P4SymTest: A Framework for Modular Verification and Early Bug Detection for P4 Programs

Vitor Camargo de Moura, Eder John Scheid, Lisandro Z. Granville, Alberto E. Schaeffer-Filho
Federal University of Rio Grande do Sul, Porto Alegre, Brazil
{vcmoura, ejscheid, granville, alberto}@inf.ufrgs.br

***Abstract*—Data plane programming in P4 has introduced unprecedented flexibility to networks; yet, it has also amplified network complexity, making program verification and bug detection challenging tasks. To manage this complexity, we seek inspiration from software engineering principles, where modularity and iterative testing are fundamental for ensuring correctness. This paper introduces P4SymTest, a framework that applies these principles to P4 verification through modular symbolic execution. P4SymTest decomposes program analysis, allowing developers to verify parsers and match-action tables independently, treating each component as a testable unit. By reusing symbolic states between stages, it provides rapid feedback and mitigates path explosion. Our approach transforms data plane verification into a scalable and accessible process, integrating it seamlessly into the software development lifecycle. The effectiveness of P4SymTest is demonstrated through the analysis of synthetic P4 programs, where the framework successfully performed component analyses in milliseconds and exhaustive full program analyses in a few minutes for P4 programs with up to 8 parser states and 14 match-action tables.**

***Index Terms*—Programmable Data plane, P4, Software Verification, Symbolic Execution, Modularity, Compositional Symbolic Execution**

I. Introduction

Recent advances in Programmable Data Planes (PDPs) have redefined the capabilities of modern networks, shifting packet processing from fixed-function hardware to flexible, software-defined logic [1]. This programmability enables the rapid deployment of new protocols, fine-grained forwarding control, and greater adaptability of network infrastructures in different applications, such as telemetry, in-network Computation, distributed consensus, and load balancing [2].

However, despite their potential, the adoption of PDPs remains challenging due to the difficulties in ensuring correctness and reliability. For example, programming data planes in languages such as P4 [3] requires reasoning at a low level of abstraction, where small mistakes in parsers, tables, or control flows can propagate and disrupt network-wide behavior [4]. Further, the coupling between data plane code, control plane configuration, and hardware constraints amplifies this complexity, creating a wide space for subtle bugs. Thus, ensuring correctness in such a context demands systematic, early-stage code verification.

On the one hand, traditional software engineering practices (*e.g.*, Test-Driven Development (TDD) [5], continuous integration, and unit testing) have long mitigated complexity in conventional software development. On the other hand, bringing these principles to PDP development remains challenging, as meaningful tests typically require complete environments with switches, control software, and realistic traffic, which hinders incremental validation. Yet, adopting such practices holds significant potential. If supported by *modular verification* techniques capable of analyzing PDP programs incrementally, they could enable systematic and iterative testing; hence, fostering reliable and maintainable software development for PDPs.

In this sense, existing research efforts, including *symbolic execution*-based frameworks [6], [7] and automated test generation approaches [8], [9], have advanced PDP verification. Still, they are rarely integrated with iterative coding workflows and usually demand complete programs, external specifications, or custom assertion languages that provide results detached from the developer environment, which leads to scalability issues and complex pipelines. As a result, the debugging of PDP code remains fragmented and inefficient.

In this paper, we present P4SymTest, a framework for early bug detection in P4 programs that utilizes modular symbolic execution to decompose verification into manageable, independent tasks. By decoupling the analysis of parsers, deparsers, and match-action tables through persistent snapshots and path merging at table granularity, P4SymTest avoids the redundant exploration of identical symbolic paths and mitigates the path explosion problem [10] while maintaining full pipeline consistency. Beyond these efficiency gains, this modular design enables a developer-guided workflow, empowering users to orchestrate the analysis by prioritizing specific program fragments and interpreting symbolic outputs via an interface. By allowing for the precise inspection of intermediate snapshots to identify undesired behaviors, P4SymTest effectively bridges the gap between formal guarantees and the practical, iterative requirements of P4 software development.

The remainder of this paper is organized as follows. Section II provides background on PDPs, software engineering, and verification techniques. Section III surveys advances in P4 verification and symbolic execution. Section IV details the design of our modular verification framework. Section V presents evaluation results. Finally, Section VI outlines the concluding remarks and directions for future work.

II. Background

This section presents PDPs and the P4 language as the foundation of our framework, linking software engineering principles (*e.g.*, modularity and TDD) to our verification approach.

It concludes with an overview of verification techniques, emphasizing Compositional Symbolic Execution (CSE), which drives our scalable analysis model.

A. Programmable Data Planes

PDPs shift packet processing from fixed-function hardware to programmable logic. This flexibility allows operators to implement custom behaviors, support new protocols, and deploy policies directly in the data plane while preserving line-rate performance [11]. As a result, network design and updates become faster and more adaptable.

P4 [3] is the dominant language for programming PDPs. It follows the Protocol Independent Switch Architecture (PISA), which divides processing into a *parser*, ingress and egress *pipelines*, and a *deparser*. The parser extracts headers; the pipelines apply match-action tables that update metadata and determine forwarding actions; and the deparser rebuilds the packet. Together, these components define the packet-processing behavior of a device.

Although P4 is target-independent, real deployments face hardware constraints such as memory limits, pipeline depth, and table dependencies. The language's flexibility also increases the chance of subtle semantic or configuration errors. Despite progress in tooling, the ecosystem still lacks mature debugging and verification support. Recent efforts to overcome monolithic P4 development through modular extensions and program composition [12], [13] further underscore the need for efficient and versatile verification. These factors motivate early, modular validation to help developers understand and test behavior before deployment.

B. Software Engineering for Testing and Validation

Software engineering provides systematic methods to improve correctness, robustness, and maintainability in complex systems. Practices such as modularity, unit testing, and TDD help control complexity and detect defects early [14], [15]. By decomposing software into small components with clear interfaces, developers can reduce the reasoning space and make verification and debugging more manageable.

TDD encourages writing tests before implementation, following the *red–green–refactor* cycle [5]. This process clarifies specifications, provides immediate feedback, and supports the iterative refinement of code and design. Unit tests complement TDD by checking each module in isolation [16], verifying correctness before integration. Frameworks like JUnit [17], along with mocks, stubs, and dependency injection [15], [18], enable automated, repeatable testing workflows.

Beyond correctness, these practices improve comprehension and evolution: modularity simplifies reasoning, and automated tests increase confidence when modifying or refactoring code. Figure 1 illustrates this idea: verifying small, well-defined components catches errors closer to their source and accelerates feedback. We apply the same principle to P4 verification by treating parsers, match-action tables, and the deparser as independent units. This enables incremental analysis and reuse of intermediate results, extending established testing methods to the data plane.

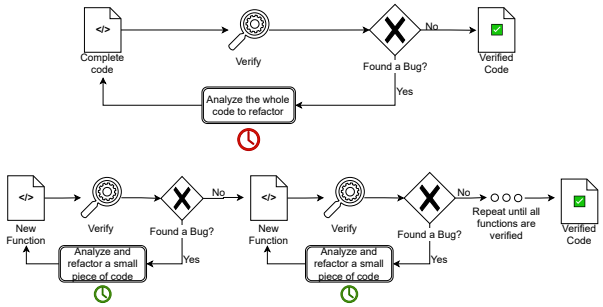

Fig. 1. Benefit of modular verification for early and focused feedback.

C. Verification Techniques

Several techniques aim to ensure program correctness. Formal verification provides mathematical rigor [19], [20], but it is computationally expensive and often requires manual modeling. Runtime assertions and conventional testing [7], [21] offer faster feedback but only cover behaviors chosen by the tester. Automated test generation, including fuzzing and reinforcement learning [8], [9], increases coverage but still depends on concrete execution, which scales poorly in complex PDP pipelines.

Symbolic Execution (SE) [22] treats inputs as symbols and collects constraints along execution paths. It exposes corner cases missed by concrete testing [23] and requires no hardware environment, making it suitable for early PDP validation. Its main limitation is *path explosion*, where the number of feasible paths grows exponentially with program size.

Dynamic Symbolic Execution (DSE) [24] mitigates this by using concrete execution for guidance; however, it is less suitable for PDPs, which lack a conventional runtime. Compositional Symbolic Execution (CSE) [25]–[27] improves scalability by analyzing modules independently and reusing summaries of their input-output behavior. Once pre- and post-conditions are derived, they can replace re-execution in later analyses, reducing redundancy and controlling path growth.

This compositional approach aligns well with P4 verification. As shown in Figure 2, analyzing smaller units greatly reduces the number of paths explored. Our framework follows this principle, performing incremental, table-by-table symbolic analysis, enabling scalable reasoning similar to structured, test-driven software workflows.

III. Related Work

Next, we examine state-of-the-art symbolic execution addressing path explosion via modularity and compositional reasoning, followed by a review of P4 verification techniques.

A. Enhanced Symbolic Execution

Efforts to improve symbolic execution often focus on modularity and compositional reasoning to make the analysis more scalable and interpretable. Gillian [28] is a parametric framework that unifies correctness and incorrectness reasoning across languages such as JavaScript, C, and Rust. It supports whole-program symbolic testing and semi-automatic modular

verification via separation logic. Similarly, VeriFast [29] applies separation logic for function-level reasoning in C and Java with manual annotations and explicit symbolic resource management, trading automation for stronger guarantees.

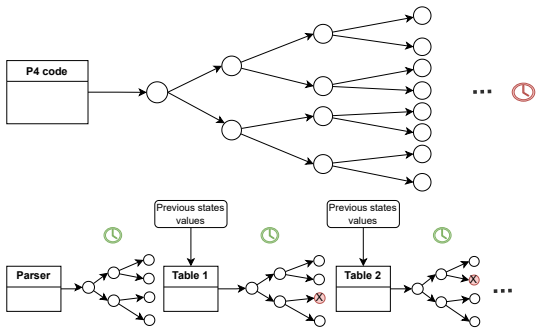

Fig. 2. Effect of compositional analysis on controlling path explosion.

A common limitation in these systems is their focus on general-purpose programming, where modularity is applied at the function level. MACKE [30], for instance, targets vulnerability detection by applying compositional symbolic execution through KLEE: it explores isolated C functions and then merges feasible paths across inter-procedural boundaries. This path-feasibility-based combination relates to our approach, although we apply it inversely: infeasible paths are pruned before subsequent steps, ensuring that only valid symbolic states are propagated.

Other works extend modular reasoning to different domains. Troika [31] partitions web layouts into independently verified components, caching results to accelerate re-analysis. Although the domain differs, its incremental philosophy mirrors ours. SymNet [32] adapts symbolic execution to networking through SEFL, abstracting packet headers and metadata to control path growth. Our framework builds on this idea while integrating symbolic reasoning into an iterative development workflow, enabling table-by-table exploration, state snapshots, and early pruning of infeasible paths rather than deferring verification to a post hoc stage.

B. Verification in P4

Several research efforts bring verification capabilities to P4, ranging from formal reasoning to automated testing and runtime checking [7]–[9], [33]. Formal tools such as p4v [34] use SMT solvers to reason about control and data plane consistency but face scalability and usability challenges. ASSERT-P4 [7], [33] embeds assertions into P4 code, combining compile-time checking with symbolic exploration, even though it requires manual annotations and lacks incremental verification.

Symbolic execution has also been applied more directly. Vera [6] verifies P4 snapshots through NetCTL specifications, offering expressiveness but limited coverage due to user-defined assertions. P4pktgen [8] generates concrete test cases symbolically but struggles to scale, while P4RL [9] employs reinforcement learning to improve path diversity, sacrificing interpretability and control over the symbolic space.

Beyond verification, systems such as BF4 [35] offer runtime safety checks and automated repair, bridging compile-time and runtime assurance, although they rely on specialized configurations and remain detached from continuous development.

Overall, two trends emerge: (1) formal and symbolic methods provide rigor but suffer from scalability and usability issues, and (2) testing and learning-based methods improve practicality but lack exhaustive guarantees. Our framework bridges these extremes by combining symbolic reasoning with modular, interactive analysis. Through independent verification of components, it delivers early feedback and supports a test-driven workflow that integrates verification into the P4 development cycle rather than treating it as a separate phase.

IV. P4SymTest Design

This section presents P4SymTest, our framework for the modular symbolic execution of P4 programs. Its design emphasizes interactivity and incremental analysis: symbolic states can be reused, pruned, or rolled back at any stage. We first outline the execution cycle, then describe the transformation from P4 source to symbolic form, and finally detail the mechanisms for state management and path pruning.

A. Overview

P4SymTest is guided by two principles: providing rapid visualization of execution results to facilitate early error detection and pruning infeasible paths to prevent invalid states from propagating. These principles shape the iterative, cyclic workflow at the framework's core.

Figure 3 shows the overall view of P4SymTest. Execution begins with the compilation of the P4 source code (*step 1*), which produces the intermediate representation used by the framework. From this representation, the parser structure is extracted and expressed as a Finite State Machine (FSM) (*step 2*). The FSM is then symbolically executed (*step 3*), exploring all feasible parsing paths and generating symbolic packets that represent possible header configurations and metadata values. From these initial symbolic states, match-action tables can be executed on demand (*step 4*). Table match conditions are encoded as solver queries, feasible matches are identified, and actions are symbolically applied (*step 5*), producing successor states that reflect packet modifications, metadata updates, or control-flow transitions (*step 6*). All executions are recorded as snapshots, forming a checkpoint system that enables branching, rollback, or incremental exploration without recomputing prior stages. The resulting data are then presented in a visualization layer (*step 7*), recording input symbolic packets, path constraints, applied tables and actions, and resulting states. Programmers can inspect symbolic effects, identify unreachable entries, and observe unexpected updates without requiring explicit assertions in the P4 program. After visualization, states corresponding to packet drops or infeasible matches are discarded immediately, reducing unnecessary computation (*step 8*).

Building a modular symbolic execution framework introduces challenges beyond translating code into solver con-

straints, including parsing and modular extraction, usable visualizations, persistent symbolic states, and path explosion mitigation. To illustrate our approach concretely, the next subsection introduces the transformation pipeline from P4 source to symbolic representation.

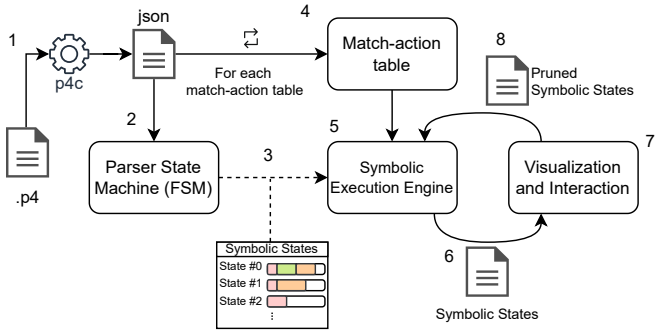

Fig. 3. P4SymTest workflow: state-preserved table execution loop.

B. From P4 Source to Symbolic Representation

A central requirement of P4SymTest's modular verification approach is the ability to analyze fragments of P4 programs independently. This is achieved through a two-step transformation, first, parsing the intermediary P4 JSON representation generated by the `p4c` compiler [36], and second, translating the parsed structures into executable symbolic logic.

1) Parsing and Selective Extraction: When the P4 program is compiled, `p4c` produces a JSON file that captures the essential semantics of the code, removing syntactic noise (*e.g.*, comments, formatting) while exposing structural elements such as parsers, tables, and control blocks. P4SymTest leverages this JSON to isolate and process only the elements relevant to each verification step.

For instance, in the P4 snippet presented in Figure 4 (left), the parser state `parse_ethernet` defines a transition based on the field `ethernet.etherType`. The corresponding entry in the compiler's JSON representation would appear as a set of transition rules mapping possible field values to their destination parser states, explicitly reflecting the branching structure of the source program, as we can see in the upper fragment in the center part of Figure 4.

Unlike traditional compilers that discard intermediate results when encountering errors, P4SymTest is designed to tolerate partially compiled programs. Even when a full build fails, the framework can parse well-formed JSON fragments and provide early feedback on structural components, such as parser transitions or table keys, helping developers diagnose issues before a complete build is achievable.

Once the relevant JSON fragments are extracted, P4SymTest converts them into solver-executable logic to run using the Z3 solver [37]. Each transition, match field, and action is translated into a corresponding logical constraint. Figure 4 (right) shows fragments of the generated constraints for the running example.

Parser transitions become equality or inequality constraints over symbolic bit-vectors. For instance, the condition `ethernet.etherType == ipv4` is encoded as a 16-bit equality. When multiple transitions originate from the same parser state, P4SymTest generates a disjunction of cases, along with a default branch that negates all previous conditions.

Match-action tables follow the same principle. A Longest-Prefix Match (LPM), bottom right fragment of Figure 4, table entry is encoded by masking the destination address and comparing it against the stored prefix. If the match condition is satisfiable, its action is symbolically applied; otherwise, the default action is used. The resulting logic is expressed with nested `If-Then-Else` (ITE) expressions.

Because the P4 data-plane behavior depends on the control-plane state (table entries, routing rules, etc.), which is not available during p4 development, naively symbolizing all possible configurations would be infeasible. To keep exploration tractable, P4SymTest provides a lightweight *mini-topology* interface: developers specify a simplified network layout, and only the corresponding forwarding rules are injected into the symbolic model. For example, specifying a single route `10.0.0.0/24` $\rightarrow$ `port 1` results in one concrete entry being symbolically encoded. This preserves realistic control-plane behavior without requiring full configuration or exhaustive search.

Actions are encoded as direct symbolic updates, where each header or metadata field modified by an action is represented as a Z3 variable. For example, decrementing the IPv4 TTL results in:

$$\texttt{ttl}_{\text{new}} = \texttt{ttl} - 1$$

where `ttl` is an 8-bit symbolic variable. This avoids separate path enumeration while capturing the exact header and metadata transformations.

Overall, the P4 program is translated into a collection of logical constraints that preserve its semantics while enabling symbolic reasoning: headers become bit-vectors, transitions become predicates, and actions become state updates. Combined with topology-driven forwarding rules, this representation forms the basis for parser visualization, state persistence, and modular table execution in P4SymTest.

C. Symbolic State Representation and Persistence

A fundamental challenge in modular symbolic execution is managing state space across decoupled analysis stages. Classical approaches maintain all reachable states in memory throughout end-to-end execution [22]. For our analysis, where parsers, tables, and egress logic may be verified independently, this monolithic approach fails.

P4SymTest addresses this through *checkpoint-based state persistence*. Following symbolic execution theory [23], a symbolic state is defined as $\sigma = (\vec{x}, \Phi, \vec{m})$, where $\vec{x}$ are symbolic variables over header and metadata fields, Φ is a conjunction of path constraints, and $\vec{m}$ is metadata (header validity, history). Rather than maintaining all states in memory, P4SymTest persists the set of feasible symbolic states $S = \{\sigma_1, \ldots, \sigma_n\}$

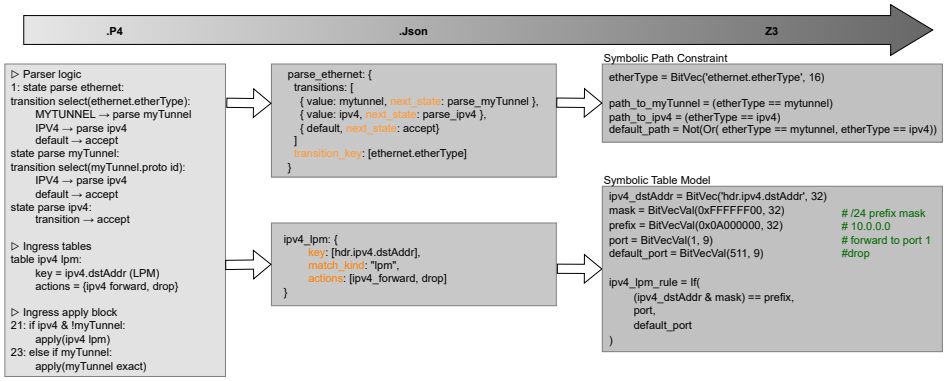

Fig. 4. From P4 pseudocode (left) and its compiled JSON fragment (center) to the corresponding Z3-compatible symbolic logic (right).

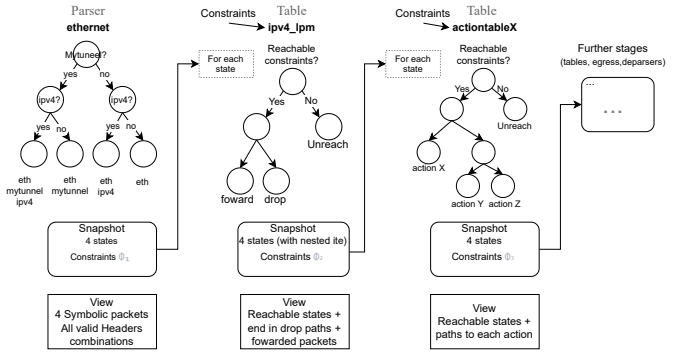

Fig. 5. Representation of a modular iterative verification

at pipeline stages as snapshots. This enables modular reachability: $S_i = \text{reach}_{S_{i-1}}(P_i)$, where subsequent stages begin from persisted checkpoints without recomputation.

Constraints are serialized via the SMT-LIB format [38] to provide lossless, portable representations. Upon entering the next stage, these snapshots are reconstructed as logically equivalent Z3 objects, transforming symbolic execution into an incremental, modular process. By progressively refining states and pruning infeasible paths without end-to-end recomputation (*cf.* Figure 5), this approach reduces memory overhead and enables independent, interactive inspection across the pipeline.

D. Execution Flow and Pruning Mechanisms

Path explosion, the exponential growth of possible execution paths, remains the primary obstacle to symbolic execution [22]. P4SymTest mitigates this through multi-layered pruning applied at each analysis stage. Algorithm 1 formalizes key pruning operations, where C denotes accumulated parser constraints, R denotes reachability conditions, and E denotes table entries. Figure 6 illustrates the complete execution flow for table analysis, showing how symbolic states from parser snapshots are combined with reachability conditions and table entries to produce pruned output states.

1) Layer 1 - Reachability Filtering: Before analyzing each table, P4SymTest performs control-flow reachability analysis by extracting logical conditions (R) from the program's

Algorithm 1 Multi-layered Pruning

Require: C, R, E
Ensure: output_states
where: C = parser constraints, R = reachability conditions, E = table entries, s = symbolic state

▷ **Layer 1: Reachability**
1: **for each** s **do**
2: **if solver($C \wedge R$) = UNSAT then**
3: **PRUNE**(s)

▷ **Layer 2: Path Merging via Nested ITE**
4: **result** $\leftarrow$ **default_action_result**
5: **for each entry** $e \in E$ **(reversed) do**
6: **match** $\leftarrow$ **encode_match(**e**.key)**
7: **result** $\leftarrow$ **`If`(match,** e**.action, result)**

▷ **Layer 3: Drop Detection**
8: **if solver(egress = 511) = SAT then**
9: **PRUNE**(s)
10: **else**
11: **ADD**(s)

▷ **Layer 4: Modular Decomposition**
12: **Call Symbolic_exec_engine(desired_snapshot)**

intermediary JSON. These conditions specify what must be true to reach a given table, based on the ingress apply block (bottom left of Figure 4). The framework then takes a snapshot of symbolic states (C) from the preceding component (e.g., the parser) and checks whether $C \wedge R$ is satisfiable using Z3 (Algorithm 1, lines 1-3). If unsatisfiable, the table is unreachable for that specific symbolic state (packet class), and the state is pruned immediately, eliminating logically impossible paths as depicted in Figure 6. For reachable states, the analysis proceeds to evaluate table entries and actions to identify feasible match outcomes and produce resulting output states, detecting potential forwarding or drop behaviors.

2) Layer 2 - Formula-Based Path Merging: Instead of branching per match outcome, P4SymTest encodes table results into a single symbolic state using nested If-Then-Else (ITE) expressions (lines 4–7). By processing entries in **reverse order**, this bottom-up construction faithfully reproduces P4's

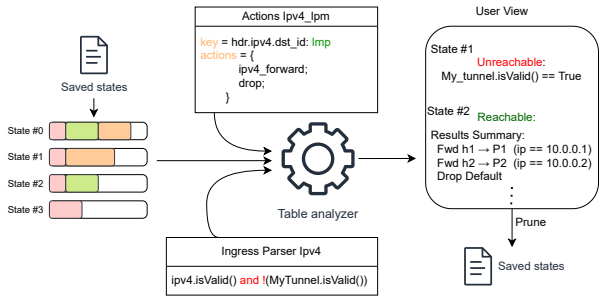

Fig. 6. Execution flow for individual table analysis

priority logic without the overhead of explicit rule negations. Since solvers like Z3 are optimized for nested ITE, P4SymTest efficiently shifts complexity from path enumeration to constraint solving [23], reasoning about the entire table logic as a compact, unified symbolic representation.

3) Layer 3 - Drop Detection and Dead-Path Elimination: After table execution, the framework identifies which packet trajectories lead to drop actions by querying the solver for the special drop port (port 511). States corresponding to explicit drops are excluded from subsequent snapshots (lines 9 and 11), preventing dead paths from propagating to later analysis stages. Non-dropped states are annotated with the constraint `egress_spec` $\neq 511$, ensuring that downstream modules reason only about packets that are not discarded.

4) Layer 4 - Modular Decomposition: Finally, P4SymTest's architecture itself acts as a high-level pruning mechanism. By verifying components independently and reusing symbolic summaries as checkpoints rather than expanding end-to-end paths, the framework avoids redundant recomputation and mitigates combinatorial explosion [25]. Together, these mechanisms of reachability filtering, path merging, drop detection, and modular decomposition maintain scalability and precision.

E. Framework Architecture

The architecture of P4SymTest, illustrated in Figure 7, consists of a Graphical User Interface (Frontend) and an analysis core (Backend), represented by the dotted box in the figure. The verification process begins even before execution: the P4 program is submitted to a compiler, which disassembles it into its logical components (Parser, Ingress, etc.). The graphical interface presents these components, empowering users to select and orchestrate the verification of specific fragments.

When the user selects a component for analysis (e.g., *parser*), the interface sends a request via an API to the analysis core. This core is composed of two main modules: the *State Manager* and the *Symbolic Execution Engine*.

The *Symbolic Execution Engine*, a concept introduced in Figure 3, is represented in the detailed architecture (Figure 7) by the combination of three processes: the execution modules (*run_ methods*), which load the specific P4 component; the translation of the component's logic and table rules into a set of SMT (*Satisfiability Modulo Theories*) constraints; and the execution of the Z3 solver, which processes the constraints to calculate the output states.

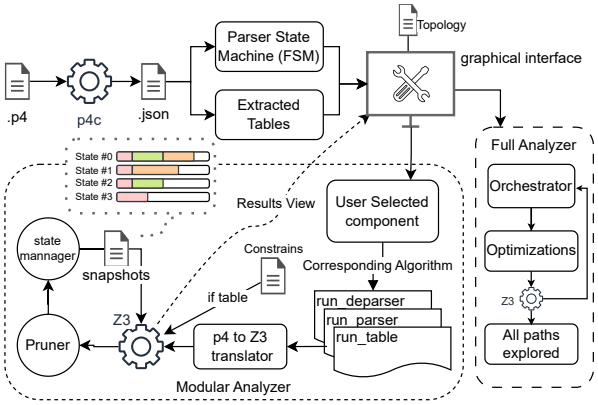

Fig. 7. Full architecture of P4SymTest

Concurrently, the *State Manager* acts as the orchestrating component that manages the data flow between these stages. Communication between the analysis modules is not handled in memory; instead, the *State Manager* persists the resulting symbolic states (the *snapshots*) as JSON files. At the end of a stage's analysis, this manager collects the generated *snapshots*, performs the filtering of unfeasible paths (*pruning*), and provides the correct set of valid *snapshots* as input for the next pipeline stage, ensuring incremental analysis.

The results are then rendered in the Graphical User Interface, which interprets the JSON snapshots to provide a step-by-step trace of each execution path. For every transition, the GUI displays the specific symbolic conditions that were satisfied, allowing the developer to understand the exact logic leading to a given state. To streamline bug detection, P4SymTest incorporates a query-based filtering mechanism. This allows users to verify specific hypotheses, such as whether a certain condition was met at any point or if a single execution path traversed two distinct modules, facilitating the identification of undesired behaviors or side effects.

This focus on a developer-guided, modular workflow highlights a central trade-off of our approach. By allowing the user to filter states and decide which paths to propagate between components, the analysis remains manageable and avoids the state-space explosion typical of monolithic tools, significantly enhancing usability. While this means that full path coverage is not automatically guaranteed in the guided mode, it empowers the developer to focus on critical logic and iteratively build confidence in the program. To address this limitation, P4SymTest also offers an exhaustive verification mode where completeness is prioritized over interactive exploration (discussed next).

1) Full Program Verification: While pruning strategies and controlled component entry points reduce per-table analysis overhead, complete behavioral verification requires evaluating tables using *snapshots* from all execution paths. Figure 8 illustrates this behavior: the left graph shows that only two of five parser-derived paths reach `ipv4`, allowing the remaining states to be disregarded. However, the right graph highlights that convergent tables like `tcp` must still be executed across

multiple paths to cover all cases in which they are reached (e.g., via `ipv4` or `ipv6`)

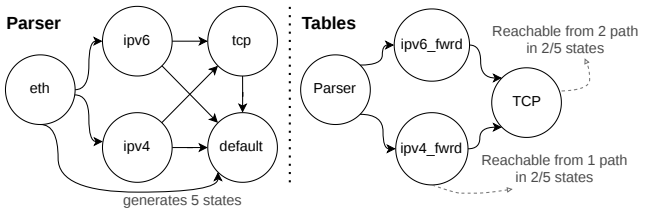

Fig. 8. Reach conditions

To support exhaustive verification, P4SymTest provides an automated mode that traces and executes all possible paths through the program. This leads to exponential state growth, particularly across the ingress and egress pipelines, which may be executed for hundreds of snapshots, and in the deparser, which combines all egress snapshots, thereby reaching thousands of accumulated states.

To mitigate this complexity, two key optimizations leverage our snapshot-based modular architecture. First, during deparser analysis, input states sharing identical header constructions and variable assignments are merged into a single representative state, substantially reducing redundant evaluations without losing coverage. Second, a persistent *cache* stores the results of table executions, allowing identical input states, arising from distinct paths, to reuse previously computed outputs. This avoids re-solving equivalent SMT queries and cuts down on symbolic solver invocations. In both optimizations, snapshots are normalized by temporarily removing path markers before processing and restoring them afterward, ensuring that merged or cached states remain consistent within the overall execution trace.

V. Evaluation

A. Prototype

We implemented a prototype[1] of P4SymTest in Python 3.8, using the `z3-solver` library to handle SMT queries. $P4_{16}$ programs [39] are compiled with `p4c` [36] targeting the BMv2 software switch [40]. The framework processes the JSON output generated by the compiler and performs modular analysis over the parser, table, and deparser components. A React/Vite interface provides interactive visualization. All components, including the compiler and analysis scripts, run inside Docker containers orchestrated with Docker Compose, ensuring reproducibility and ease of deployment.

B. Methodology

To evaluate scalability, we developed a synthetic P4 generator. It creates $P4_{16}$ programs with configurable complexity, varying the number of parser states , ingress and egress tables, headers extracted per parser state, and actions per table.

The generator outputs BMv2-compatible P4 code with basic header definitions, a chained parser with conditional transitions, ingress/egress controls with parameterized match-action tables, and a deparser. It also creates matching topology and runtime configuration files. By adjusting these parameters, we produced a controlled suite of P4 programs to benchmark parser exploration and table execution independently.

All experiments ran within the same Docker environment, orchestrated by Python scripts and automated using PowerShell (.ps1). Execution time and memory usage were collected over 10 runs for statistical relevance, using `psutil`. The results were aggregated with Pandas and plotted using Matplotlib. Tests were run on a machine with an AMD Ryzen 5 5600G CPU and 32GB RAM.

C. Modular Component Analyses

The analysis of isolated components begins with the parser. We utilized synthetically generated P4 programs where the complexity, defined by the number of parser states, was scaled linearly from 3 to 20. As shown in Figure 9, the analysis of execution time reveals a non-linear growth relative to this state count. This is caused by the "path explosion" inherent in symbolic execution; the number of resulting symbolic paths (`parser_output_states`) scales significantly faster than the number of P4 state definitions. This super-linear growth is a direct consequence of the parser's graph structure, where branching logic (e.g., conditional transitions plus a `default: accept` transition in our *synth_p4_generator*) generates a number of exit paths that follow a polynomial relationship (specifically $3N - 5$ for $N \geq 4$ states).

While such path explosion presents a scalability challenge for monolithic verification, its impact is mitigated in our modular system. Realistic parsers are unlikely to scale to a point where analysis time becomes unacceptable. Furthermore, a large fraction of these generated paths will fail to satisfy the header validity conditions required to reach subsequent pipeline tables and will thus be efficiently discarded by the system's pruning layers.

Ingress and egress pipelines have similar behaviors; thus, for our second benchmark, we evaluated a single table (`MyIngress.ingress_table_0`) by varying the number of input symbolic states from the parser (4 to 40) and the table's internal complexity via its actions (2 to 15). As seen in Figure 10, results indicate that execution time (`table_time_avg`) correlates primarily with the number of *input states*, increasing from 0.18s to 0.35s across the tested range, while remaining largely insensitive to the number of *actions* within the table. Even at peak complexity, the analysis remained sub-second, demonstrating efficiency.

Significantly, the number of output states consistently matched the number of reachable input states. This 1:1 mapping is a key architectural benefit, as action effects are encoded via nested conditionals within each state, preventing a secondary state explosion within the table analysis module. This benchmark also highlighted the effectiveness of reachability analysis, which pruned the workload by 50-65% (only 35-50% of parser-generated states required full execution), contingent on the table's `apply` block conditions. More restrictive conditions lead to fewer reachable states.

[1]Source code available at: https://github.com/VitorCamargs/p4symtest

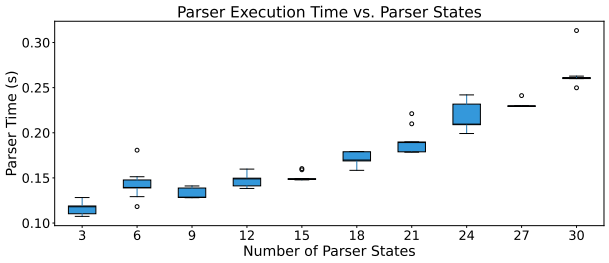

Fig. 9. Parser verification with increasing complexity.

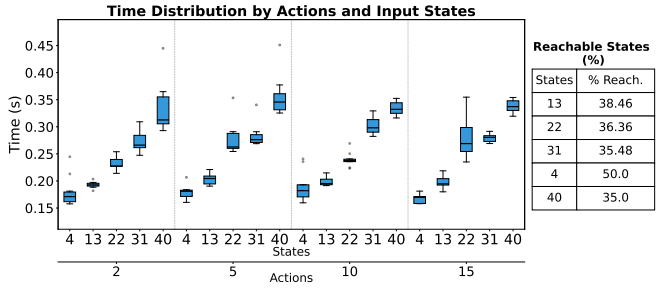

Fig. 10. Table verification with increasing complexity.

D. Full Program Verification

We evaluated the exhaustive verification mode using the same P4 generator, fixing the parser to 8 stages and varying the number of tables from 2 to 12. To force a worst-case scenario, all tables were generated to be independent, making every path combination reachable. Although uncommon in real P4 programs, where tables often enforce mutual exclusivity, it stresses the upper bound of state growth. Independent `run_table` calls were also executed in parallel to utilize all 12 processor threads.

Left and middle plots in Figure 11 show that without optimizations, the table pipeline and deparser exhibit exponential growth. Table growth stems from exhaustive path exploration and repeated invocations, whereas deparser growth results from processing compounded sets of accumulated states. In contrast, with optimizations enabled (Section IV-E1), deparser and parser growth are controlled, allowing exhaustive verification of up to 14 table configurations (Figure 11, right). These results are directly enabled by our snapshot-based modular architecture, which facilitates normalizing, comparing, and reusing intermediate states to prevent performance collapse.

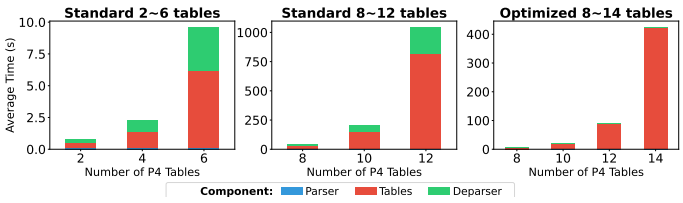

Fig. 11. Full program verification scaling: pre vs. post-optimization.

Bug Detection and Interaction. The current version of P4SymTest provides an assisted verification workflow, allowing users to inspect execution paths and symbolic states through an interactive interface. Since our framework prioritizes guided exploration over a fully automated black-box approach, a direct quantitative comparison against existing tools would result in a biased assessment of its distinct capabilities. While this interactive mode is the current focus, automated detection is under development, including AI-driven analysis to generate warnings about faulty behavior.

Structural Limitations. Despite its benefits, P4SymTest has structural constraints. As a static analysis framework, it does not execute code on real hardware, which prevents the detection of runtime bugs or target-specific inconsistencies that emerge during physical deployment. Furthermore, the tool is currently limited to single-device verification and does not support multi-router topologies or end-to-end network properties, reinforcing P4SymTest as a specialized tool for validating the internal logic of individual P4 components.

VI. Conclusion and Future Work

In this paper, we introduce P4SymTest, a framework that brings modular symbolic execution to P4 program verification. Our approach, rooted in software engineering principles, treats P4 components such as parsers and tables as independently testable units, enabling early bug detection and an iterative development workflow. Our prototype evaluation demonstrates both the feasibility and scalability of this modular approach.

Benchmark results confirm that modular symbolic execution achieves scalable performance. Parser analysis showed manageable execution times despite the inherent polynomial path explosion caused by branching logic. For tables, performance correlated mainly with the number of input symbolic states and remained largely insensitive to the number of actions. This efficiency stems from encoding action effects directly into the symbolic state through nested conditionals and applying reachability pruning to reduce the explored state space. While full program verification requires exploring all paths, our snapshot-based architecture supports state merging and caching strategies, making exhaustive verification feasible when needed. Overall, P4SymTest provides an efficient and scalable method for P4 verification, mitigating path explosion while preserving rapid feedback during development.

For future work, we plan to expand P4SymTest capabilities by introducing an automatic warning system based on snapshot analysis and AI-assisted reasoning. Additionally, we aim to enhance usability by refining the orchestration of complex pipeline analyses and integrating P4SymTest directly with code editors, making modular verification a seamless part of the daily P4 development cycle.

Acknowledgments

This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, FAPERGS (grant #25/2551-0002825-6), CNPq (grants #307826/2025-2, #405940/2022-0 and #407304/2025-8) and FAPESP (grants #2020/05152-7, #2023/00673-7 and #2023/00764-2).

REFERENCES

[1] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for sdn," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 99–110, Aug. 2013.

[2] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The Programmable Data Plane: Abstractions, Architectures, Algorithms, and Applications," *ACM Computing Surveys*, vol. 54, no. 4, May 2021.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014.

[4] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021.

[5] Beck, *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[6] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 518–532.

[7] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in p4 programs with assertion-based verification," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: Association for Computing Machinery, 2018.

[8] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "p4pktgen: Automated test case generation for p4 programs," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: Association for Computing Machinery, 2018.

[9] A. Shukla, K. N. Hudemann, A. Hecker, and S. Schmid, "Runtime verification of p4 switches with reinforcement learning," in *Proceedings of the 2019 Workshop on Network Meets AI & ML*, ser. NetAI'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–7.

[10] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, p. 82–90, Feb. 2013.

[11] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Comput. Surv.*, vol. 54, no. 4, May 2021.

[12] R. Parizotto, L. Castanheira, F. Bonetti, A. Santos, and A. Schaeffer-Filho, "Prime: Programming in-network modular extensions," in *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, 2020, pp. 1–9.

[13] ——, "Consistent composition and modular data plane programming," *IEEE Communications Magazine*, vol. 59, no. 6, pp. 60–65, 2021.

[14] R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. USA: McGraw-Hill, Inc., 2009.

[15] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. USA: Prentice Hall PTR, 2006.

[16] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 3rd ed. Wiley Publishing, 2011.

[17] K. Beck, *JUnit Pocket Guide*. O'Reilly Media, 2004.

[18] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[19] E. Clarke, O. Grumberg, D. Peled, and D. Peled, *Model Checking*, ser. The Cyber-Physical Systems Series. MIT Press, 1999.

[20] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[21] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, p. 57–72, Oct. 2001.

[22] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, Jul. 1976.

[23] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[24] T. Chen, X. song Zhang, S. ze Guo, H. yuan Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1758–1773, 2013, including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond.

[25] P. Godefroid, "Compositional dynamic test generation," *SIGPLAN Not.*, vol. 42, no. 1, p. 47–54, Jan. 2007.

[26] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 367–381.

[27] Y. Lin, T. Miller, and H. Søndergaard, "Compositional symbolic execution using fine-grained summaries," in *2015 24th Australasian Software Engineering Conference*, 2015, pp. 213–222.

[28] J. F. Santos, P. Maksimović, S. Élie Ayoun, and P. Gardner, "Gillian: Compositional symbolic execution for all," 2020.

[29] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: a powerful, sound, predictable, fast verifier for c and java," in *Proceedings of the Third International Conference on NASA Formal Methods*, ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 41–55.

[30] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer, "Macke: Compositional analysis of low-level vulnerabilities with symbolic execution," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 780–785.

[31] P. Panchekha, M. D. Ernst, Z. Tatlock, and S. Kamil, "Modular verification of web page layout," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019.

[32] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 314–327.

[33] M. Neves, L. Freire, A. Schaeffer-Filho, and M. Barcellos, "Verification of p4 programs in feasible time using assertions," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 73–85. [Online]. Available: https://doi.org/10.1145/3281411.3281421

[34] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "p4v: practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 490–503.

[35] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu, "bf4: towards bug-free p4 programs," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 571–585.

[36] Consortium, "p4c: The p4 compiler," https://github.com/p4lang/p4c, 2025, accessed: 2025-05-13.

[37] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[38] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[39] P4 Language Consortium, "$P4_{16}$ language specification v1.2.5," P4.org, Tech. Rep., October 2024, accessed: 2025-10-26.

[40] Consortium, "behavioral-model (bmv2)," https://github.com/p4lang/behavioral-model, 2025, accessed: 2025-05-13.