P4Inv: Inferring Packet Invariants for Verification of Stateful P4 Programs

Delong Zhang, Chong Ye, Fei He[†]
School of Software, BNRist, Tsinghua University, Beijing 100084, China
Key Laboratory for Information System Security, MoE
zhangdl22@mails.tsinghua.edu.cn, yc_thu@163.com, hefei@tsinghua.edu.cn

***Abstract*—P4 is widely adopted for programming data planes in software-defined networking. Formal verification of P4 programs is essential to ensure network reliability and security. However, existing P4 verifiers overlook the stateful nature of packet processing, rendering them inadequate for verifying complex stateful P4 programs.**

In this paper, we introduce a novel concept called *packet invariants* to address the stateful aspects of P4 programs. We present an automated verification tool specifically designed for stateful P4 programs. This algorithm efficiently discovers and validates packet invariants in a data-driven manner, offering a novel and effective verification approach for stateful P4 programs. To the best of our knowledge, this approach represents the first attempt to generate and leverage domain-specific invariants for P4 program verification. We implement our approach in a prototype tool called P4Inv. Experimental results demonstrate its effectiveness in verifying stateful P4 programs.

I. Introduction

Software-Defined Networking (SDN) [1] is an innovative networking approach that explicitly separates the control plane from the data plane, offering centralized control and remarkable flexibility. Its adoption spans various industrial applications, encompassing data centers [2], cloud computing [3], Internet of Things (IoT) [4], telecommunications [5], etc.

P4 [6] is a promising language for programming hardware data plane. It provides network administrators with the ability to specify protocol-independent packet processing, enabling the implementation of complex packet handling pipelines. However, the programmability and flexibility of P4 also introduce the possibility of errors and flaws in program design. To ensure the reliability and security of SDN deployments, formal verification becomes indispensable in guaranteeing the correctness of P4 programs.

In recent years, significant progress has been made in the formal verification of P4 programs [7]–[11]. The current state-of-the-art P4 verifiers, such as bf4 [8], p4v [10], Vera [11], and Aquila [9], have made notable contributions. However, these verifiers overlook the practical utilization of registers in network devices, treating packet processing as isolated events. Registers play a crucial role in network devices by providing consistent storage of information during packet processing, enabling stateful processing where information is carried over to subsequent stages. Unfortunately, these existing verifiers fail to consider the stateful nature of packet processing, as they treat register states as fully non-deterministic at the start of each packet processing event. Consequently, they are fundamentally inadequate for verifying complex P4 programs involving registers.

The capability to specify stateful packet processing in P4 has attracted increasing attention from the research community, as evidenced by in-network functions such as load balancing [12], consensus [13], link failure recovery [14], [15] and source protection [16]. However, due to the additional register states, these in-network functions are far more complicated than stateless ones that merely forward packets. Moreover, the existence of registers poses challenges for effective verification using existing P4 verifiers.

To address the stateful aspects of P4 programs, we introduce a novel concept called *packet invariants*. Since the switch constantly processes packets, we can view the packet processing as an infinite loop, where the execution of its body represents the execution of the P4 program. In this context, a packet invariant resembles a loop invariant. It represents a logical formula about the register states that remains true after arbitrary rounds of packet processing. Moreover, we consider a logical formula as a packet invariant if it can conclusively verify the correctness of the P4 program. Consequently, verifying stateful P4 programs can be reduced to inferring proper packet invariants.

Nevertheless, inferring appropriate packet invariants poses a technical challenge. To address this issue, we present an automatic packet invariant generation algorithm specifically tailored for stateful P4 programs. This algorithm efficiently discovers and validates packet invariants in a data-driven manner, offering a novel and effective verification approach for stateful P4 programs.

We have implemented our approach in a prototype tool called P4Inv. In our evaluation, we focused on stateful data plane functionalities such as load balancing [12], consensus [13] and link failure recovery [14], [15]. We compared P4Inv with state-of-the-art P4 analyzers [8], [10], [11]. The experiment results show that P4Inv successfully verifies the correctness of these stateful P4 programs, while other P4 verifiers failed to do so. The performance analysis also demonstrates the efficiency of our packet invariant inferring algorithm.

Contributions. The main technical contributions of this

[†]Fei He is the corresponding author.

paper are as follows:

- We introduce a novel concept called *packet invariants* to address the stateful aspects of P4 programs.
- We propose an automatic packet invariant generation algorithm tailored for stateful P4 programs. To the best of our knowledge, this represents the first attempt to generate and leverage domain-specific invariants for P4 program verification.
- We provide P4Inv, the implementation of our approach. Experimental results demonstrate its effectiveness in verifying stateful P4 programs.

II. Related Work

Network verification. The verification for network can be categorized into control plane and data plane verification.

Studies analyzing the control plane tend to analyze topological properties such as reachability, isolation, and waypoints among nodes. Arc [17] enables fast control plane verification without generating the data plane. Batfish [18] proposes a general approach to detect errors in network configuration files by effectively deriving the data plane. CrystalNet [19] provides means to validate network operations proactively with the emulation technique. Minesweeper [20] translates network configuration into logical formulas and checks satisfiability to detect whether a network state violating the query exists. These studies analyze network configurations and aim to answer queries for network-wide properties.

Data plane verification tools answer similar queries requiring a network snapshot of the data plane. Many techniques have been proposed to verify the data plane from different angles. Anteater [21] uses graph theory algorithms to verify network-wide properties by modeling the network as a graph. HSA [22] develops an optimized formalism for network transfer function to identify network failures. Veriflow [23] aims to enable real-time checking of network-wide invariants while Symnet [24] checks the queries by injecting symbolic packets and tracking them through the network. Our tool, P4Inv, belongs to a data plane verification tool, and is specifically designed for verifying the correctness of P4 programs.

P4 program verification. Existing P4 program verifiers include ASSERT-P4 [7], Aquila [9], bf4 [8], p4v [10], P4-NOD [25] and Vera [11]. These verifiers typically follow the common approach of translating P4 into intermediate domain-specific languages, such as GCL [26], SEFL [24], NOD [27] and C, which can then be processed with existing verification methods. For instance, ASSERT-P4 [7] and Vera [11] translate P4 to C and SEFL, respectively, and employ symbolic execution technique. Aquila [9], bf4 [8] and p4v [10] translate P4 to GCL and utilize the classic program verification technique. [26], [28]. P4-NOD [25] translates P4 to NOD and mainly focuses on network-wide properties like reachability. While ensuring the correctness of P4 programs, these P4 verifiers approximate the states of P4 programs by assuming them fully nondeterministic for each packet processing, disregarding their continuous preservation and ongoing effects. Although this technique facilitates verification regardless of state, these verifiers are fundamentally hindered from reasoning the proper behaviors of stateful P4 programs [29]. On the other hand, P4Inv distinguishes itself from existing P4 verifiers in its automatic generation of packet invariant, which leads to more accurate modeling and verification of stateful P4 programs.

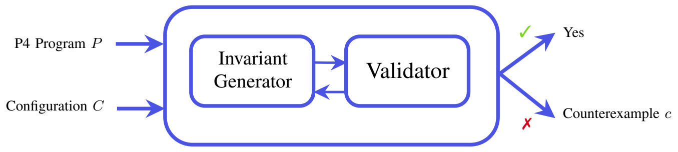

Fig. 1. Overview of P4Inv.

Invariant synthesis techniques. To the best of our knowledge, P4Inv is the first verification tool to leverage invariant synthesis techniques for the verification of P4 programs. Many traditional invariant synthesis techniques can be recognized as white-box approaches, such as abstract interpretation [30], interpolation [31], and counterexample abstraction refinement [32]. In contrast, our verification framework for stateful P4 programs follows the black-box learning setting [33]–[36], which is composed of a learner and a teacher for invariant generation and validation. Typical mechanisms for black-box invariant generation include Houdini [33], decision trees [34], reinforcement learning [37], and support vector machines [38]. P4Inv incorporates a variant of Houdini referred to as Sorcar [35] into the packet invariant learner component. In the future, we expect to discover more efficient and effective generation methods for packet invariants.

III. Overview of P4Inv

P4Inv is an automated verification framework designed for verification of stateful P4 programs. An overview of our verification approach is provided in Fig. 1. Given a P4 program $\mathcal{P}$ with assertions as its specifications and a configuration $\mathcal{C}$ that initializes the registers, P4Inv verifies whether the program $\mathcal{P}$ satisfies the assertions under the provided configuration $\mathcal{C}$. If a violation is identified, P4Inv reports a counterexample c to the developers; otherwise, it reports "Yes," demonstrating that program $\mathcal{P}$ is correct with respect to its assertions.

In contrast to prior efforts, P4Inv automatically generates and leverages domain-specific invariants for P4 program verification. The verification framework of P4Inv comprises two components: the invariant generator and the validator, and it operates iteratively. In each iteration, the generator produces a candidate invariant and sends it to the validator. Subsequently, the validator checks whether the candidate invariant is sufficient for verifying the assertions. If the candidate invariant is deemed as inadequate for verification, the validator generates a counterexample and returns it to the generator, prompting the generator to refine the invariant towards a correct form. The generator and validator cooperate iteratively to revise the invariant until all assertions are successfully verified.

We demonstrate our approach using the Blink failover mechanism [14] as an example, which utilizes retransmissions to detect link failures and trigger rerouting.

```
1  control fs(..., sw, sw_sum, fs_last_index, sw_index) {
2  // declare variables, including fs_index,
3  // index_pre, index_tmp, val_tmp, sum_tmp, etc.
4  ...
5  apply {
6    #include "sliding_window.p4"
7    ...
8    fs_index = E0; // compute the index by hash
9    ...
10   if (B0) {   // if this is a retransmission
11     ...
12     if (B1) { // if the time requirement is met
13       // get index of the previous retransmission
14       fs_last_index.read(index_pre, fs_index);
15       // decrease values in the previous window
16       sw.read(val_tmp, meta.id*4 + index_pre);
17       sw_sum.read(sum_tmp, meta.id);
18       sw.write(meta.id*4 + index_pre, val_tmp - 1);
19       sw_sum.write(meta.id, sum_tmp - 1);
20     }
21     // increase values in the current window
22     sw_index.read(index_tmp, meta.id);
23     sw.read(val_tmp, meta.id*4 + index_tmp);
24     sw_sum.read(sum_tmp, meta.id);
25     sw.write(meta.id*4 + index_tmp, val_tmp + 1);
26     sw_sum.write(meta.id, sum_tmp + 1);
27     ...
28     // update index of the previous retransmission
29     fs_last_index.write(fs_index, index_tmp);
30   }
31   ...
32   // assert: sw[0]+sw[1]+sw[2]+sw[3] == sw_sum[0]
33   sw.read(sw0, 0); sw.read(sw1, 1);
34   sw.read(sw2, 2); sw.read(sw3, 3);
35   sw_sum.read(sw_sum0, 0);
36   @assert(sw0 + sw1 + sw2 + sw3 == sw_sum0);
37 }
38 }
```

Fig. 2. Code snippet [1] for the flow selector component of Blink [14].

P4 Program. We consider one of the main components in Blink, i.e., the flow selector component that is responsible for monitoring the link status. The source code of it is displayed in Fig. 2. For clarity, we have omitted certain code segments (indicated by "..."), a complex arithmetic expression (represented by "E0"), and several intricate Boolean expressions (represented by "B0, B1, ...") from the program.

To monitor the number of retransmission packets for multiple flows with varying time intervals, the flow selector utilizes the sw register. Each element in sw corresponds to the number of retransmission packets in a specific time window, referred to as a "sliding window." Additionally, the flow selector utilizes the sw_sum register to store the sum of all sliding window values. The registers fs_last_index and sw_index are also employed to track the last accessed window and current window indexes during retransmission.

In the apply block, the flow selector imports the sliding window component (at Line 6), which calculates the current window index and ensures it does not exceed the total number of windows. Upon detecting a retransmission packet (at Line 10), the flow selector checks if the time interval exceeds a specific threshold (at Line 12). If the threshold is not exceeded, the flow selector proceeds to decrement the count value of the previous retransmission window. Subsequently, it increments the count value in the current window. These update operations are synchronized with the global sum (at Lines 19 and 26). Afterward, the flow selector updates the retransmission information stored in registers such as fs_last_index for future processing (at Line 29). It is important to note that for the sake of conciseness, we have omitted many other code segments in the program that implement similar updates.

```
1    fs_last_index ALL 0
2    sw_index ALL 0
3    sw_sum 0 0
4    sw 0 0
5    sw 1 0
6    sw 2 0
7    sw 3 0
```

Fig. 3. Configuration for Blink [14].

In the P4 language, registers are accessed through indexes. When referring to the entry of a register r at index i, we will represent it as $r[i]$. There are two types of statements for accessing register entries in P4, namely the read statement `r.read(v,i)` and write statement `r.write(i,e)`. Here, v is a variable and e is an expression. These two statements can be equivalently interpreted to assignment statements $v := reg[i]$ and $reg[i] := e$, respectively.

In Blink, sw_sum[i] is designed to store the sum of sw[i *SIZE+0], sw[i*SIZE+1], ..., sw[i*SIZE+SIZE-1], where SIZE represents the window's size and is assumed to be 4 in this example. The sum register plays a crucial role in the rerouting mechanism. If the sum is greater than the threshold, the fast rerouting is triggered. To this end, we have added an assertion at the end of the program (from Line 32 to Line 36) to confirm the correct implementation of sw_sum[i] when $i == 0$.

Configuration. Rather than considering states as entirely non-deterministic, we observed that the register values are initialized by the configuration and consistently maintained by the P4 program. As a result, we incorporate the configuration into the verification process.

Fig. 3 displays the common zeroing configuration for Blink, where each line represents the initialization of a register, with the first argument denoting the register, the second denoting the index, and the third denoting the value. Specifically, we have introduced a special construct ALL which signifies "for all indexes." It is worth noting that the configuration does not require to cover all registers. Any register that does not appear in the configuration is considered to be initialized randomly.

A configuration can be seen as a collection of assignments to registers. For instance, the first line in Fig. 3 can be interpreted as `forall i.fs_last_index[i] := 0`, and the third line can be interpreted as `sw_sum[0] := 0`.

Verification by Existing Verifiers. To ensure that the assertion holds at every packet processing, the verifier needs to analyze the register fs_last_index, whose element is used as the updating index for windows (at Line 14). If the element of fs_last_index exceeds the window's size, inconsistencies between the sum register and the actual sum

[1]Originated from: https://github.com/nsg-ethz/Blink/tree/master/p4_code.

can occur. For example, consider a scenario where `meta.id` equals 0 and `fs_last_index[fs_index]` equals 4. The value 4 is transferred to `index_pre` at Line 14. Consequently, at Lines 16 to 19, the registers `sw_sum[0]` and `sw[4]` are decreased, leading to an invalidation of the assertion, as `sw_sum[0]` is supposed to only be synchronized with window `sw[0]`, `sw[1]`, `sw[2]`, and `sw[3]`. Therefore, the verifier needs to ensure that for every flow, the value read from `fs_last_index` with the hashed index `fs_index` does not exceed the window's size, i.e., $\forall i.fs\_last\_index[i] < 4$. A similar conclusion can be drawn by examining Lines 22 to 26 for the register `sw_index`, i.e., $\forall i.sw\_index[i] < 4$.

Existing P4 verifiers, such as p4v, Vera, and Aquila, make the assumption that all register values are non-deterministic for each incoming packet. These verifiers treat each packet processing as an independent event and lack the capability to verify properties that require considering the preservation of states. In the given example, the existing verifiers assume that `fs_last_index[i]` and `sw_index[i]` can have arbitrary values, disregarding the conditions mentioned above, which ultimately results in reporting the violation of the assertion.

Packet Invariant. P4Inv uses *packet invariants* to track the values of registers across consecutive packet processing. We provide the definition of packet invariant as follows.

A *register state* refers to an assignment of values to register variables. A *register predicate* is a predicate defined over the register variables, which defines a set of register states. The *strongest postcondition* of a register predicate ϕ with respect to a P4 program $\mathcal{P}$, denoted as $sp(\phi, \mathcal{P})$, represents the set of register states that are reachable from any register state satisfying ϕ after the execution of $\mathcal{P}$.

Definition 1: Given a P4 program $\mathcal{P}$ and a configuration $\mathcal{C}$, a *packet invariant* is a register predicate I satisfying the following two conditions:

1) I holds after the configuration $\mathcal{C}$, i.e.,

$$sp(\top, \mathcal{C}) \rightarrow I \tag{1}$$

2) I is preserved by the execution of P4 program $\mathcal{P}$, i.e.,

$$sp(I, \mathcal{P}) \rightarrow I \tag{2}$$

Note that the assertions are included within the program $\mathcal{P}$. Condition (2) holds only when the packet invariant I is sufficiently strong to verify all the assertions in $\mathcal{P}$. If any assertion is violated, the program will be abnormally terminated, and a special "error" state will be present in $sp(I, \mathcal{P})$, which falsifies $sp(I, \mathcal{P}) \rightarrow I$ since it does not belong to I. Consequently, when we discover a packet invariant that satisfies the conditions outlined in Definition 1, we can effectively validate all the assertions in the P4 program. In other words, the verification of stateful P4 programs can be reduced to inferring proper packet invariants.

Verification by P4Inv. With the technique that will be introduced in Section IV, P4Inv can effectively infer the following packet invariant:

$$\forall i.sw\_index[i] < 4 \wedge \forall i.fs\_last\_index[i] < 4$$
$$\wedge\ sw[0] + sw[1] + sw[2] + sw[3] == sw\_sum[0].$$

Utilizing this packet invariant, P4Inv can successfully verify the validity of the assertion. As far as we know, P4Inv is the first of its kind, to identify domain-specific invariants and utilize them in the verification of P4 programs.

Algorithm 1 Association Graph Construction

```
Input: a P4 program $\mathcal{P}$ and a Configuration $\mathcal{C}$
Output: an association graph $(V, E, E^{=})$
 1: Let $\mathcal{P}_{\mathcal{C}}$ be the concatenation of $\mathcal{C}$ and $\mathcal{P}$
 2: $V \leftarrow \Sigma(\mathcal{P}_{\mathcal{C}})$, $E \leftarrow \emptyset$, $E^{=} \leftarrow \emptyset$
 3: for each relational expression in $\mathcal{P}_{\mathcal{C}}$ do
 4:     Let $e_1$, $e_2$ be its left and right operands, respectively
 5:     $E \leftarrow E \cup \{(t_1, t_2) \mid \forall t_1 \in \Sigma(e_1), \forall t_2 \in \Sigma(e_2)\}$
 6: for each assignment statement in $\mathcal{P}_{\mathcal{C}}$ do
 7:     if it is register-related: $r[e'] := e$ or $e := r[e']$ then
 8:         if $e'$ evaluates to $c$ then
 9:             $E^{tmp} \leftarrow \{(r[c], t) \mid \forall t \in \Sigma(e)\}$
10:         else
11:             $E^{tmp} \leftarrow \{(r[c'], t) \mid \forall r[c'] \in V, \forall t \in \Sigma(e)\}$
12:     else
13:         Let the assignment statement be $v := e$
14:         $E^{tmp} \leftarrow \{(v, t) \mid \forall t \in \Sigma(e)\}$
15:     $E \leftarrow E \cup E^{tmp}$
16:     if $singleton(e)$ then
17:         $E^{=} \leftarrow E^{=} \cup E^{tmp}$
18: return $(V, E, E^{=})$
```

IV. Verification Framework

P4Inv distinguishes itself from existing P4 verifiers [7]–[11] by automatically generating and utilizing packet invariant during the verification of P4 programs. However, a technical challenge of inferring the packet invariant is the intrinsic huge search space. To address the issue, we present a novel verification framework based on the general black-box learning setting [33]–[35] and an effective, domain-specific atomic predicate algorithm for stateful P4 programs. We elaborate on our verification framework in this section.

A. Atomic Predicate Generator

Our approach considers a packet invariant as the conjunction of atomic predicates. A natural challenge is to generate suitable atomic predicates for packet invariant generation based on the given P4 program and configuration. To address this issue, we propose a syntactic-guided heuristic for generating atomic predicates.

Recall that the purpose of packet invariants is to keep track of register values across consecutive packet processing. A packet invariant is defined over a set of register states. Consequently, we require each atomic predicate to be a predicate defined over register variables and constants.

1) Association Graph Construction: In order to generate appropriate atoms, we construct a graph called the *association graph*. This graph captures the associations between registers and constants in the given program and configuration.

Definition 2: An *association graph* is defined as a triple $(V, E, E^=)$. In this graph, the node set V consists of nodes that represent either variables or constants. The edge set E represents associations between nodes, while the edge set $E^=$ represents the equality relation between nodes.

The algorithm for constructing the association graph is illustrated in Algorithm 1. This algorithm takes a P4 program $\mathcal{P}$ and a configuration $\mathcal{C}$ as inputs, and outputs the association graph. Note that the configuration can be interpreted as a collection of assignments to registers. Initially, we concatenate the interpreted assignments of $\mathcal{C}$ to the beginning of the program $\mathcal{P}$, resulting in a merged program called $\mathcal{P}_\mathcal{C}$.

In $\mathcal{P}_\mathcal{C}$, both register and non-register variables are present. We represent a non-register variable as v and a register variable as $r[e]$, where e is an indexing expression. In situations where the value of v or e can be determined statically during parsing, we replace the corresponding variable or index with the determined constant.

In order to construct the association graph, we collect all the constants, non-register variables, and register variables with constant indexes that appear in $\mathcal{P}_\mathcal{C}$. These elements are gathered into a set named $\Sigma(\mathcal{P}_\mathcal{C})$. Note that register variables with non-constant indexes are not included in this set, as they can refer to any element within the registers. The set $\Sigma(\mathcal{P}_\mathcal{C})$ is utilized as the node set for the association graph.

We will now proceed with establishing the relations E. Given two elements t_1 and t_2 from $\Sigma(\mathcal{P}_\mathcal{C})$, there are two situations in which t_1 is associated with t_2 (i.e., $(t_1, t_2) \in E$):

- If there is a relational expression in the program where t_1 and t_2 appear in the operands on opposite sides of the relational operator (Lines 4 to 5 of Algorithm 1).
- If there is an assignment statement in the program where t_1 represents the left operand, and t_2 represents the element in the right operand (Lines 7 to 14 of Algorithm 1).

In the latter case, when the register in the assignment statement (i.e., register read or write statement) is accessed with a non-constant index, it is necessary to consider all possible elements of the register and associate each of them with t_2, as in Line 11 of Algorithm 1.

However, when generating atoms, the relation E may be too coarse for precise analysis. In order to address this limitation, we introduce an equality relation $E^=$ as its complement. As the name suggests, for any two elements t_1 and t_2 in $\Sigma(\mathcal{P}_\mathcal{C})$, $(t_1, t_2) \in E^=$ if t_1 is equal to t_2 at some point in $\mathcal{P}_\mathcal{C}$. Discovering all semantically equivalent elements can be time-consuming and sometimes impossible. Therefore, we choose to identify evident equality relations between program elements. Specifically, we identify such relations when there is an assignment statement in the form of $t_1 := t_2$, $r[e] := t_2$, or $t_1 := r[e]$, where $r[e]$ represents a register variable with a non-constant index (at Lines 16-17 and recall that $r[e]$ is not an element of V).

Algorithm 2 Atomic Predicate Generation

```
Input: an association graph $(V, E, E^=)$
Output: an atomic predicate set $AP$
 1: $AP \leftarrow \emptyset$
 2: for each register $r[c] \in V$ do
 3:     for each reachable constant $cc \in V$ via $E$ do
 4:         $AP \leftarrow AP \cup \mathrm{genAtom}(r[c], cc)$
 5:     for each reachable register $r'[c'] \in V$ via $E$ do
 6:         $AP \leftarrow AP \cup \mathrm{genAtom}(r[c], r'[c'])$
 7: for each branch condition and assertion $b$ in $\mathcal{P}_\mathcal{C}$ do
 8:     if each non-register variable $v$ of $b$ is reachable from
        some register $r[c] \in V$ via $E^=$ then
 9:         $b^r \leftarrow b[v/r[c]]$ for each $v$ in $b$
10:         $AP \leftarrow AP \cup \{b^r\}$
11: return $AP$
```

2) Atoms Generation: Given the association graph we constructed, we can now proceed with generating atomic predicates. The generation algorithm is presented in Algorithm 2. A register variable, denoted by $r[c]$, is considered *associated* with a constant cc (or another register variable, $r'[c']$) if there exists a path from $r[c]$ to cc (or $r'[c']$) through edges in E. For each pair of such associated elements, we employ the function *genAtom* to generate the corresponding atoms (at Lines 4 and 6 of Algorithm 2). It is important to note that our focus in this context is solely on these two types of associated elements, as an atom encompasses only these two program elements.

The *genAtom* function generates atoms using predefined templates. On the one hand, we require the generated atoms to be sufficiently expressive, which is necessary for constructing a proper packet invariant to conclude the verification. On the other hand, generating too many atoms can bring significant computational overhead to the subsequent packet invariant generation. Therefore, designing an appropriate template for *genAtom* is crucial to ensure the effectiveness of our verification framework.

In this paper, we focus on generating atoms in the following forms: $r[c] \prec cc$, $cc \prec r[c]$, and $r[c] \prec r'[c']$, where $\prec \in \{<, \leq\}$. We have opted not to generate atoms like $r_1[c_1] \prec r_2[c_2] + r_3[c_3]$, or $(r[c])^2 \prec cc$, as these complex atom templates could lead to a significantly large search space during subsequent packet invariant generation. To summarize, the function *genAtom* is defined as follows:

$$genAtom(r[c], cc) = \{r[c] \prec cc, cc \prec r[c]\}$$
$$genAtom(r[c], r'[c']) = \{r[c] \prec r'[c'], r'[c'] \prec r[c]\}$$

Specifically, the `ALL` index in the configuration is treated as a special constant. Consequently, $r[\texttt{ALL}]$ is also a register variable with a constant index and can be included in the association graph. If $r[\texttt{ALL}]$ is associated with t, where t is a constant or another register variable (without the `ALL` index), we introduce $\forall$ to interpret the `ALL` index. As a result, we generate atoms $\forall i.r[i] \prec t$ and $t \prec \forall i.r[i]$. If t is also a register variable with the `ALL` index and it has the same size as r, denoted as $r'[\texttt{ALL}]$, we generate atoms $\forall i.r[i] \prec r'[i]$

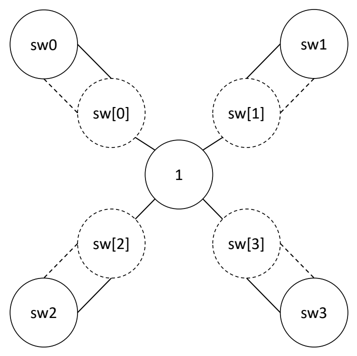

Fig. 4. Subgraph of the association graph for Blink.

and $\forall i.r'[i] \prec r[i]$. It is worth noting that we avoid generating complex quantified atoms such as $\forall i \forall j.r[i] \prec r'[j]$, which involve nested quantifiers that are difficult to be efficiently solved by existing SMT solvers.

Furthermore, we acknowledge the significance of branch conditions and assertions in program verification. However, these Boolean formulas contain non-register variables, making them unsuitable for direct use as atoms. To overcome this limitation, we employ the equality relation $E^{=}$ to identify register variables that can substitute for the non-register variables in these formulas. When all non-register variables in a Boolean formula can be replaced with register variables, the resulting formula becomes a register predicate and can be included in AP as an atom (Lines 8 to 10). It is important to note that we utilize $E^{=}$ instead of E for substitutions, since the latter relation is too coarse and can lead to a combination explosion when substituting.

Example. To better illustrate the workflow of the atomic predicate generator, let's consider the Blink mechanism presented earlier in Fig. 2. For simplicity, we will use i to represent the constants 0, 1, 2 and 3. We will demonstrate the generation process based on Line 25 and Lines 33-36. At Line 25, since the index does not evaluate to a constant, we link every corresponding register node with a constant index (i.e., `sw[i]`) to the variable and constant in the value expression (i.e., constant 1 and variable `val_tmp`). At Lines 33-36, both the relation E and the equality relation $E^{=}$ are generated, such as links between `sw0` and `sw[0]`. As a result, the association graph shown in Fig. 4 is obtained, with `val_tmp` omitted for brevity. In the graph, the dashed node represents the register node, the solid line represents relation E, and the dashed line represents the equality relation $E^{=}$. Since `sw[i]` is reachable by the constant 1 via E, we can obtain atoms like $sw[0] \leq 1$. Additionally, since `sw[i]` is reachable from each other, we can obtain atoms like $sw[0] \leq sw[1]$. Furthermore, since the variable `swi` is reachable via $E^{=}$ from `sw[i]`, and similarly `sw_sum0` is reachable via $E^{=}$ from `sw_sum[0]`, we can obtain $sw[0] + sw[1] + sw[2] + sw[3] == sw\_sum[0]$ by replacing variables with the corresponding registers at Line 36.

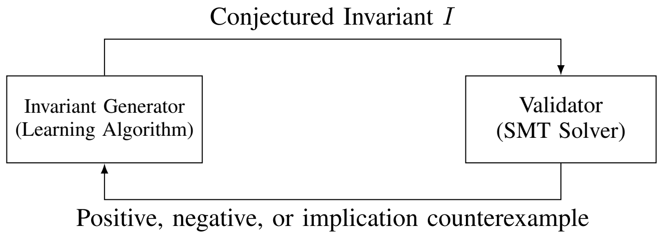
Conjectured Invariant I
Invariant Generator (Learning Algorithm)
Validator (SMT Solver)
Positive, negative, or implication counterexample

Fig. 5. The general black-box framework.

B. Invariant Generator

After generating atomic predicates, we employ a black-box framework [34], [36] as shown in Fig. 5 to automate the generation and validation of packet invariants, which is the general framework for learning conjunctive invariants over a finite class of predicates. For completeness, we briefly describe the core idea in the context of packet invariant as follows.

The black-box framework consists of two main components: the invariant generator and the validator. It operates iteratively, with an initial setting of the conjectured invariant I as $\top$. In each iteration, the validator receives the conjectured invariant I from the invariant generator and checks if it satisfies the conditions (1) and (2) for the packet invariant. If I does not meet these conditions, the validator provides a counterexample, which is then passed on to the invariant generator to improve the conjectured invariant for the next iteration. Specifically, the framework categorizes the counterexample into one of the following three forms:

1) If the conjectured invariant I falsifies condition (1), then the validator returns a *positive counterexample* σ such that $\sigma \in sp(\top, \mathcal{C})$ but $\sigma \not\models I$.
2) If the conjectured invariant I leads to the violation of assertion, then the validator returns a *negative counterexample* σ such that $\sigma \models I$ but $sp(I, \mathcal{P})$ contains the "error" state.
3) If I falsifies condition (2) but $sp(I, \mathcal{P})$ does not contain the "error" state , then the validator returns a *implication counterexample* such that $\sigma_1 \models I$ but $\sigma_2 \not\models I$.

Intuitively, a positive counterexample satisfies the configuration but falsifies the conjectured invariant. A negative counterexample adheres to the conjectured invariant but results in a violation of the assertion. An implication counterexample indicates that the conjectured invariant is not inductive.

The invariant generator aims to conjecture invariant I over the atomic predicate set $AP = \{p_1, \cdots, p_n\}$ using the collected counterexample set Cex. The counterexample set is divided into three categories: S_+ for positive counterexamples, S_- for negative counterexamples, and S_H for implication counterexamples. A conjectured invariant I is said to be *consistent* with Cex if

1) I is consistent with S_+, i.e., $\forall \sigma \in S_+.\ \sigma \models I$,
2) I is consistent with S_-, i.e., $\forall \sigma \in S_-.\ \sigma \not\models I$, and
3) I is consistent with S_H, i.e., $\forall (\sigma_1, \sigma_2) \in S_H$, $\sigma_1 \models I$ implies $\sigma_2 \models I$.

We employ the widely-used Houdini algorithm [33] for synthesizing conjunctive invariants consistent with Cex. Initially, $I = p_1 \wedge \cdots \wedge p_n$ and we remove all atomic predicates p from I that violate a positive example (i.e., $\exists \sigma \in S_+ . \sigma \not\models p$) to derive the largest conjunctive formula I that is consistent with S_+. Then, for every $(\sigma_1, \sigma_2) \in S_H$ that is not satisfied by I, indicating that $\sigma_1 \models I$ and $\sigma_2 \not\models I$, it adds c_2 as a new positive counterexample to S_+ to enforce σ_2 to satisfy I, as otherwise I will not be consistent with S_H. Furthermore, we repeat these two steps until a fixed point is reached, resulting in the largest conjunctive set that is consistent with S_+ and S_H. Finally, we verify if I satisfies S_-. If it does, then I represents the formula that is consistent with Cex; otherwise, no consistent conjunctive formula based on AP exists. By definition, if there exists no counterexample that violates the conditions (1) and (2), the conjectured invariant I is no other than a packet invariant. As proved in [35], the algorithm ensures convergence, or it reports that no conjunctive invariant exists over the provided atomic predicates.

C. *Validator*

The role of the validator is to verify whether the conjectured formula I, satisfies the conditions (1) and (2) of packet invariant. Since the P4 language does not contain loop instructions, the only component that may contain a loop is the parser. According to the P4 specification [39], the hardware may limit the max number of times a packet can visit a parser state. Therefore, infinite loops are impossible. As a result, we follow the unrolling technique used in bf4 [8]. By taking advantage of the loop-free feature of P4, we can encode the conditions of packet invariant into SMT formulas, leveraging the classic program verification approach [26], [28], and check their satisfiability using SMT solvers. Similar to the approach used in p4v [10] and bf4 [8], we expand the application of tables into nondeterministic decision processes for all actions, assuming that all rules are possible.

V. Implementation

In this section, we present the detailed implementation of P4Inv. The implementation comprises roughly 10,000 lines of code, written in C# and C++. Along with the implementation for the validator, atomic predicates and invariant generator, we describe P4Inv's noteworthy verification features.

To verify the validity of the packet invariant, namely conditions (1) and (2), P4Inv employs a two-step process. Firstly, similar to p4v [10], it translates the P4 code into an intermediate representation called Boogie [40]. Then, it utilizes the Boogie verifier [40] to encode the conditions as SMT formulas and uses the Z3 solver [41] to check their satisfiability. Our translator is implemented as a separate backend to p4c, the P4 compiler suite. The implementation of the atomic predicate generator is integrated with the translator since both require traversing the P4 syntax tree. For the invariant generator, we build it upon the Sorcar implementation [35], which is an extension of Houdini algorithm [33] that aims to generate tighter conjunctive formulas based on heuristics. Furthermore, we adapt its implementation to be compatible with the latest version of the Boogie verifier.

A. *Verification Features*

To facilitate network functionality verification of stateful P4 programs, P4Inv offers different features as follows.

Assertion annotation. Network programmers can use P4Inv to query general security and functionality properties using assertions. Line 36 of Fig. 2 illustrates how developers can annotate P4 programs with the `@assert` keyword to indicate intended behavior. The assertion language syntax currently supports full boolean expression features in P4, allowing programmers to write properties similar to P4 branch conditions. This feature relieves programmers of learning other domain-specific languages.

Assumption annotation. Bug-free P4 programs can be difficult to code due to the non-deterministic contents of incoming packets in actual environments. In specific cases, we may only expect switches to work as intended in particular situations. For example, we may not anticipate malicious entities in a fail-recover consensus protocol. To account for this, P4Inv allows programmers to use the `@assume` keyword to inject assumptions about the network environment during verification. Consequently, program traces that violate the assumed condition will not be considered in verification.

User-defined packet invariant. The manual specification of a user-defined packet invariant is possible through P4Inv. With the `@invariant` keyword, developers can specify packet invariants manually. With this feature, expert knowledge can be harnessed to enhance verification by specifying packet invariants beyond the limits of the template presented in Section IV-A, thereby further assisting the verification process.

B. *Limitations*

While P4Inv provides multiple verification features, it also has limitations in its current version. To begin with, P4Inv fully supports V1Model, but its support for TNA and PSA is limited. This limitation arises from the fact that full support for TNA and PSA necessitates more sophisticated modeling for the ingress and egress process than the V1Model. Nonetheless, we believe that the concept of packet invariant still functions effectively even in these intricate architectures, and providing full support for them is part of our future endeavors.

It is also important to note that for specific cases, the packet invariant generation process in P4Inv may not be general enough to cover them. However, it's worth noting that P4Inv suffices to verify stateful functionalities while existing P4 verifiers fail to prove, as outlined in Section VI. Additionally, allowing user-defined packet invariants enables users to leverage expert knowledge to mitigate the situation.

VI. Evaluation

In this section, we present our evaluation results and analysis, which demonstrates the effectiveness and efficiency of P4Inv in verifying stateful P4 programs.

A. Evaluation Settings

Benchmark. To test P4Inv's ability to verify stateful P4 programs, we collected five non-trivial benchmarks varying size and complexity. These benchmarks implement stateful network functionalities including consensus, load balancing, and link recovery. These programs were obtained from the open-source community. For each benchmark, we meticulously examined its design and manually formalized safety properties that we expect them to be satisfied. Currently we only provide the common zeroing configuration to P4Inv.

Setup. We compared P4Inv with other state-of-the-art P4 verifiers, including:

- bf4 [8], a state-of-the-art verification approach for P4 programs that uses a mix of static verification techniques.
- p4v [10], an innovative verification approach for P4 programs that resolves verification conditions.
- Vera [11], a novel verification approach for P4 programs that is grounded in symbolic execution.

For p4v, since it is not open-sourced, we followed its approximate implementation introduced in [8]. For bf4 and Vera, we leveraged the publicly released code. All experiments in this section were conducted on a machine with an Intel(R) Core(TM) i5-12400 2.50 GHz CPU and 32GB RAM.

B. Proving Safety Properties

With evaluations, we showcase the effectiveness of P4Inv in verifying the correctness of stateful P4 programs.

Blink [14]. Blink is a data-driven mechanism that uses characteristic failure signals to detect link failure and trigger rerouting. As shown previously in Section III, P4Inv successfully verifies the correct implementation of the sum register by inferring packet invariant. Moreover, note that we made slight modifications to the original version of the Blink implementation. These modifications involve changing the sliding window size in `macro.p4` from 10 to 4 due to the limitation of the backend solver and addressing the issue of register index overflow (by introducing an additional branch that ensures the indexing metadata will not go out of bounds).

P4xos [13]. The P4xos explores the utilization of a stateful data plane to improve the performance of the Paxos protocol [42]. Within the P4xos, there are multiple roles involved in the protocol. In this study, our focus is on verifying the correctness of the acceptor and learner. Additionally, note that it is crucial to initialize the round registers with zero for the proper functioning of P4xos, as illustrated in [13].

The role of the acceptor in the Paxos protocol is to achieve agreement on a specific value. Each acceptor is tasked with selecting and voting for a value to ensure consensus is maintained for each instance. One important property of the acceptor is that the stored consensus value round must always be less than the current valid round. This ensures that outdated messages are ignored, which is crucial for consensus. However, proving the correctness of this property is non-trivial due to the need to keep track of the round and value round registers. By inferring packet invariant

$$\forall i. registerVRound[i] \leq registerRound[i],$$

P4Inv can successfully prove the safety property while bf4, p4v and Vera failed to verify.

The learner's role in the Paxos protocol is to collect votes from acceptors and deliver the corresponding vote value once a quorum (i.e., a majority of votes) is obtained for a specific instance. In the implementation of the learner for P4xos, it is important to ensure that no abnormal vote status will be recorded, meaning that the number of votes should not exceed the number of acceptors during any vote round. However, we discovered a potential issue where malicious acceptor IDs might be included in the packet, which would violate this property. Nevertheless, P4Inv can prove a stronger version of the property: when only valid acceptor IDs are assumed, the vote statuses will remain valid regardless of the number of packet processing times.

To verify this property, we introduce an additional branch condition to the implementation that accepts only valid IDs. By effectively inferring packet invariant

$$\forall i. registerHistory2B[i] \leq \mathit{MAX\_ACCEPTOR\_NUM},$$

P4Inv can ensure the validity of the safety property.

P4NIS [12]. The Processors based Network Immune Scheme (P4NIS) leverages the stateful data plane to protect against packet eavesdropping. This scheme incorporates three defense lines. The first line of defense employs load balancing to distribute packet traffic across various network links. The second line of defense focuses on encrypting the port fields of incoming packets. Finally, the third line of defense utilizes existing encryption measures.

For P4NIS, we focus on verifying the correctness of the first line of defense, which utilizes a register called `count` to distribute packets in a circular manner. It is crucial to guarantee that this defense should not entail implicit drops, as this is a common bug in P4 programs and can negatively impact subsequent defense lines. However, if the value in the `count` register is incorrectly configured (i.e., a value that exceeds the number of available circular ports), it can lead to implicit drops. Nonetheless, this line of defense does not trigger any implicit drops when provided with zeroing configuration. P4Inv successfully verifies this property by inferring

$$count[0] \leq \mathit{NUM\_PORTS}.$$

On the other hand, bf4, p4v and Vera failed to verify the property and produce false positives.

Fast Reroute [15]. The benchmark [15] implements a fast in-network failover mechanism in P4 without control plane interference. This mechanism utilizes the stateful feature of P4 to indicate the availability status of ports. Additionally, we added minor modifications to the implementation by utilizing P4 registers to record the reroute status of packets, which reduces the number of bits transmitted in the network.

TABLE I
RESULTS FOR VERIFICATION OF STATEFUL P4 PROGRAMS.

Benchmark	P4Inv	Bf4	P4v	Vera
Blink [14]	✓	✗	✗	✗
P4xos-acceptor [13]	✓	✗	✗	✗
P4xos-learner [13]	✓	✗	✗	✗
P4NIS [12]	✓	✗	✗	✗
Fast Reroute [15]	✓	✗	✗	✗

✓: A tool successfully validates the safety property.
✗: A tool reports a spurious counterexample.

An essential requirement of the fast reroute mechanism is to ensure that a packet is transmitted through the parent link or the available links configured for rerouting. By effectively discovering the following packet invariant

$$\forall i.\text{pktcurr}[i] \leq PORT,$$

P4Inv is able to prove the safety property while other P4 verifiers fail to do so.

Effectiveness. Table I presents the description and evaluation results in detail for the benchmark. The symbol ✓ denotes a successful proof of the corresponding safety property by the tool, whereas the symbol ✗ signifies a false positive (i.e., the tool reports a spurious counterexample that will not occur under the configuration).

The results presented in Table I indicate that P4Inv outperforms existing verifiers in validating the functional property of stateful P4 programs.

Among the compared verifiers, none of them succeeded in verifying the correctness of stateful network functionalities. Although they support verification for P4 programs, they overestimate the register states of P4 programs by always assuming that they are non-deterministic during every packet processing. Due to their inability to trace the states, they cannot accurately capture the behavior of stateful P4 programs. Hence, they generate false positives.

On the other hand, P4Inv accurately verifies the safety properties of stateful P4 programs, effectively validating all anticipated behaviors. Verifying such queries requires it to infer the precise range of registers, track the interactions between registers, and reason about the complex state operations for each packet processing, as the packet invariants indicated. P4Inv captures the interactions between states and the P4 program accurately by effectively discovering packet invariants, and consequently solving the verification task.

Efficiency. All verification tasks finish in less than 6 seconds except for Blink. P4Inv takes about 7 hours to verify the correctness of Blink. We surmise that the time cost may result from the presence of numerous states (on the order of 10k register elements) and the requirement for complex invariants to verify the property. However, it is important to note that even when provided with the correct invariant directly, meaning that the program verification is done solely with the Boogie verifier backend, it still takes over an hour to prove the safety property. Consequently, we conjecture that the inherent complexity of the verification task contributes to the relatively long proofing time. For the rest of the cases, P4Inv is able to verify their correctness efficiently.

C. *Case Study: switch.p4*

As far as we know, P4Inv is the first tool to use packet invariant generation for correctness verification of stateful P4 programs. However, in this section, we experimentally compared P4Inv with bf4, p4v and Vera to gain confidence in its verification capabilities for finding bugs of P4 programs.

Our experiment is conducted on `switch.p4`, which is a representative open-source P4 program that comprises over 6000 effective lines of code. `switch.p4` is an industrial-grade router implementation containing complex packet processing operations. In the experimental setting, we checked for the common P4 program bug "invalid header". We adopted the definition of invalid header introduced in [8] and implemented the corresponding assertion instrumentation technique for P4Inv.

P4Inv, bf4 and p4v all successfully confirm the existence of invalid accesses for 32 headers, while reporting safe for the remaining ones. It takes 82s for bf4 to verify `switch .p4` while p4v takes about 1 minute. For P4Inv, verifying the invalid accesses takes about 3 minutes. For Vera, it takes about 10s for a concrete snapshot of `switch.p4`, whereas identifying all invalid accesses takes over 25 minutes.

As a result, P4Inv is capable of effectively and efficiently verifying P4 programs compared to existing P4 verifiers. Bf4 and the approximation implementation for p4v optimize the control flow graph of P4 programs using various techniques, such as constant propagation, local copy propagation [43], and domain-specific slicing to achieve better efficiency. P4Inv, bf4 and p4v utilize the classic program verification technique [26], [28]. Vera utilizes symbolic execution and is effective in identifying all program paths that lead to vulnerabilities. Nevertheless, as indicated in [11], it might encounter the path explosion issue inherent in symbolic execution, hindering complete code coverage, especially for large P4 programs.

VII. Conclusions

This paper introduces P4Inv, an automated verification tool designed for stateful P4 programs. Additionally, we propose a novel concept called *packet invariants* for verifying stateful P4 programs. To the best of our knowledge, this is the first attempt to employ domain-specific invariants in the verification of P4 programs. Furthermore, we present an algorithm that efficiently infers packet invariants through a data-driven approach. Our evaluation showcases the effectiveness of P4Inv in verifying stateful P4 programs.

The authors have provided public access to their code and data at https://thufv.github.io/research/p4inv.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (No. 62072267 and No. 62021002).

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] Y. Lu and S. Zhu, "Sdn-based TCP congestion control in data center networks," in *IPCCC*. IEEE Computer Society, 2015, pp. 1–7.

[3] S. Azodolmolky, P. Wieder, and R. Yahyapour, "Sdn-based cloud computing networking," in *2013 15th international conference on transparent optical networks (ICTON)*. IEEE, 2013, pp. 1–4.

[4] O. Flauzac, C. Gonzalez, A. Hachani, and F. Nolot, "SDN based architecture for iot and improvement of the security," in *AINA Workshops*. IEEE Computer Society, 2015, pp. 688–693.

[5] S. Jain, M. Khandelwal, A. Katkar, and J. Nygate, "Applying big data technologies to manage qos in an SDN," in *CNSM*. IEEE, 2016, pp. 302–306.

[6] 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," *Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, 2014.

[7] L. Freire, M. C. Neves, L. Leal, K. Levchenko, A. E. S. Filho, and M. P. Barcellos, "Uncovering bugs in P4 programs with assertion-based verification," in *SOSR*. ACM, 2018, pp. 4:1–4:7.

[8] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu, "bf4: towards bug-free P4 programs," in *SIGCOMM*. ACM, 2020, pp. 571–585.

[9] B. Tian, J. Gao, M. Liu, E. Zhai, Y. Chen, Y. Zhou, L. Dai, F. Yan, M. Ma, M. Tang, J. Lu, X. Wei, H. H. Liu, M. Zhang, C. Tian, and M. Yu, "Aquila: a practically usable verification system for production-scale programmable data planes," in *SIGCOMM*. ACM, 2021, pp. 17–32.

[10] J. Liu, W. T. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Cascaval, N. McKeown, and N. Foster, "p4v: practical verification for programmable data planes," in *SIGCOMM*. ACM, 2018, pp. 490–503.

[11] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 programs with vera," in *SIGCOMM*. ACM, 2018, pp. 518–532.

[12] G. Liu, W. Quan, N. Cheng, N. Lu, H. Zhang, and X. Shen, "P4NIS: improving network immunity against eavesdropping with programmable data planes," in *INFOCOM Workshops*. IEEE, 2020, pp. 91–96.

[13] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé, "P4xos: Consensus as a network service," *IEEE/ACM Trans. Netw.*, vol. 28, no. 4, pp. 1726–1738, 2020.

[14] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *NSDI*. USENIX Association, 2019, pp. 161–176.

[15] R. Sedar, M. Borokhovich, M. Chiesa, G. Antichi, and S. Schmid, "Supporting emerging applications with low-latency failover in P4," in *NEAT@SIGCOMM*. ACM, 2018, pp. 52–57.

[16] S. Lindner, M. Häberle, F. Heimgaertner, N. Nayak, S. Schildt, D. Grewe, H. Löhr, and M. Menth, "P4 in-network source protection for sensor failover," in *Networking*. IEEE, 2020, pp. 791–796.

[17] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *SIGCOMM*. ACM, 2016, pp. 300–313.

[18] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. D. Millstein, "A general approach to network configuration analysis," in *NSDI*. USENIX Association, 2015, pp. 469–483.

[19] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "Crystalnet: Faithfully emulating large production networks," in *SOSP*. ACM, 2017, pp. 599–613.

[20] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *SIGCOMM*. ACM, 2017, pp. 155–168.

[21] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *SIGCOMM*. ACM, 2011, pp. 290–301.

[22] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*. USENIX Association, 2012, pp. 113–126.

[23] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *NSDI*. USENIX Association, 2013, pp. 15–27.

[24] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *SIGCOMM*. ACM, 2016, pp. 314–327.

[25] N. Lopes, N. Bjørner, N. McKeown, A. Rybalchenko, D. Talayco, and G. Varghese, "Automatically verifying reachability and well-formedness in p4 networks," *Technical Report, Tech. Rep*, 2016.

[26] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[27] N. P. Lopes, N. S. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *NSDI*. USENIX Association, 2015, pp. 499–512.

[28] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[29] Q. Kang, J. Xing, Y. Qiu, and A. Chen, "Probabilistic profiling of stateful data planes for adversarial testing," in *ASPLOS*. ACM, 2021, pp. 286–301.

[30] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *POPL*. ACM Press, 1979, pp. 269–282.

[31] R. Jhala and K. L. McMillan, "A practical and complete approach to predicate refinement," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 3920. Springer, 2006, pp. 459–473.

[32] T. Ball and S. K. Rajamani, "The SLAM toolkit," in *CAV*, ser. Lecture Notes in Computer Science, vol. 2102. Springer, 2001, pp. 260–264.

[33] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *FME*, ser. Lecture Notes in Computer Science, vol. 2021. Springer, 2001, pp. 500–517.

[34] P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan, "Horn-ice learning for synthesizing invariants and contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 131:1–131:25, 2018.

[35] D. Neider, S. Saha, P. Garg, and P. Madhusudan, "Sorcar: Property-driven algorithms for learning conjunctive invariants," in *SAS*, ser. Lecture Notes in Computer Science, vol. 11822. Springer, 2019, pp. 323–346.

[36] A. Champion, T. Chiba, N. Kobayashi, and R. Sato, "Ice-based refinement type discovery for higher-order functional programs," *J. Autom. Reason.*, vol. 64, no. 7, pp. 1393–1418, 2020.

[37] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning loop invariants for program verification," in *NeurIPS*, 2018, pp. 7762–7773.

[38] J. Li, J. Sun, L. Li, Q. L. Le, and S. Lin, "Automatic loop-invariant generation and refinement through selective sampling," in *ASE*. IEEE Computer Society, 2017, pp. 782–792.

[39] P. L. Consortium. (2022) $P4_{16}$ language specifcation.

[40] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*, ser. Lecture Notes in Computer Science, vol. 4111. Springer, 2005, pp. 364–387.

[41] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340.

[42] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[43] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, ser. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.