Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.075747

ARTICLE

NetVerifier: Scalable Verification for Programmable Networks

Ying Yao[1], **Le Tian**[1,2,3], **Yuxiang Hu**[1,2,3,*] **and Pengshuai Cui**[1,2,3]

[1]Information Engineering University, Zhengzhou, China
[2]National Key Laboratory of Advanced Communication Networks, Zhengzhou, China
[3]Key Laboratory of Cyberspace Security, Ministry of Education, Zhengzhou, China

*Corresponding Author: Yuxiang Hu. Email: huyuxiangchn@163.com

Received: 07 November 2025; Accepted: 15 January 2026; Published: 12 March 2026

ABSTRACT: In the process of programmable networks simplifying network management and increasing network flexibility through custom packet behavior, security incidents caused by human logic errors are seriously threatening their safe operation, robust verification methods are required to ensure their correctness. As one of the formal methods, symbolic execution offers a viable approach for verifying programmable networks by systematically exploring all possible paths within a program. However, its application in this field encounters scalability issues due to path explosion and complex constraint-solving. Therefore, in this paper, we propose NetVerifier, a scalable verification system for programmable networks. To mitigate the path explosion issue, we develop multiple pruning strategies that strategically eliminate irrelevant execution paths while preserving verification integrity by precisely identifying the execution paths related to the verification purpose. To address the complex constraint-solving problem, we introduce an execution results reuse solution to avoid redundant computation of the same constraints. To apply these solutions intelligently, a matching algorithm is implemented to automatically select appropriate solutions based on the characteristics of the verification requirement. Moreover, Language Aided Verification (LAV), an assertion language, is designed to express verification intentions in a concise form. Experimental results on diverse open-source programs of varying scales demonstrate NetVerifier's improvement in scalability and effectiveness in identifying potential network errors. In the best scenario, compared with ASSERT-P4, NetVerifier reduced the execution path, verification time, and memory occupation of the verification process by 99.92%, 94.76%, and 65.19%, respectively.

KEYWORDS: Programmable network; network verification; symbolic execution; scalability

1 Introduction

Programmable networks empower the developer to customize packet-processing logic using domain-specific languages like P4 [1], to meet diverse application scenarios and design goals. This programmability has spurred the innovation of network services, such as in-band network telemetry [2], in-network caching [3], and in-network aggregation [4]. However, this comes at a cost: the developer must accurately specify the complex, interdependent behavior of the data plane, the control plane, and their interactions, separately. Errors can thus originate from two sources: bugs in the data plane program logic itself, or incorrect configurations (e.g., table entries) supplied by the control plane. Verifying the combined effect, i.e., whether the data plane program, when instantiated with a given set of control plane configurations, behaves as intended, is essential for ensuring the correctness of programmable networks, creating a critical need for robust checking mechanisms.

Testing and verification are two complementary approaches that can help solve this problem. Operating after the network deployment, testing aims to generate input-output test packets for a given network implementation, and then execute the implementation with each input packet to check if the actual output packet matches the desired one [5–7,8–10]. Testing is an important method to guarantee the security of programmable networks, through functional faults and performance errors identification. However, its effectiveness is often constrained by the completeness of the generated test cases. In practice, testing is often conducted in isolated testbeds (e.g. Mininet [11]) using synthetic traffic generated by tools like Scapy [12] or via traffic replay, which mitigates the dependency on production traffic and avoids potential losses due to faulty programs or configurations deployed in the production environment. While, this design principle shifts the challenge to how to systematically generate a minimal yet sufficient set of test packets that can expose all logical errors, especially subtle corner-case bugs. Automated testing frameworks employ techniques like fuzzing or heuristic search to tackle this problem, but achieving high path or state coverage remains computationally expensive and difficult to exhaust [13,14]. In contrast, verification is a formal, pre-deployment process that mathematically proves the absence of errors in network programs and configurations. Independent of real traffic, rigorous mathematical proofs, and pre-deployment execution, these characteristics ensure the programs and configurations deployed in the production environment are bug-free, making verification an important method to ensure the correctness of programmable networks.

In this paper, we focus on **verification** before the network deployment. As an active area of research, a large number of research results have been proposed so far. However, general network verification tools [15,16] operate on abstract models that lack native support for the rich semantics of programmable data planes (e.g., parsers, match-action pipelines), making them inefficient or unsuitable for programmable network verification. As Network programs and configurations used to implement network behavior become increasingly complex, programmable-network-specific verifiers [17–20] encounter the scalability issue. Furthermore, to the best of our knowledge, even though some studies consider the scalability issue, they only introduce a single solution, which is inadequate to solve the scalability challenge.

Considering these issues, leveraging the success of formal methods in software engineering [21], this paper proposes NetVerifier, a scalable verification system for programmable networks using the symbolic execution technique, one of the formal methods. To mitigate the scalability issues of symbolic execution when applied to large-scale programmable networks due to path explosion and computational resource consumption for the constraint-solving process [22–24], we present a series of solutions to reduce the execution space and decrease the memory occupation. Firstly, we observe that the execution paths of different packet types within the network are distinct. Thus, executing all paths when verifying specified properties is unnecessary. Based on this observation, we present multiple pruning strategies to eliminate irrelevant execution paths to address the path explosion problem. Secondly, we notice there may be dependencies between network verification requirements. So, an execution results reuse strategy is introduced to handle the computational resource consumption issue for the constraint-solving process. Specifically, the main contributions of NetVerifier include:

1. We propose a set of solutions to deal with the scalability issue, including multiple pruning strategies to address the path explosion problem and an execution results reuse strategy to handle the computational resource consumption issue of the constraint-solving process, thereby accelerating the verification process.
2. An intelligent matching algorithm is implemented to automatically select the appropriate solution with insufficient scalability based on the characteristics of the verification requirements, eliminating the learning curve for the developer when selecting solutions.

3. We designed an assertion language, LAV, allowing the developer to express diverse verification intentions in a concise form and lowering the learning cost.
4. We implemented the prototype of NetVerifier and conducted extensive experiments on open-source programs of varying scales. The experimental results demonstrate that NetVerifier's improvement in scalability and effectiveness in discovering potential network faults.

The rest of this paper is organized as follows. In Section 2, we summarize the related work and outline the causes of the scalability issue. Section 3 first provides an overview of NetVerifier, and then elaborates on the design of its key components. In Section 4, we demonstrate the rationality of all proposed solutions for the scalability issue. Section 5 details our implementation and presents the evaluation result. In Section 6, we clarify the scope the limitations of the research in this paper. Finally, the conclusion and directions for future work are presented in Section 7.

2 Motivation

Programmable networks enable network customization and automation. However, their high flexibility also leads to frequent logical errors. The complexity of multi-dimensional programming—describing interactions across the data plane, control plane, and inter-plane communication—makes the developer prone to defining erroneous programs or configurations due to logical errors. This leads to actual network execution deviating from design logic, directly compromising the secure and reliable operation of programmable networks. Robust verification mechanisms are required to ensure their correctness.

As an active research area, a large amount of research has been conducted, which can be divided into two categories.

General network verification tools (e.g., Minesweeper [15], Header Space Analysis [16]) excel at checking network-wide properties like reachability but operate on abstract models that lack native support for the rich semantics of programmable data planes (e.g., parsers, match-action pipelines), making them inefficient or unsuitable for programmable network verification.

Consequently, programmable-network-specific verifiers have emerged. These verifier can be further categorized based on their core technique: tools like P4V [17] and Aquila [18] employ deductive methods (e.g., weakest precondition) or abstract interpretation, while Vera [19] and ASSERT-P4 [20] leverage symbolic execution. Compared with general network verification research, these tools have made significant strides in semantic accuracy and bug-finding. However, as user demands for network services continue to grow, network scale has expanded dramatically, and the network programs and configurations used to implement network behavior have become increasingly complex. These programmable-network-specific verifiers face challenges in handling large-scale networks, where verification becomes time-consuming or even impossible to complete (i.e., poor scalability). Deductive verifiers (P4V, Aquila) encounter bottlenecks in their constraint solvers as path conditions become increasingly complex. Tools based on symbolic execution also have scalability limitations: Vera introduced a new tree-based forest data structure to address the path explosion problem caused by the large number of matching action branches. However, P4 programs consist of multiple structural components, Vera only addresses the scalability issues arising from individual match-action logic, lacking the flexibility to adapt to diverse verification scenarios (e.g., verification of specific packet types, which corresponds to the parser portion in the P4 program). ASSERT-P4 uses the symbolic execution engine KLEE to verify the C model that is equivalent to the network program and configuration, but it does not introduce network-specific optimizations to address scalability problems.

The common shortcoming is that existing verifiers treat scalability either as a secondary concern or address it with isolated, non-adaptive solutions. There is a lack of a system that is architected from the ground

up with a suite of strategies that can be intelligently selected and combined to combat the distinct causes of scalability issues based on the specific verification task.

These issues prompted us to develop a scalable programmable network verification system. Symbolic execution has emerged as a critical pre-deployment network verification method due to its comprehensive path exploration capabilities and rigorous constraint-solving mechanisms. However, as user requirements continue to expand and the network programs grow increasingly complex, symbolic execution faces the scalability issue. Specifically, this stems from: (1) Path explosion in the Control Flow Graph (CFG) of the P4 program, symbolic execution requires systematically exploring all possible paths within the P4 program's CFG and recording constraints for each path. An execution path corresponds to a unique sequence of conditional branches and table action selections taken during the processing of a symbolic packet. Large-scale programs with numerous conditional statements and tables cause the number of such paths to grow exponentially, making exhaustive traversal infeasible. (2) After collecting the path constraints along each execution path, symbolic execution needs to solve them using its constraint solvers. The size and complexity of these constraints from large-scale programs place immense demands on the solver's capabilities.

Therefore, developing a scalable programmable network verification system is essential.

3 System Design

NetVerifier overview. NetVerifier focuses on verifying the data plane program under a static snapshot of control plane configurations (i.e., a set of table entries). This captures a critical scenario where misconfigurations are deployed, and establishes the foundation for reasoning about the data plane behavior. Fig. 1 presents an overview of NetVerifier's workflow. Firstly, the developer expresses high-level verification intents using LAV and injects them into the original network program (Section 3.1), where LAV is a declarative specification language designed to describe properties of programmable networks in a concise form. Then, the intelligent matching algorithm is employed to determine the appropriate solution for the scalability issue (all solutions are introduced in Section 3.2) based on the characteristics of assertions expressing the verification requirements (Section 3.3). Subsequently, a translator is utilized to convert the network program and table entries into an equivalent C model, which is verified by KLEE [25], a symbolic execution engine, to detect potential network errors (Section 3.4). Finally, the verification results are automatically organized into an error report and fed back to the developer for error localization and reparation.

3.1 Assertion Language: LAV

Building an effective verification system for programmable networks requires the precise articulation of expected network properties to provide the reference for the verification process [26]. Therefore, a Domain-Specific Language (DSL) is required to express network properties explicitly. Inspired by the demands of network verification and the evolutionary trends of other programming languages, we suggest that the core design principles of the specification language should have sufficient expressive power and a high degree of abstraction.

Sufficiently expressive. Primarily but importantly, the specification language must be capable of precisely describing the execution behavior of data packets, to provide the reference baseline for property verification.

Highly abstract. The evolution of programming languages increasingly emphasizes high-level abstractions. To facilitate the adoption of the verification system by other developers, the specification language should follow the abstraction-driven design principle to minimize the learning effort.

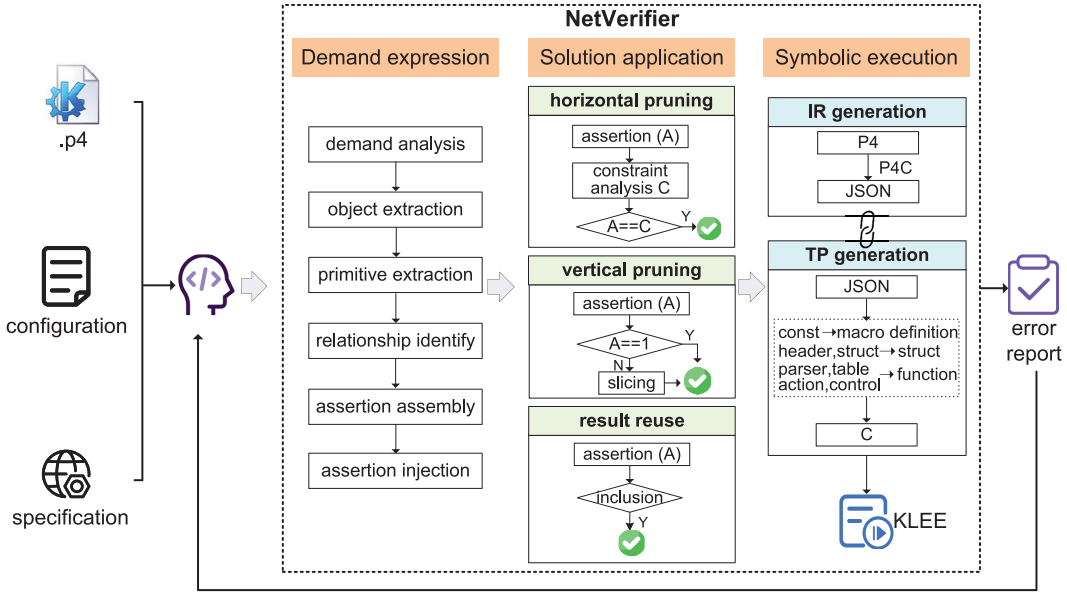

Figure 1: Workflow of NetVerifier.

Following these design principles and drawing inspiration from [27], LAV is designed for NetVerifier. As a declarative specification language, it enables the developer to express network properties precisely in a concise form, with detailed syntax and corresponding verification purposes displayed in Table 1.

Table 1: Design of LAV.

Syntax	Verification Purpose
Not()	Inverse
ForAll()/Hold	For All Packets with a Certain Characteristic, Verify Whether They All Have a Certain Behavior
If(con,action)	Verify Whether Packets Satisfying Conditions Behave Correctly
isValid	Verify Whether Headers of Packets are Valid
And/Or	Combine/Either, Supports Combinable Properties
EqualWith	Equal to, Expresses Package Properties
NotEqualWith	Not Equal to, Expresses Package Properties
Less/More	Less Than/Greater Than, Expresses a Package Property
Invariant()	Constant, Checks Whether There are no Invalid Writes in P4
Extract()/Emit()	Extract Packet Information/Assemble Packets
Forward/Drop	Forward/Drop Packets
MarkAsDrop	Marks the Packet as Dropped

As shown in Table 1, LAV specifies the network properties in the form of assertions using multiple elements. In detail, *isValid* is utilized to determine whether the header of the packet is valid, *If(con, action)* checks whether packets that satisfy the specific condition (*con*) behave correctly, and *Forward* is defined to check whether the packet executes the forward action. Operators like *Not()*, *And/Or*, *EqualWith/NotEqualWith*, etc., are similar to their Python counterparts.

To facilitate understanding of LAV, we illustrate the application of LAV using a specific verification requirement within a concrete network scenario, as displayed in Fig. 2. In this example, the network program to be deployed handles the basic forwarding operations, and the deployment environment comprises three switches and two Local Area Networks (LANs). The network communication rules are set as: (1) intra-LAN communication is permitted (LAN1: H21<->H22, LAN2: H31<->H32), (2) communication between H22 and H31 is allowed for inter-LAN communication, and (3) IPv4 packets with the Time To Live (TTL) of 0 are not forwarded.

$$\textbf{A1:}@assert(If(hdr.ipv4.srcAddr\ EqualWith\ H22, hdr.ipv4.dstAddr\ NotEqualWith\ H32)) \quad (1)$$

$$\textbf{A2:}@assert(If(hdr.ipv4.ttl\ EqualWith\ 0, Not\ Forward)) \quad (2)$$

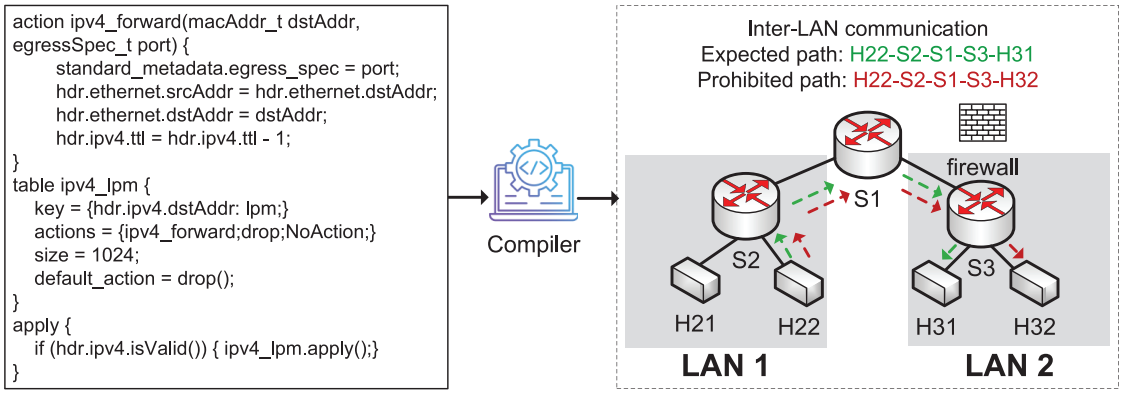

Figure 2: The network scenario to explain the application of LAV.

Assertion 1 first obtains all IPv4 packets with the specific source address "H22", then determines whether the destination addresses of these packets are unauthorized (communication between H22 and H21 is allowed for intra-LAN communication, and communication between H22 and H31 is permitted for inter-LAN communication). Similarly, assertion 2 first obtains all IPv4 packets with the TTL of 0, then judges whether these packets have not been forwarded.

Using LAV, the developer can flexibly express the network property to be verified concisely, establishing the reference for the verification process.

3.2 Solution for Scalability Issue

The scalability issue of symbolic execution in programmable network verification stems from path explosion and complex constraint solving. To solve these challenges, we put forward a series of solutions. Firstly, to mitigate the path explosion problem, we propose multiple pruning strategies, including horizontal pruning and vertical pruning. These strategies ensure the verification reliability while focusing the analysis on specific packets and paths, thereby enhancing the scalability. Subsequently, we analyze the absolute inclusion relationship among the verification requirements expressed by the developer, and introduce the execution result reuse strategy to reduce the complexity of the constraint-solving process.

(1) HP: horizontal pruning

During the verification process, the developer may be concerned with the verification of properties pertaining only to certain types of packets or control flows. Simultaneously, the network program naturally forms a Directed Acyclic Graph (DAG) structure, and different types of packets follow distinct execution

paths within the network. These observations inspire us to accurately pinpoint the verification work and avoid executing irrelevant paths. Therefore, we propose the horizontal pruning strategy, which adds packet and control flow constraints to the verification process, effectively narrowing down the execution scope of symbolic execution to a particular region, thereby improving the scalability.

Fig. 3a displays the process of this strategy. The network program handles two types of packets (IPv4 and IPv6), while the current verification only focuses on the behavior of IPv4 packets. After applying the horizontal pruning strategy, all execution paths related to IPv6 packets are eliminated, reducing the execution scale of symbolic execution.

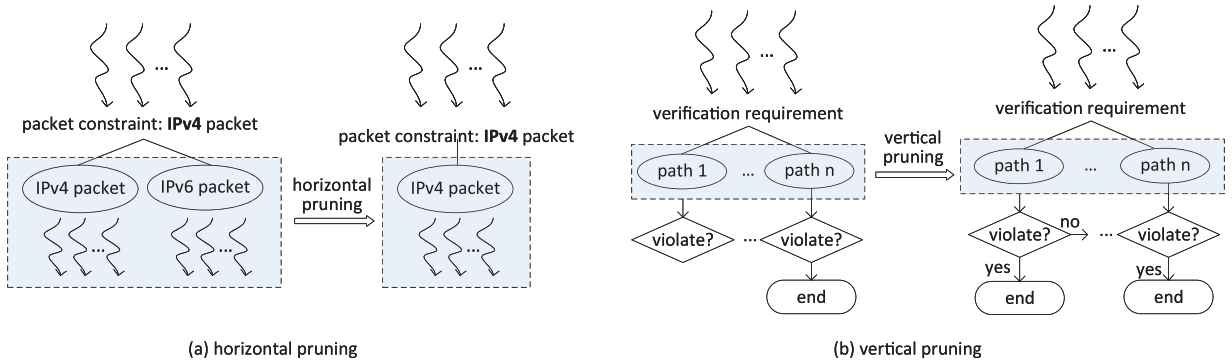

Figure 3: The process of horizontal pruning of vertical pruning.

Notably, whether the horizontal pruning can be applied directly depends on the constraints of the verification requirements on packet types. Therefore, if all verification requirements have a unique constraint on packets, then the horizontal pruning can be applied directly. Otherwise, NetVerifier classifies verification requirements based on their constraints on packets, yielding multiple subroutines, within each subroutine, all verification demands impose a unique constraint on packets. Subsequently, NetVerifier applies the horizontal pruning to each subroutine and leverages the parallel execution technique for all subroutines to accelerate the verification process.

(2) VP: vertical pruning

Network verification examines the validity of the property across the entire network. Similarly, in logical predicates, the property defined by the universal quantifier ($\forall$) holds only when all variables satisfy the condition. These insights inspire us to incorporate the concept of $\forall$ into network verification, halting the verification process once a situation violating the property is detected. Therefore, we propose the vertical pruning strategy, and design the "ForAll/Hold" primitive in LAV to implement this strategy. This strategy avoids executing irrelevant paths that do not affect the verification results, thereby enhancing verification scalability.

Fig. 3b illustrates the vertical pruning process. With this strategy, symbolic execution no longer needs to traverse all paths to determine whether the property holds. Instead, in the process of symbolic execution, once a path is found to violate the property, the property can be immediately determined to be violated, and the symbolic execution process is terminated.

The design logic of the vertical pruning strategy dictates that its direct application is only feasible when the verification requirement is unique, as terminating the work of the symbolic execution engine upon finding a violation may interfere with the verification results of other requirements. To compensate for this shortcoming, we introduce program slicing and parallel execution techniques. Firstly, NetVerifier employs program slicing to partition the network program that expresses all verification requirements into

multiple subprograms, with each containing one requirement. Subsequently, NetVerifier applies the vertical pruning strategy to each subprogram individually and accelerates the verification process through parallel execution techniques.

(3) ERR: execution result reuse

The complex constraint-solving is another factor contributing to the scalability issue of using symbolic execution in network verification. Obviously, repeatedly calculating the same constraint problems will further exacerbate this issue. During network verification, there may be absolute inclusion relationships among the verification intents. These observation inspires us to reuse the execution results of the same verification requirements. Thereby, we propose the execution results reuse strategy. When the verification intent expressed by the developer has an absolute inclusion relationship and there is a dependency between the execution results of the assertion, NetVerifier reuses the calculation results of the included assertions to avoid the waste of resources caused by multiple calculations for the same constraint problem, improving the verification scalability.

For instance, consider two assertions, where assertion 1 is *"if a, then c"* and assertion 2 is *"if a or b, then c"*. The execution result of assertion 2 depends on assertion 1: if assertion 1 is false, then assertion 2 must also be false. Otherwise, assertion 2 can be simplified as *"if b, then c"*.

To sum up, multiple pruning strategies designed to address the path explosion problem, along with the execution result reuse strategy developed to mitigate the computational resource consumption issue, can reduce the number of paths traversed and lower the complexity of constraint-solving during symbolic execution, thereby enhancing the scalability of programmable network verification.

3.3 Strategies Application

To assist the developer in utilizing suitable solutions for the scalability issue, NetVerifier implements an intelligent matching algorithm to automatically select appropriate strategies based on the characteristics of the verification requirements, as illustrated in Algorithm 1.

Algorithm 1: Intelligent Solution Matching

Input: assertions-annotated network program, *P*
Output: suitable solution, *suitableStrategies*

```
assertionList ← all assertions in P
constraintSet ← ( )
suitableStrategies ← [ ]
function HP(assertionList)
    for constraint in assertionList do
        constraintSet.add(constraint)
    if len(constraintSet) then == 1
        suitableStrategies.append("horizontal pruning")
    if len(constraintSet) then == len(assertionList)
        suitableStrategies.append("get subroutines – horizontal pruning – parallel execution")
function VP(assertionList)
    if len(assertionList) == 1 then
        suitableStrategies.append("vertical pruning")
    else
```

(Continued)

Algorithm 1 (continued)

```
15:         suitableStrategies.append("get subroutines – vertical pruning – parallel execution")
16:   function ERR(assertionList)
17:       if exist absolute containment in assertionList then
18:         suitableStrategies.append("execution result reuse")
19:   HP(assertionList), VP(assertionList), ERR(assertionList)
```

HP: horizontal pruning (function HP). NetVerifier first extracts all constraints imposed by assertions on packets, and documents them into *constraintList* (line 5–6). If the constraints imposed by assertions on packets are unique, then the horizontal pruning can be used directly (line 7–8). Otherwise, if all verification requirements impose constraints on packet types, then the horizontal pruning remains applicable after processing (line 9–10). Specifically, NetVerifier first classifies assertions based on restricted packet types, then groups assertions within the same classification into a single network program to obtain multiple subprograms. Subsequently, the horizontal pruning is applied to each subprogram, with all subprograms executed in parallel.

VP: vertical pruning (function VP). Since the vertical pruning is designed to terminate upon detecting one path violating the property, to guarantee the verification reliability, the use of this solution needs to be determined based on the assertion number. Therefore, NetVerifier first calculates the number of assertions. If the number of assertions is single, then the vertical pruning solution can be employed directly (line 12–13). In contrast, program slicing and parallel execution techniques need to be employed (line 14–15). Initially, we use the program slicing technique to divide the assertions-annotated network program into multiple subroutines with the same number of assertions, whereby each subroutine contains only one assertion. Subsequently, the vertical pruning is applied to each subprogram, with all subprograms executed in parallel.

ERR: execution result reuse (function ERR). A thorough examination is undertaken to uncover the absolute inclusion relationships among assertions expressing the verification intents, to elucidate their dependencies among the outcomes of their execution results. If the absolute inclusion relationships are identified among assertions, then this solution can be applied.

The matching algorithm simultaneously analyzes the feasibility of three solutions, automatically selects the viable solution, eliminating the need for the developer to choose solutions, and accelerating the verification process.

3.4 Verification

After handling the scalability issue, the symbolic execution technique is leveraged to analyze the network program to implement the verification.

Developing a P4-specific symbolic execution engine presents multiple challenges. First, as a DSL for network programming, the programming model of P4 centers on packet processing, encompassing diverse network-specific operations such as protocol header parsing, flow table matching, etc. While traditional symbolic execution engines are primarily designed for general-purpose computational logic, they struggle to directly handle such domain-specific semantics, making it impractical to follow the design approaches of conventional existing tools when developing symbolic execution tools for the P4 language. Second, network programs need to be compiled through specific architectures into executable code for different target platforms (e.g., Application-Specific Integrated Circuit (ASIC), Field Programmable Gate Array (FPGA), software switches). Hardware resource constraints and instruction set differences across these targets may cause divergent program behavior [28], making it challenging to accurately capture execution details when

developing symbolic execution tools. Finally, the lack of standardized semantic definitions within the P4 ecosystem constrains the development of symbolic execution tools.

Inspired by [20], we developed a translator that converts the P4 program and table rules into an equivalent C model for verification using existing symbolic execution engines. The core workflow of the translator comprises Intermediate Representation (IR) generation and rule-based transformation. In detail, the IR generation utilizes P4C [29], the standard, open-source P4 compiler, to convert the assertions-annotated P4 program into an equivalent JSON representation, which fully preserves all core semantics in the P4 program, such as parser logic, rule matching logic, and action invocations, etc. This ensures the translation process builds upon a community-standard, semantically-accurate foundation. Rule-based transformation converts this IR and table rules defined by the network developer into an equivalent C program using the rule map defined in [20], ensuring behavioral consistency between the transformed C program and the original P4 program. This enables symbolic execution using C language-based engines to accurately reflect the behavior and characteristics of the original P4 program.

The equivalence between the generated C model and the original P4 program is crucial for ensuring the reliability of NetVerifier. Our translation methodology is directly adapted from the existing verification framework, ASSERT-P4, which has been peer-reviewed and demonstrated effective. ASSERT-P4 employs a rule-based, syntax-directed translation from the P4 language to the C language, meticulously mapping core P4 constructs (header, parsers, tables, actions, etc.) to behaviorally equivalent C code segments, ensuring the equivalence between the C model and the P4 program.

After obtaining the equivalent C program, NetVerifier analyzes it using KLEE, a symbolic execution engine for the C language. Upon completion of symbolic execution, the verification results are automatically organized into an error report and fed back to the developer for error localization and reparation.

4 Proof

To address the scalability issue, NetVerifier proposes multiple solutions. By eliminating the irrelevant path and reusing the execution result, these solutions enhance scalability without compromising verification outcomes. In this section, we rigorously analyze the mathematical foundations of these solutions to demonstrate their validity.

We define the original program path space as S, the property to be verified as $Prop$, the program path space obtained by the path pruning as S_p, and the pruned path space as S_r (where $S = S_p \cup S_r$ and $S_p \cap S_r = \varnothing$).

Theorem 1: The horizontal pruning does not affect the verification result. The proof of this theorem is equivalent to demonstrating the consistency between the execution results of verifying $S_p \vDash \neg Prop$ and $S \vDash \neg Prop$.

We prove this theorem by contradiction. Assume that the solution affects the correctness of the verification result. Then one of the following two cases must exist: (1) $S \vDash \neg Prop$ is false and $S_p \vDash \neg Prop$ is true, meaning the original path space did not cause a property violation, but the pruned path space did; (2) $S \vDash \neg Prop$ is true and $S_p \vDash \neg Prop$ is false, meaning the original path space caused a property violation, while the pruned path space did not.

(1) $S \vDash \neg Prop$ is false, $S_p \vDash \neg Prop$ is true. If $S \vDash \neg Prop$ is false, then for all $\pi \in S$, $\pi \vDash Prop$ holds. Since $S_p \subseteq S$, it follows that for all $\pi \in S_p$, $\pi \vDash Prop$ also holds. Thus, $Sp \vDash Prop$ implies $Sp \vDash \neg Prop$ is false, contradicting the assumption. Therefore, this case does not exist.

(2) $S \vDash \neg Prop$ is true, $S_p \vDash \neg Prop$ is false. If $S \vDash \neg Prop$ is true, then there exists $\pi \in S$ that makes $\pi \vDash \neg Prop$ true. Since $S = S_p \cup S_r$, this path π must belong to either S_p or S_r.

(i) $\pi \in S_p$

In this case, $S_p \vDash \neg Prop$ is directly proven to be true, contradicting the assumption.

(ii) $\pi \in S_r$

The horizontal pruning preserves all paths related to the verification property, so there must exist $\pi \in S_p$, contradicting the assumption.

By proof by contradiction, all counterexamples fail, theorem 1 holds.

Theorem 2: The vertical pruning does not change the verification result.

Vertical pruning halts symbolic execution whenever a path violates the property. Since it is designed to prune the program path space after verification results are determined, the correctness of verification remains unaffected.

Theorem 3: This solution does not change the verification outcome.

This solution is designed by analyzing dependency relationships among assertion execution results and leveraging the outcomes of dependent assertions to avoid the redundant constraint-solving process. For example, assertion 1 specifies that *"if a, then c"*, action c is executed if condition a is satisfied, and assertion 2 specifies that *"if a or b, then c"*. The violation of assertion 1 implies the existence of a program path where $a \rightarrow c$ is false, i.e., $a \wedge \neg c$ is true. Similarly, the violation of assertion 2 implies a path where $(a \wedge b) \rightarrow c$ is false, i.e., $(a \vee b) \wedge \neg c$ is true. Since $(a \vee b) \wedge \neg c$ being true is equivalent to $(a \wedge \neg c) \vee (b \wedge \neg c)$ being true, if assertion 1 is violated, assertion 2 must also be violated. However, if assertion 1 holds, assertion 2 cannot be directly confirmed as true. This is because the assertion *"if b, then c"* (i.e., $b \rightarrow c$) might be violated. Nevertheless, assertion 2 can be simplified to verifying whether $b \rightarrow c$ is violated. Therefore, this solution does not compromise the verification outcome.

5 Evaluation

In this section, we evaluate NetVerifier through a series of experiments. The translator and the matching algorithm are implemented in Python 3.8. The symbolic execution engine employs KLEE with Z3 [30] installed as its core solver. All experiments were conducted on Ubuntu 20.04 with 32 GB of memory. We use the experiments to determine whether NetVerifier can alleviate the scalability issue and identify network errors effectively.

5.1 Scalability

This subsection evaluates the effectiveness of the proposed solution for the scalability issue through three metrics: (i) the total number of execution paths, (ii) the verification time, and (iii) the memory occupation. Based on the application scenarios of these solutions and the actual verification requirements, we design assertions tailored to each solution for several open-source P4 programs of varying scales (from 100 LOCs to 500+ LOCs) [31–35]. Subsequently, we compare these strategies individually with ASSERT-P4 and observe the changes in these three metrics as the program scale increases.

Programmable network verification has been explored through multiple verification paradigms, including symbolic execution and deductive verification, leading to a diverse set of systems with distinct design goals and scalability characteristics. Among them, works based on symbolic execution include ASSERT-P4 and Vera. In this evaluation, we focus on comparing NetVerifier with ASSERT-P4, as they share the same underlying verification paradigm (P4-to-C and KLEE) and P4 language standard, allowing us to directly assess the scalability improvements brought by our optimization strategies. Meanwhile, the evaluation is designed to focus on a representative baseline in order to enable a controlled and interpretable

analysis of scalability improvements. To clarify the rationale behind this evaluation design and to better position NetVerifier within the broader verification landscape, we provide a qualitative comparison with representative prior approaches, as displayed in Table 2.

Table 2: Qualitative Comparison of Programmable Network Verification Systems.

Tool	Language	Core Technique	Scalability Focus	Methodological Alignment
P4V	$P4_{16}$	Deductive	Solver-Level	Low
Aquila	$P4_{16}$	Deductive	Domain Abstraction	Low
Vera	$P4_{14}$	Symbolic Execution	Table-Level Pruning	Medium
ASSERT-P4	$P4_{16}$	Symbolic Execution (KLEE)	Baseline	High
NetVerifier	$P4_{16}$	Symbolic Execution (KLEE)	Multiple Adaptive Pruning	–

From a methodological perspective, existing programmable network verification systems explore different points in the design space. P4V and Aquila adopt deductive verification techniques, such as weakest precondition reasoning and abstract interpretation, to verify network properties. These approaches are effective for certain classes of invariants but typically rely on solver efficiency and abstraction precision, which introduces scalability bottlenecks that differ from those encountered in symbolic execution. As a result, their performance characteristics are not primarily dominated by execution-path explosion, which is the main focus of this work.

Vera and ASSERT-P4 are more closely related to NetVerifier, as they leverage symbolic execution to verify network properties. Vera introduces a forest-based data structure to mitigate path explosion in match-action tables. However, its optimization is largely confined to table-level branching behavior and is tightly coupled to the semantics of the legacy $P4_{14}$ language. In contrast, NetVerifier is designed for $P4_{16}$ and addresses scalability in a more general and adaptive manner by incorporating assertion-driven, verification-purpose-aware pruning strategies that span multiple components of the program, including parsers, control logic, and inter-assertion dependencies. ASSERT-P4 provides the most methodologically consistent baseline for evaluating the scalability benefits of NetVerifier. Both systems adopt the same verification workflow, translating $P4_{16}$ programs into C models and employing KLEE for symbolic execution. This alignment allows the evaluation to more directly attribute observed improvements in execution paths, verification time, and memory consumption to the proposed pruning strategies and execution result reuse mechanisms, rather than to differences in verification paradigms or language standards. Therefore, ASSERT-P4 is selected as the quantitative baseline in this subsection.

(1) HP: horizontal pruning

As demonstrated in Fig. 4 and Table 3, compared to ASSERT-P4, the horizontal pruning effectively reduces the number of execution paths, verification time, and memory occupation for programs of all scales. On average, the reduction ratios are 23.91%, 12.91%, and 6.53%, respectively. Under optimal conditions, the reductions reach 50.00%, 49.41%, and 30.91%. This is because this strategy analyzes the constraints imposed by verification requirements on packet types, focusing path traversal and constraint collection on the scope relevant to verification, thereby eliminating the execution of numerous irrelevant paths and the computation of many irrelevant constraints.

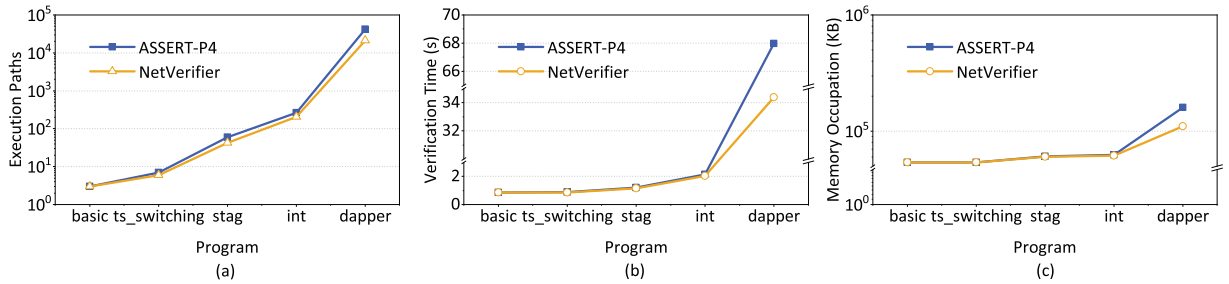

Figure 4: Performance analysis of the horizontal pruning.

Table 3: Performance Gains (Reduction Percentages in Each Metric) of Each Solution Relative to ASSERT-P4.

Program (LOC)	Execution Path			Verification Time			Memory Occupation		
	HP	VP	ARR	HP	VP	ARR	HP	VP	ARR
basic (100)	0.00%	25.00%	0.00%	1.16%	3.33%	0.58%	0.06%	0.76%	0.55%
ts_switching (132)	14.29%	0.00%	0.00%	3.37%	5.38%	4.32%	0.21%	6.03%	0.17%
stag (234)	30.00%	46.25%	30.83%	5.00%	6.67%	4.92%	0.3%	2.31%	0.07%
int (443)	22.93%	83.80%	47.37%	5.61%	9.90%	11.11%	1.16%	4.21%	0.95%
dapper (559)	50.00%	99.92%	49.99%	49.41%	94.76%	54.65%	30.91%	65.19%	32.34%

(2) VP: vertical pruning

As shown in Fig. 5 and Table 3, the vertical pruning demonstrates effective path truncation, verification acceleration, and memory reduction capabilities. For example, on dapper.p4, the vertical pruning reduced the number of execution paths, verification time, and memory consumption by 99.93%, 94.76%, and 65.19%, respectively. This is because the vertical pruning terminates the execution upon encountering a violation, thereby avoiding the execution of paths and the resolution of constraints irrelevant to the verification outcome.

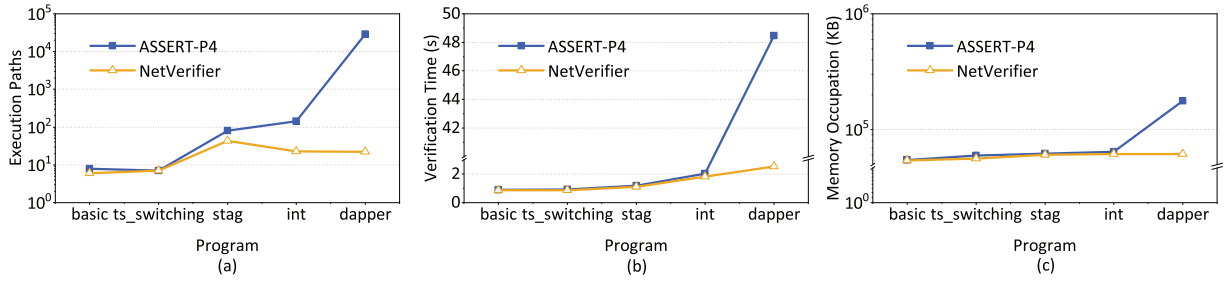

Figure 5: Performance analysis of the vertical pruning.

(3) ERR: execution result reuse

As displayed in Fig. 6 and Table 3, the execution result reuse effectively reduces the number of execution paths, verification time, and memory consumption. Under optimal conditions, reductions of 49.99%, 54.65%, and 32.34% are achieved in execution paths, verification time, and memory usage, respectively. This is because the execution result reuse solution leverages the inclusion relationships between assertions and the

dependencies of their execution results to reuse verification outcomes, thereby avoiding redundant constraint solving and irrelevant path traversal processes.

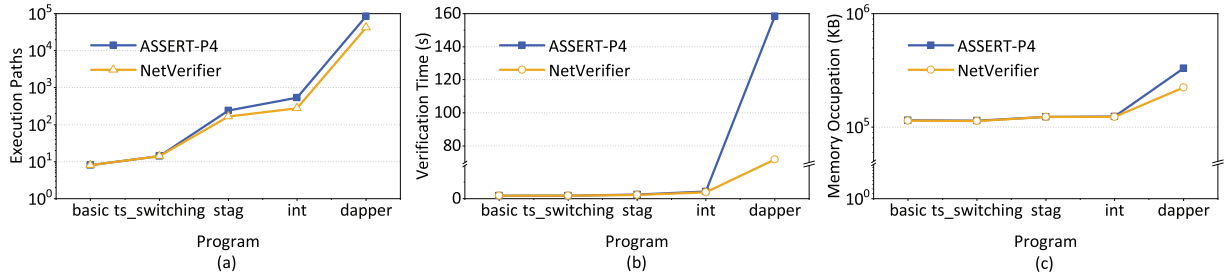

Figure 6: Performance analysis of the execution result reuse.

Meanwhile, we observed that in specific scenarios, multiple solutions can be employed to mitigate the scalability issue. Therefore, we design the experiment to determine whether the effect of solution stacking outperforms a single solution. Specifically, we created assertions for dapper.p4 to ensure that three strategies are available: horizontal pruning only, vertical pruning only, and both. As displayed in Fig. 7, all solutions are effective in reducing the execution path, verification time, and memory occupation, and the combined application of solutions yields the most significant effect. Compared with ASSERT-P4, NetVerifier using the horizontal pruning only (HP), using the vertical pruning (VP), and using both of them (HP&VP) reduced the execution path of symbolic execution by 50.00%, 99.96%, 99.98%, the verification time by 49.41%, 96.10%, 96.25%, and the memory occupation by 30.91%, 60.80%, 61.15%, respectively.

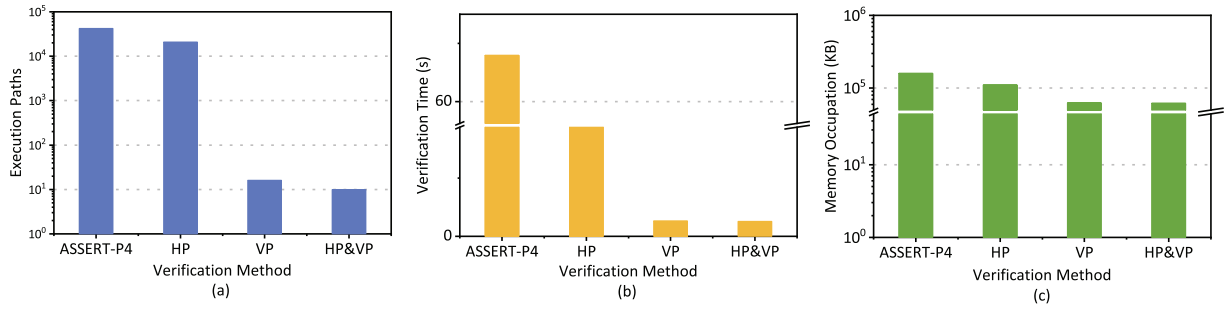

Figure 7: Performance analysis of multiple solutions.

To make a conclusion, compared with ASSERT-P4, NetVerifier possesses more effective scalability.

5.2 Network Verification

This section evaluates NetVerifier's capability of error detection through verification of multiple open-source P4 programs. The experimental design covers critical components such as data plane forwarding logic, policy execution engines, and distributed protocols, etc. The verification results demonstrate that NetVerifier not only effectively identifies known errors detectable by existing tools but also uncovers errors previously undetected by existing tools.

Forwarding logic fault (dapper.p4). During verification of IPv4 packet forwarding behavior, NetVerifier triggered an exception by injecting a TTL validity assertion at a critical packet processing node. The program continued forwarding behavior even when the input packet's TTL was set to zero. Analysis of the original program reveals that this error stems from failing to verify the effectiveness of the TTL field during packet

forwarding. This error may cause packet looping in programmable networks, potentially triggering or exacerbating network congestion.

Strategy execution vulnerability (sTag.p4). During verification of color-based port isolation policies, NetVerifier discovered that ports marked with different colors could still establish connections by constructing cross-port communication test cases. Symbolic execution trace analysis revealed that this error stemmed from a logical flaw in handling composite conditions, causing traffic that should have been discarded to be erroneously forwarded. This defect could easily enable abnormal traffic to bypass isolation, directly threatening the network security boundaries.

Distributed Protocol Verification (NetPaxos). As a network implementation of the Paxos consensus protocol, this application uses two distinct types of P4 programs (leader.p4 and acceptor.p4) to interact and iterate to achieve consensus. During verification of this network implementation, NetVerifier successfully detected two critical defects: message drop conflicts (leader.p4) and out-of-bounds register access (acceptor.p4). In leader.p4, KLEE discovered that messages were silently discarded during processing despite not being explicitly marked for drop. Code analysis revealed that this issue stemmed from another action incorrectly flagging packet discard behavior. In acceptor.p4, NetVerifier detected an out-of-bounds register access caused by the program failing to validate the index validity when reading register contents. ASSERT-P4, one of the tools used for symbolic execution verification of the network, did not detect this error.

Experimental results demonstrate that NetVerifier can effectively identify potential errors, ensuring the secure and reliable operation of programmable networks.

6 Discussion

This section clarifies NetVerifier's scope and limitations.

Verification effectiveness. NetVerifier requires the developer to describe the verification requirements using the specification language, LAV, to start the verification process. The verification effectiveness is related to the completeness and quality of user-defined assertions, incomplete (e.g., those that do not adequately express the verification requirements) or low-quality (e.g., those that are logically contradictory) assertions can affect the verification effectiveness. This problem emphasizes the importance of the quality of assertions. Our ongoing work on automatic high-quality assertion generation is a solution to this problem. In addition, the property description capability of LAV limits the verification effectiveness to some extent, network properties that cannot be described cannot be able to be verified. The current design of LAV focuses on the verification of static network properties and is hard for verifying the expected relationships of header fields, and therefore cannot verify changes in the value of a certain field. Developing a more descriptive language to express adequate network properties is a solution to this problem.

Testing and verification. Testing and verification are two different approaches that can help to ensure the security of programmable networks. In this paper, we focus on verification, and present NetVerifier. It is worth emphasizing that the verification should be considered complementary to testing, due to the need for performance errors identification. Thus, NetVerifier does not replace but complements existing testing work. These approaches are complementary, as the percentage of defects found by them together may be higher than the percentage found by one technique alone.

Boarder language support. Thanks to superior architectural model, expressiveness, and tool support, $P4_{16}$ has become a widely adopted standard in both academia and industry. Therefore, NetVerifier is built on the sustainable $P4_{16}$ ecosystem. Nevertheless, we noticed that some engineering implementation is still implemented based on $P4_{14}$, extending NetVerifier to support not only $P4_{16}$ but also $P4_{14}$ is essential to improve its practical usability, and we consider this as our future work.

Sufficient network verification. NetVerifier currently verifies the data plane program coupled with a static set of table entries, which models the control plane's configuration effect but not its dynamic decision logic. This is a well-established scope for static verification tools, as it addresses the vital problem of finding bugs in the data plane logic and detecting harmful configuration snapshots before deployment. However, verifying the dynamic control plane and its stateful interactions with the data plane is a challenge.

7 Conclusion

This paper presents NetVerifier, a scalable verification system for programmable networks. Experimental results on open-source programs of varying scales demonstrate NetVerifier's improvement in scalability and capability in error detection. In the future, we plan to work on these fields.

Extending NetVerifier's specification language. As discussed above, the ability of LAV to express property directly affects the verification effectiveness. Currently, LAV focuses on verifying static network properties rather than on the expected relationships of the header fields, and next, we plan to extend LAV to support verification of such properties.

Automatically bug localization. Verification can identify violated properties, but these violations may stem from a series of potential bugs, making rapid fixes challenging. Therefore, we plan to propose a more intelligent algorithm to locate the root causes of the violations accurately, thus reducing the time required to fix errors in large-scale programs.

Dynamic requirement update mechanism during verification. The designed workflow of NetVerifier introduces a potential flaw: any modification to the verification requirement forces the developer to restart the entire workflow, resulting in significant repetitive work. Inspired by P4runpro [36], we plan to implement a runtime demand update interface for NetVerifier, eliminating workflow reconstruction burden.

Extending NetVerifier to $P4_{14}$. Considering the fact that $P4_{16}$ has become a widely adopted standard in both academia and industry due to its excellent architecture model, expressiveness, and tool support, NetVerifier is designed for $P4_{16}$. However, in actual engineering implementations, some work is still done on $P4_{14}$. Therefore, next, we will extend NetVerifier to support not only $P4_{16}$ but also $P4_{14}$, and verify the engineering implementation based on $P4_{14}$, enhancing its practicality.

Boarder evaluation baseline. As NetVerifier and ASSERT-P4 share the same underlying verification paradigm and P4 language standard, we have currently only implemented a comparison between them in terms of scalability. In the future, we plan to develop a test suite to enable comparisons with multiple verification tools.

Extension toward control plane interaction. Modeling and verifying the dynamic control plane and its stateful interactions with the data plane is a challenge. It involves reasoning about controller programs, asynchronous events, and the evolution of network state over time. Our work provides an essential component for such future full-stack verifiers: a scalable and precise engine for analyzing the data plane side of any configuration state. A promising direction is to integrate NetVerifier with a model of the control plane, allowing us to explore not just "what if this set of table entries is wrong", but verify the control plane, plus the interaction between the control plane and the data plane.

Acknowledgement: Not applicable.

Funding Statement: This work was supported by the National Key Research and Development Program of China under Grant 2023YFB2903902; and in part by the Science and Technology Innovation Leading Talents Subsidy Project of Central Plains under Grant 244200510038.

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Ying Yao and Le Tian; methodology, Ying Yao and Yuxiang Hu; formal analysis, Yuxiang Hu; investigation, Le Tian; writing—original draft preparation, Ying Yao; writing—review and editing, Ying Yao and Le Tian; supervision, Yuxiang Hu; funding acquisition, Yuxiang Hu, Le Tian and Pengshuai Cui. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The data that support this study are available from authors.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Bosshart P, Daly D, Gibb G, Izzard M, McKeown N, Rexford J, et al. P4: programming protocol-independent packet processors. SIGCOMM Comput Commun Rev. 2014;44(3):87–95. doi:10.1145/2656877.2656890.
2. Liang J, Bi J, Zhou Y, Zhang C. In-band network function telemetry. In: Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos, SIGCOMM '18. New York, NY, USA: Association for Computing Machinery; 2018. p. 42–4. doi:10.1145/3234200.3234236.
3. Jin X, Li X, Zhang H, Foster N, Lee J, Soulé R, et al. NetChain: scale-free sub-RTT coordination. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). Renton, WA, USA: USENIX Association; 2018. p. 35–49.
4. Lao C, Le Y, Mahajan K, Chen Y, Wu W, Akella A, et al. ATP: in-network aggregation for multi-tenant learning. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). Renton, WA, USA: USENIX Association; 2021. p. 741–61.
5. Ruffy F, Liu J, Kotikalapudi P, Havel V, Tavante H, Sherwood R, et al. P4Testgen: an extensible test oracle for P4. In: Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery; 2023. p. 136–51. doi:10.1145/3603269.3604834.
6. Shukla A, Hudemann KN, Vági Z, Hügerich L, Smaragdakis G, Hecker A, et al. Fix with P6: verifying programmable switches at runtime. In: 40th IEEE Conference on Computer Communications, INFOCOM 2021; 2021 May 10–13; Vancouver, BC, Canada. Piscataway, NJ, USA: IEEE; 2021. p. 1–10. doi:10.1109/INFOCOM42981.2021.9488772.
7. Zheng N, Liu M, Zhai E, Liu HH, Li Y, Yang K, et al. Meissa: scalable network testing for programmable data planes. In: Proceedings of the ACM SIGCOMM, 2022 Conference, SIGCOMM '22. New York, NY, USA: Association for Computing Machinery; 2022. p. 350–64. doi:10.1145/3544216.3544247.
8. Bressana P, Zilberman N, Soulé R. PTA: finding hard-to-find data plane bugs. IEEE/ACM Trans Netw. 2023;31(3):1324–37. doi:10.1109/TNET.2022.3214062.
9. Ruffy F, Wang T, Sivaraman A. Gauntlet: finding bugs in compilers for programmable packet processing. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). Renton, WA, USA: USENIX Association; 2020. p. 683–99.
10. Xu X, Beckett R, Jayaraman K, Mahajan R, Walker D. Test coverage metrics for the network. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21. New York, NY, USA: Association for Computing Machinery; 2021. p. 775–87. doi:10.1145/3452296.3472941.
11. Contributors MP. Mininet: an instant virtual network on your laptop or other PC. 2022 [cited 2025 Jan 12]. Available from: https://mininet.org/.
12. Community S. Scapy. 2008 [cited 2025 Jan 12]. Available from: https://scapy.net/.
13. Shukla A, Fathalli S, Zinner T, Hecker A, Schmid S. P4Consist: toward Consistent P4 SDNs. IEEE J Sel Areas Commun. 2020;38(7):1293–307. doi:10.1109/JSAC.2020.2999653.
14. Yaseen N, Yu L, Stanford C, Beckett R, Liu V. FP4: line-rate greybox fuzz testing for P4 switches. arXiv:2207.13147. 2022.

15. Beckett R, Gupta A, Mahajan R, Walker D. A general approach to network configuration verification. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17. New York, NY, USA: Association for Computing Machinery; 2017. p. 155–68. doi:10.1145/3098822.3098834.
16. Kazemian P, Varghese G, McKeown N. Header space analysis: static checking for networks. In: Gribble SD, Katabi D, editors. Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012; 2012 Apr 25–27; San Jose, CA, USA. Renton, WA, USA: USENIX Association; 2012. p. 113–26.
17. Liu J, Hallahan W, Schlesinger C, Sharif M, Lee J, Soulé R, et al. p4v: practical verification for programmable data planes. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18. New York, NY, USA: Association for Computing Machinery; 2018. p. 490–503. doi:10.1145/3230543.3230582.
18. Tian B, Gao J, Liu M, Zhai E, Chen Y, Zhou Y, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21. New York, NY, USA: Association for Computing Machinery; 2021. p. 17–32. doi:10.1145/3452296.3472937.
19. Stoenescu R, Dumitrescu D, Popovici M, Negreanu L, Raiciu C. Debugging P4 programs with vera. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18. New York, NY, USA: Association for Computing Machinery; 2018. p. 518–32. doi:10.1145/3230543.3230548.
20. Freire L, Neves M, Leal L, Levchenko K, Schaeffer-Filho A, Barcellos M. Uncovering bugs in P4 programs with assertion-based verification. In: Proceedings of the Symposium on SDN Research, SOSR '18. New York, NY, USA: Association for Computing Machinery; 2018. doi:10.1145/3185467.3185499.
21. ter Beek MH. Review on formal methods for software engineering: languages, methods, application domains. Form Asp Comput. 2025;37(4):35. doi:10.1145/3746236.
22. Kuznetsov V, Kinder J, Bucur S, Candea G. Efficient state merging in symbolic execution. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12. New York, NY, USA: Association for Computing Machinery; 2012. p. 193–204. doi:10.1145/2254064.2254088.
23. Trabish D, Mattavelli A, Rinetzky N, Cadar C. Chopped symbolic execution. In: Proceedings of the 40th International Conference on Software Engineering, ICSE '18. New York, NY, USA: Association for Computing Machinery; 2018. p. 350–60. doi:10.1145/3180155.3180251.
24. Cadar C, Sen K. Symbolic execution for software testing: three decades later. Commun ACM. 2013;56(2):82–90. doi:10.1145/2408776.2408795.
25. Cadar C, Dunbar D, Engler DR. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves R, van Renesse R, editors. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. Renton, WA, USA: USENIX Association; 2008. p. 209–24 doi: 10.1109/msp.2010.134.
26. Kumar KS, Ranjith K, Prashanth PS, Arashloo MT, Venkanna U, Tammana P. DBVal: validating P4 data plane runtime behavior. In: Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR), SOSR '21. New York, NY, USA: Association for Computing Machinery; 2021. p. 122–34. doi:10.1145/3482898.3483352.
27. Beckett R, Zou XK, Zhang S, Malik S, Rexford J, Walker D. An assertion language for debugging SDN applications. In: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN '14. New York, NY, USA: Association for Computing Machinery; 2014. p. 91–6. doi:10.1145/2620728.2620743.
28. Sultana N, Sonchack J, Giesen H, Pedisich I, Han Z, Shyamkumar N, et al. Flightplan: dataplane disaggregation and placement for P4 programs. In: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). Renton, WA, USA: USENIX Association; 2021. p. 571–92.
29. p4language. GitHub—p4lang/p4c: P4_16 reference compiler. 2021 [cited 2025 Feb 12]. Available from: https://github.com/p4lang/p4c?tab=readme-ov-file#sample-backends-in-p4c.
30. de Moura LM, Bjørner NS. Z3: an efficient SMT solver. In: Ramakrishnan CR, Rehof J, editors. Tools and algorithms for the construction and analysis of systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008; 2008 Mar 29–Apr 6; Budapest, Hungary. Cham, Switzerland: Springer; 2008. p. 337–40. doi:10.1007/978-3-540-78800-3.

31. p4language. Tutorials/exercises/basic at master. p4lang/tutorials—github.com. 2018 [cited 2025 May 11]. Available from: https://github.com/p4lang/tutorials/tree/master/exercises/basic.
32. Edwards TG, Ciarleglio N. Timestamp-aware RTP video switching using programmable data plane. In: Proceeding of the ACM SIGCOMM 2017 Conference (Industrial Demo); Aug 21–25, 2017; Los Angeles, CA, USA. Vol. 17. [cited 2025 Dec 10]. Available from: https://conferences2.sigcomm.org/sigcomm/2017/files/program-industrial-demos/sigcomm17industrialdemos-paper2.pdf.
33. Lopes NP, Bjørner NS, Godefroid P, Jayaraman K, Varghese G. Checking beliefs in dynamic networks. In: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15; 2015 May 4–6; Oakland, CA, USA. Renton, WA, USA: USENIX Association; 2015. p. 499–512.
34. Saha N. GitHub—niloysh/int-v1: implementation of In-band Network Telemetry (INT) version 1.0 on Bmv2 switches—github.com. 2023 [cited 2026 Jan 10]. Available from: https://github.com/niloysh/int-v1.
35. Ghasemi M, Benson T, Dapper J. Dapper: data plane performance diagnosis of TCP. In: Proceedings of the Symposium on SDN Research, SOSR '17. New York, NY, USA: Association for Computing Machinery; 2017. p. 61–74. doi:10.1145/3050220.3050228.
36. Yang Y, He L, Zhou J, Shi X, Cao J, Liu Y. P4runpro: enabling runtime programmability for RMT programmable switches. In: Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM '24. New York, NY, USA: Association for Computing Machinery; 2024. p. 921–37. doi:10.1145/3651890.3672230.