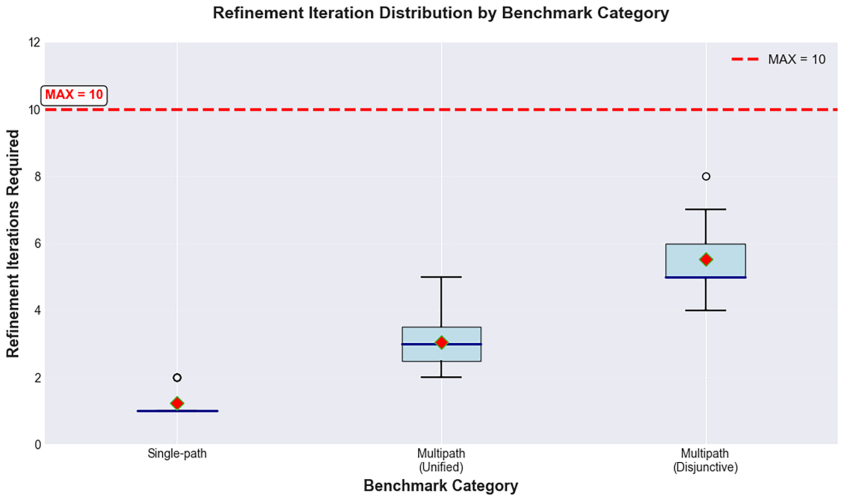


Figure 4: Distribution of refinement iterations required for successful invariant inference across 72 benchmarks.

6.3 Evaluation Metrics

To measure the effectiveness of AutoINF, we define success based on the following correctness criteria:

- **Path-completeness:** The inferred invariant must hold for all feasible execution paths of the loop, ensuring that multipath behaviors are fully captured.
- **Property consistency:** A strong invariant must not only hold throughout the loop execution but also be consistent with the loop's preconditions and postconditions. We verify that inferred invariants do not violate the loop preconditions and correctly imply postconditions, ensuring that the invariants are meaningful in the context of the surrounding program. A valid invariant must satisfy three correctness conditions:
  - **Initialization:** $P \Rightarrow inv$ (the invariant holds before the loop starts).
  - **Maintenance:** The invariant inv must hold at the beginning of each iteration ($inv \wedge C$).
  - **Termination:** When the loop terminates, inv and the negation of the loop condition imply the postcondition ($\neg$ C $\wedge$ $inv \Rightarrow Q$).
- **Disjunctive invariant verification:** For loops requiring disjunctive invariants, each disjunct must be valid for its corresponding execution path, and the combined disjunctive invariant must hold across all paths. This ensures accurate modeling of incompatible behaviors across different execution paths. *We provide a formal mathematical proof of the soundness of this two-level inductive maintenance strategy in Section 5.3.*
- **Computation time:** The total time required to generate candidate templates, instantiate them over traces, and verify the resulting invariants using the SMT solver.

The `success rate (✓)` is then defined as the proportion of loops for which AutoINF infers invariants satisfying all of the above criteria. This metric captures both the correctness and completeness of the inferred invariants in multipath settings.

7 Results Discussion

This section presents the experimental results in the context of broader research challenges in automated loop invariant inference:

- **RQ1**: How can AutoINF accurately infer both linear and nonlinear invariants for single-path loops?
- **RQ2**: How can AutoINF generate a unified invariant that is valid across all execution paths of a multipath loop?
- **RQ3**: How effectively can AutoINF generate disjunctive invariants that accurately capture incompatible execution paths, including nonlinear behaviors?
- **RQ4**: How does AutoINF compare to other invariant inference tools

7.1 Answer to RQ1: How Can AutoINF Accurately Infer Both Linear and Nonlinear Invariants for Single-Path Loops?

This experiment evaluates AutoINF on 26 programs containing single-path loops with both simple linear and complex nonlinear variable relationships. These loops are relatively straightforward for invariant inference, as the absence of branching reduces the complexity of the analysis. Table 3 summarizes the results: the `VAR` column indicates the number of variables involved in each loop, while the `T` column reports the total execution time for the invariant inference phase implemented in Python. As expected, inference time increases with the number of variables, reflecting the higher computational cost of analyzing larger state spaces. The `D` column denotes the polynomial degree of the inferred invariant, and the `Correct` column uses ✓ to indicate that the invariant satisfies the correctness criteria and ✗ otherwise.

As shown in Table 3, AutoINF successfully infers all correct linear invariants. For nonlinear invariants, AutoINF succeeds in 13 out of 14 programs, producing strong invariant. The only exception is the `cohenCu` program, where AutoINF was unable to derive a complete equality invariant covering all variables of interest (x, y, z, n). This program failed to converge within the maximum refinement limit due to a fundamental limitation in our current prototype when dealing with conjunctive invariant. Because AutoINF is architected to isolate divergent control flows into distinct PDA path states, its refinement loop evaluates paths independently to resolve disjunctive bottlenecks. However, `cohenCu` requires a complex conjunction of higher-order non-linear cubic properties that must simultaneously bind all tracking variables universally across every path. Without a dedicated global conjunctive template-merging pass to recombine these independent path constraints, our pattern-matching heuristics cannot converge on the global relational bounds required for this specific arithmetic structure. These results establish a strong baseline for AutoINF's accuracy and efficiency on relatively simple loop structures.

Table 3: Experimental results for linear SPL benchmarks. (✓: produce results that match the documented invariants. ✗: failed).

	Prog.	VAR	T	D	Correct
Linear	H01	2	1.5	–	✓
	H02	2	1.6	–	✓
	H03	3	2.0	–	✓
	H05	2	1.4	–	✓
	H07	3	2.2	–	✓
	H19	2	1.5	–	✓
	H22	3	2.1	–	✓
	H30	2	1.4	–	✓
	H35	2	1.5	–	✓
	H37	2	1.6	–	✓
	H41	4	2.6	–	✓
	H44	3	2.3	–	✓
NonLinear	cav09_fig5b	5	4.1	2	✓
	cohenCu	5	–	2	✗
	freire1	3	3.0	2	✓
	freire2	4	3.7	2	✓
	freire1_int	3	3.2	2	✓
	geo1	4	3.9	2	✓
	geo2	4	3.7	2	✓
	geo3	5	4.7	3	✓
	ps2	3	2.9	2	✓
	ps3	3	3.6	3	✓
	ps4	3	4.0	4	✓
	ps5	3	5.0	5	✓
	ps6	3	5.8	6	✓
	sqrt1	4	5.3	4	✓

7.2 Answer to RQ2: How Can AutoINF Generate a Unified Invariant That Is Valid across All Execution Paths of a Multipath Loop

This experiment evaluates AutoINF on multipath loops where the required invariant can be expressed in a unified, non-disjunctive form. In such cases, the correctness of the inferred invariant must be ensured across all execution paths, which requires systematically exploring each path. AutoINF inherently supports such verification through its loop model, which tracks execution paths independently. This enables inferring and validating a non-disjunctive invariant while still accounting for the distinct behaviors of multiple paths. Table 4 presents the evaluation results on multipath loop benchmarks whose invariants can be expressed as a single (non-disjunctive) formula. Although these loops contain multiple execution branches, a unified invariant holds across all paths, making them a suitable test for verifying AutoINF's ability to both infer and validate an invariant in a path-sensitive setting. In general, AutoINF constructs an invariant template over the set of loop variables and attempts to solve it using the traces extracted from all execution paths. For the benchmarks reported in Table 4, AutoINF successfully solved the constructed template, producing a candidate invariant. This candidate is then validated against all loop paths using the path-sensitive model, ensuring that the inferred invariant holds globally across the loop.

The table reports the number of program variables (`VAR`), the total inference time in seconds (`T`), and the degree of the inferred invariant (`D`). AutoINF successfully inferred the correct invariant for 10 out of 11 benchmarks. The only exception was `egcd`, which failed to converge within the maximum refinement limit due to a fundamental limitation in our current prototype when dealing with dense conjunctive structures. While AutoINF is highly optimized for path-isolated disjunctive reasoning, it currently lacks a global template-merging pass to recombine isolated path constraints. It therefore cannot derive the tightly coupled conjunction of multi-variable linear relations.

Table 4: Experimental results for multipath loop benchmarks with a single (non-disjunctive) invariant. (✓: success, ✗: failure).

Prog.	VAR	T	D	Correct
H22	5	6.3	3	✓
H42	3	5.8	3	✓
H20	3	5.8	1	✓
lcm2	3	4.6	2	✓
mannadiv	3	3.9	2	✓
fermat2	3	4.0	2	✓
bresenham	3	4.5	2	✓
egcd	8	–	2	✗
Popl09_fig2_2	4	5.6	3	✓
Popl09_fig3_4	3	4.4	4	✓
Pldi09_fig4_5	3	3.8	2	✓

7.3 Answer to RQ3: How Effectively Can AutoINF Generate Disjunctive Invariants That Accurately Capture Incompatible Execution Paths, Including Nonlinear Behaviors?

In this experiment, we evaluate AutoINF's effectiveness in inferring disjunctive invariants. AutoINF generates disjunctive invariants in two main scenarios: (1) when the collected loop traces are insufficient to solve the constructed invariant template; and (2) when the refinement process fails to produce a unified invariant that holds across all loop paths, and violations are detected in any path. In both scenarios, AutoINF

analyzes each loop path independently, inferring a separate invariant for each state, which may capture linear or nonlinear behavior. These individual invariants are then combined into a disjunctive form that accurately preserves the branching structure and semantics of the loop.

Table 5 presents the experimental results for AutoINF on benchmarks requiring disjunctive invariant inference. The benchmarks are divided into linear and nonlinear categories. For each benchmark, the table reports the number of variables involved in the loop `VAR`, the total execution time in seconds `T` for the invariant inference process, and a correctness indicator `Correct` showing whether AutoINF successfully inferred a valid disjunctive invariant ✓or failed ✗.

The results demonstrate that AutoINF successfully infers disjunctive invariants for the majority of benchmarks, covering both linear and nonlinear cases. As shown in Table 5, the tool efficiently generates and validates invariants across multiple execution paths, with inference times typically below 11 s. These results highlight AutoINF's ability to reason about multipath loops that require disjunctive invariant forms, while maintaining practical runtime performance. Collectively, the findings confirm the effectiveness of the proposed path-sensitive approach in handling complex branching behaviors.

Table 5: Experimental results for disjunctive invariant Inference. (✓: success. ✗: failed).

Benchmark	Prog.	VAR	T	Correct
Linear	H08	2	7.3	✓
	H10	2	7.1	✓
	H13	2	7.0	✓
	H14	2	6.8	✓
	H21	2	6.2	✓
	H32	2	5.8	✓
	H42	3	6.2	✓
	H43	3	7.2	✓
	H46	3	7.6	✓
	Popl09_fig2_1	3	6.8	✓
	Popl09_fig4_1	3	7.1	✓
	Popl09_fig4_3	3	7.3	✓
	INF1	2	7.4	✓
	INF2	2	6.8	✓
	INF3	2	7.1	✓
Nonlinear	H08	2	7.3	✓
	INF4	2	9.3	✓
	INF5	2	9.6	✓
	INF6	2	10.2	✓
	INF7	2	8.7	✓
	INF8	2	9.2	✓
	INF9	2	9.5	✓
	INF10	2	9.2	✓
	INF11	2	10.6	✓
	INF12	2	10.8	✓

7.4 Answer to RQ4: How Does AutoINF Compare to Other Invariant Inference Tools?

We compare AutoINF with three related approaches: *"SymInfer"* [17], *"G-CLN"* [13] and *"NumInv"* [42].

NumInv: builds on DIG's algorithms to infer numerical invariants and utilizes the KLEE tool for symbolic execution to validate them. It supports C programs and handles both equalities and octagonal inequalities. NumInv achieved results comparable to AutoINF on the SPL benchmarks. However, its performance degrades significantly in programs with a large number of variables due to KLEE timing out. Additionally, NumInv does not support the inference of disjunctive invariants.

G-CLN: is a tool that employs a Gated Continuous Logic Networks (G-CLN) to learn candidate invariants from program traces, relying on user specifications (e.g., postconditions) to validate them. The approach is primarily designed to infer nonlinear invariants and has been evaluated on only the nonlinear benchmark. This approach does not support inference of the disjunctive invariant, so it fails to deal with most of our MLP benchmarks.

SymInfer: introduces a CEGIR-based approach that leverages symbolic states as an abstraction layer to enable the automated inference of complex loop invariants. It uses symbolic states not only to generate candidate invariants but also to check, refute, and iteratively refine them. A key feature of SymInfer is its ability to encode large sets of concrete program states at user-specified program locations, from which symbolic states of numerical variables within the loop are collected for analysis. SymInfer uses max/min to infer the disjunctive invariants. This approach achieves comparable results on single-path loops; however, the max/min operations can only represent certain classes of disjunctions and are generally effective for capturing simple numerical ranges. It struggles with logical relationships that don't align with ordered comparisons. Collapsing disjunctions into max/min expressions often results in the loss of explicit path conditions, thereby obscuring the underlying program logic. Additionally, many disjunctive properties form non-convex sets that max/min can only approximate, leading to less precise invariants. Finally, these operations are limited in expressing invariants where different variables become relevant under different conditions, that an issue commonly encountered in complex, branching code. As a result, the symInfer fails to infer 24 of 42 multipath loops that the disjunctive invariant is essential to be inferred.

As shown in Table 6, AutoINF successfully infers more complex disjunctive invariants. In contrast, NumInv and G-CLN do not support disjunctive invariant inference, and SymInfer only infers them using max/min expressions. In practice, only the SymInfer tool was available for direct evaluation; for the others, our analysis is based solely on their published algorithms and evaluations.

Table 6: Comparison of AutoINF results with other approaches. (✗: not supported).

	Benchmark	SymInfer [17]	NumInv [42]	G-CLN [13]	AutoINF
SL	Linear Invariant	12/12	12/12	12/12	12/12
	Non-linear Invariant	14/14	11/14	11/14	13/14
MPL	Non-disjunctive Invariant	10/11	10/11	10/11	10/11
	Disjunctive Invariant	✗	✗	✗	24/24

In addition to the comparison of results, which shows that all except fail to infer disjunctive invariants, there are also important technical differences in their implementations. These include reliance on trace collection, symbolic execution scalability, support for refinement, and how each approach handles invariant validation. Such factors significantly impact their effectiveness and applicability to complex loop structures. Table 7 presents a comparative analysis of the approaches and AutoINF. The table highlights that

AutoINF offers the highest level of automation, requiring no manual effort, unlike SymInfer and NumInv, which require inserting `vtrace` at the locations of interesting loops. While SymInfer and NumInv rely on symbolic states to validate invariants, and G-CLN depends on postcondition checking over input traces, AutoINF uses a fully SMT-encoded representation of loop semantics for validation. Furthermore, AutoINF demonstrates superior scalability, supporting both a large number of loop paths and variables, as well as handling complex control flow where different paths involve different variables. In contrast, the other tools either fail to support these features or suffer from limitations such as path explosion or a lack of refinement mechanisms.

Table 7: Technical comparison of AutoINF with existing invariant inference approaches. (✗: not supported.).

Property	SymInfer [17]	NumInv [42]	G-CLN [13]	AutoINF
Automation level	Semi-automatic (requires instrumentation)	Semi-automatic (requires instrumentation)	Low (requires loop traces)	Fully automatic pipeline
Invariant soundness guarantee	High (symbolic states)	High (symbolic states)	Trace-dependent	High (use the path-sensitive model)
Handling multiple loop paths	✗ (path explosion in symbolic execution)	✗	✗	✓ (path-sensitive PDA model)
Disjunctive invariants	Limited (max/min-based forms only)	✗	✗ (difficult to generalize across paths)	✓ (separate invariants per path)
Scalability in the number of variables	Limited by SMT complexity	Limited by symbolic execution (timeouts reported)	Limited (trace-based learning)	✓ (template-driven, constraint-guided search)
Refinement/CEGIS support	✓	✓	✗	✓
Invariant validation method	Symbolic state validation	Symbolic state validation	Feature-based statistical validation	Constraint-based validation over path-sensitive semantics

8 Threats to Validity

AutoINF currently infers invariants using polynomial equality templates of bounded degree, which constrains the class of invariants that can be expressed and verified. While the framework supports nonlinear inequalities, it does not handle conjunctions of multiple polynomial relations or disjunctions within a single execution path. These constraints affect both the types of programs that AutoINF can handle and the range of properties that can be captured, even for programs that are otherwise supported, including cases where syntactically simple loops require composite or disjunctive invariants.

Additionally, variables whose values depend on other changing variables across loop iterations, which are called non-induction variables, cannot be computed directly using closed-form expressions. Instead, their values must be determined by executing the loop path step by step, as each iteration depends on the current state of other variables. This requirement increases the computational effort needed to generate execution traces for invariant inference and may impact scalability for loops with complex interdependent variables, although it does not compromise the correctness of the inferred invariants.

Finally, we note a limitation regarding our empirical baseline evaluations: while SymInfer was locally deployed and directly executed on our hardware environment under identical constraints, the performance metrics for NumInv and G-CLN were extracted directly from their published literature. Because these external results were obtained under different hardware architectures, timeout thresholds, and potentially varied benchmark subsets, they do not constitute a direct side-by-side experimental comparison. Accordingly, these metrics are provided solely as a contextual reference for the current state-of-the-art.

9 Conclusion

This paper addresses a fundamental limitation of existing loop invariant inference techniques in handling multipath loop semantics. While prior static, dynamic, and CEGIR-based approaches have demonstrated effectiveness for single-path loops, their lack of explicit path-sensitive reasoning restricts their ability to infer sound and expressive invariants for loops with complex control flow. In particular, the absence of path-complete validation and refinement hinders the synthesis of disjunctive invariants required to capture incompatible path behaviors. To overcome this limitation, we proposed a path-sensitive loop modeling framework based on Path Dependency Automata (PDA), which represents each execution path as an independent semantic entity and encodes path-specific variable updates as mathematical constraints. Building on this model, we introduced AutoINF, a CEGIR-based invariant inference framework that employs a unified path-sensitive representation throughout both inference and verification. By generating path-specific traces and validating candidate invariants across all feasible execution paths using SMT solving, AUTOINF systematically infers sound, expressive, and disjunctive numerical invariants.

The results demonstrate that explicitly modeling multipath loop semantics prior to invariant inference is critical for achieving accurate and reliable verification. By combining path-sensitive modeling with counterexample-guided refinement, AutoINF advances the state of the art in automated loop invariant inference, particularly for programs with complex control flow and path-dependent behaviors.

While the current prototype is limited to fixed-degree polynomial templates, path-isolated reasoning without global conjunctive merging, and flat loop structures, it establishes a strong foundation for disjunctive invariant synthesis. Future work will extend the framework to support richer invariant templates, including dense conjunctive and inequality-based forms. In addition, we plan to generalize the path dependency architecture to handle nested loop structures by explicitly modeling the inter-path dependencies that arise between inner and outer loops.

Acknowledgement: The authors would like to express their sincere gratitude to all individuals and institutions whose support and contributions made this research possible. The authors also gratefully acknowledge the support provided by Imam Mohammad Ibn Saud Islamic University through the Deanship of Scientific Research under grant number IMSIU-DDRSP2603.

Funding Statement: This work was supported and funded by the Deanship of Scientific Research at Imam Mohammad Ibn Saud Islamic University (IMSIU) (grant number IMSIU-DDRSP2603).

Author Contributions: Abeer S. Hadad: Conceptualization, Methodology, Software, Formal Analysis, Writing—Original Draft Preparation. Fahman Saeed: Methodology, Validation, Investigation, Formal Analysis, Data Curation, Writing—Review & Editing. Adeeb A. Ahmed: Formal Analysis, Validation, Visualization, Writing—Review & Editing, Correspondence. Jiangbin Zheng: Conceptualization, Methodology, Supervision, Project Administration, Writing—Review & Editing. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The datasets are available from the corresponding author upon reasonable request.

Ethics Approval: This study did not involve human participants, animals, or sensitive personal data; therefore, ethics approval was not required.

Conflicts of Interest: The authors declare no competing interests.

References

1. Paul S, Cruz E, Dutta A, Bhaumik A, Blasch E, Agha G, et al. Formal verification of safety-critical aerospace systems. IEEE Aerosp Electron Syst Mag. 2023;38(5):72–88. doi:10.1109/maes.2023.3238378.
2. Yoo J, Jee E, Cha S. Formal modeling and verification of safety-critical software. IEEE Softw. 2009;26(3):42–9. doi:10.1109/ms.2009.67.
3. Goel AL. Software reliability models: assumptions, limitations, and applicability. IEEE Trans Softw Eng. 1985;SE-11(12):1411–23. doi:10.1109/tse.1985.232177.
4. Everett W, Keene S, Nikora A. Applying software reliability engineering in the 1990s. IEEE Trans Reliab. 1998;47(3):SP372–8.
5. Hoare CAR. Proof of a program: find. Commun ACM. 1971;14(1):39–45. doi:10.1007/978-1-4612-6315-9_10.
6. Cousot P, Cousot R. Abstract interpretation frameworks. J Log Comput. 1992;2(4):511–47. doi:10.1093/logcom/2.4.511.
7. Cousot P. Abstract interpretation. ACM Comput Surv. 1996;28(2):324–8. doi:10.1145/234528.234740.
8. McMillan KL. Lazy abstraction with interpolants. In: Proceedings of the Computer Aided Verification: 18th International Conference, CAV 2006; 2006 Aug 17–20; Seattle, WA, USA. p. 123–36.
9. McMillan KL. Interpolation and SAT-based model checking. In: Proceedings of the Computer Aided Verification: 15th International Conference, CAV 2003; 2003 Jul 8–12; Boulder, CO, USA. p. 1–13.
10. Ernst MD, Cockrell J, Griswold WG, Notkin D. Dynamically discovering likely program invariants to support program evolution. In: Proceedings of the 21st International Conference on Software Engineering; 1999 May 22; Los Angeles, CA, USA. p. 213–24.
11. Ernst MD. Dynamically detecting likely program invariants [dissertation]. Washington, DC, USA: University of Washington; 2000.
12. Ernst MD, Perkins JH, Guo PJ, McCamant S, Pacheco C, Tschantz MS, et al. The Daikon system for dynamic detection of likely invariants. Sci Comput Program. 2007;69(1–3):35–45. doi:10.1016/j.scico.2007.01.015.
13. Yao J, Ryan G, Wong J, Jana S, 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; 2020 Jun 15–20; London, UK. p. 106–20.
14. Krishna S, Puhrsch C, Wies T. Learning invariants using decision trees. arXiv:1501.04725. 2015.
15. Sharma R, Aiken A. From invariant checking to invariant inference using randomized search. Form Methods Syst Des. 2016;48(3):235–56. doi:10.1007/978-3-319-08867-9_6.
16. Lu H, Gui J, Wang C, Huang H. A novel data-driven approach for generating verified loop invariants. In: Proceedings of the 2020 International Symposium on Theoretical Aspects of Software Engineering (TASE); 2020 Dec 11–13; Hangzhou, China. p. 9–16.
17. Nguyen T, Nguyen K, Dwyer MB. Using symbolic states to infer numerical invariants. IEEE Trans Software Eng. 2022;48(10):3877–99. doi:10.1109/tse.2021.3106964.

18. Xie X, Chen B, Liu Y, Le W, Li X. Proteus: computing disjunctive loop summary via path dependency analysis. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering; 2016 Nov 13–18; Seattle, WA, USA. p. 61–72.
19. Xie X, Chen B, Zou L, Liu Y, Le W, Li X. Automatic loop summarization via path dependency analysis. IEEE Trans Softw Eng. 2019;45(6):537–57. doi:10.1109/tse.2017.2788018.
20. Lin Y, Zhang Y, Chen S, Song F, Xie X, Li X, et al. Inferring loop invariants for multi-path loops. In: Proceedings of the 2021 International Symposium on Theoretical Aspects of Software Engineering (TASE); 2021 Aug 25–27; Shanghai, China. p. 63–70.
21. Colón MA, Sankaranarayanan S, Sipma HB. Linear invariant generation using non-linear constraint solving. In: Proceedings of the Computer Aided Verification: 15th International Conference, CAV 2003; 2003 Jul 8–12; Boulder, CO, USA. p. 420–32.
22. Gulwani S, Srivastava S, Venkatesan R. Constraint-based invariant inference over predicate abstraction. In: Proceedings of the Verification, Model Checking, and Abstract Interpretation: 10th International Conference, VMCAI 2009; 2009 Jan 18–20; Savannah, GA, USA. p. 120–35.
23. Gupta A, Rybalchenko A. InvGen: an efficient invariant generator. In: Proceedings of the Computer Aided Verification: 21st International Conference, CAV 2009; 2009 Jun 26–Jul 2; Grenoble, France. p. 634–40.
24. Sharma R, Dillig I, Dillig T, Aiken A. Simplifying loop invariant generation using splitter predicates. In: Proceedings of the Computer Aided Verification: 23rd International Conference, CAV 2011; 2011 Jul 14–20; Snowbird, UT, USA. p. 703–19.
25. Rodríguez-Carbonell E, Kapur D. Automatic generation of polynomial loop invariants: algebraic foundations. In: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation; 2004 Jul 4–7; Santander, Spain. p. 266–73.
26. Păsăreanu CS, Visser W. Verification of Java programs using symbolic execution and invariant generation. In: Proceedings of the International SPIN Workshop on Model Checking of Software; 2004 Apr 1–3; Barcelona, Spain. p. 164–81.
27. Siegel SF, Zirkel TK. Loop invariant symbolic execution for parallel programs. In: Proceedings of the International Workshop on Verification, Model Checking, and Abstract Interpretation; 2012 Jan 22–24; Philadelphia, PA, USA. p. 412–27.
28. Csallner C, Tillmann N, Smaragdakis Y. DySy: dynamic symbolic execution for invariant inference. In: Proceedings of the 30th International Conference on Software Engineering; 2008 May 10–18; Leipzig, Germany. p. 281–90.
29. Chalupa M, Strejček J. Backward symbolic execution with loop folding. In: Proceedings of the Static Analysis: 28th International Symposium, SAS 2021; 2021 Oct 17–19; Chicago, IL, USA. p. 49–76.
30. Zhang L, Yang G, Rungta N, Person S, Khurshid S. Feedback-driven dynamic invariant discovery. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis; 2014 Jul 21–26; San Jose, CA, USA. p. 362–72.
31. Nguyen TV, Kapur D, Weimer W, Forrest S. Using dynamic analysis to discover polynomial and array invariants. In: Proceedings of the 2012 34th International Conference on Software Engineering (ICSE); 2012 Jun 2–9; Zurich, Switzerland. p. 683–93.
32. Nguyen T, Kapur D, Weimer W, Forrest S. DIG: a dynamic invariant generator for polynomial and array invariants. ACM Trans Softw Eng Methodol. 2014;23(4):1–30.
33. Nguyen TV, Kapur D, Weimer W, Forrest S. Using dynamic analysis to generate disjunctive invariants. In: Proceedings of the 36th International Conference on Software Engineering; 2014 May 31–Jun 7; Hyderabad, India. p. 608–19.
34. Galeotti JP, Furia CA, May E, Fraser G, Zeller A. Inferring loop invariants by mutation, dynamic analysis, and static checking. IEEE Trans Softw Eng. 2015;41(10):1019–37. doi:10.1109/tse.2015.2431688.
35. Bao J, Trivedi N, Pathak D, Hsu J, Roy S. Data-driven invariant learning for probabilistic programs. In: Proceedings of the International Conference on Computer Aided Verification; 2022 Aug 7–10; Haifa, Israel. p. 33–54.

36. Kamath A, Senthilnathan S, Chakraborty P, Deligiannis SK, Lahiri A, Lal A, et al. Finding inductive loop invariants using large language models. arXiv:2311.07948. 2023.
37. Pei K, Bieber D, Shi K, Sutton C, Yin P. Can large language models reason about program invariants? In: Proceedings of the International Conference on Machine Learning; 2023 Jul 23–29; Honolulu, HI, USA. p. 27496–520.
38. Janßen C, Richter C, Wehrheim H. Can ChatGPT support software verification? In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering; 2024 Apr 6–11; Luxembourg City, Luxembourg. p. 266–79.
39. Ma L, Liu S, Li Y, Xie X, Bu L. SpecGen: automated generation of formal program specifications via large language models. In: Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE); 2025 Apr 26–May 6; Ottawa, ON, Canada. p. 16–28.
40. Wen C, Cao J, Su J, Xu Z, Qin S, He M, et al. Enchanting program specification synthesis by large language models using static analysis and program verification. In: Proceedings of the International Conference on Computer Aided Verification (CAV); 2024 Jul 24–27; Montreal, QC, Canada. p. 302–28.
41. Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the control dependence graph. ACM Trans Program Lang Syst. 1991;13(4):451–90. doi:10.1145/115372.115320.
42. Nguyen TV, Dwyer MB, Visser W. Symlnfer: inferring program invariants using symbolic states. In: Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE); 2017 Oct 30–Nov 3; Urbana, IL, USA. p. 804–14.