Synthesizing Runtime Programmable Switch Updates

Yiming Qiu   Ryan Beckett[†]   Ang Chen

Rice University   [†]*Microsoft*

Abstract

We have witnessed a rapid growth of programmable switch applications, ranging from monitoring to security and offloading. Meanwhile, to safeguard the diverse network behaviors, researchers have developed formal verification techniques for high assurance. As a recent advance, network devices have become runtime programmable, supporting live program changes via partial reconfiguration. However, computing a runtime update plan that provides safety guarantees is a challenging task. FlexPlan is a tool that identifies step-by-step runtime update plans using program synthesis, guaranteeing that each transition state is correct with regard to a user specification and feasible within switch memory constraints. It develops novel, domain-specific techniques for this task, which scale to large, real-world programs with sizable changes.

1 Introduction

Programmable switches accelerate the velocity of change and facilitate innovation [11, 12, 14]. The community's ideal is to develop and deploy new features quickly in the network, without upgrading switch hardware. This has led to two synergistic lines of pursuit. On one hand, researchers have seized this opportunity to develop a slew of *switch applications*, such as monitoring [24, 48], security [31, 54, 55, 58], advanced routing [25, 33, 41], and offloading [27, 28]. On the other hand, to mitigate the potential risks of frequent changes, *formal verification* of switch programs promises to eliminate network bugs and provide high assurance [13, 18, 38, 51]. Combined, the two lines of work pave the way toward a featureful and reliable network infrastructure.

As of late, network programmability is at a new inflection point. Whereas traditional P4 programmable switches [10] require offline procedures for program updates (e.g., draining user traffic, reflashing the data plane with a new program, and undraining traffic), *runtime programmable switches* [4, 19, 52, 53] increase the benefits of programmability even further by supporting live program changes. Upon an update request (e.g., due to tenant requirements or infrastructure upgrades), the switch program is modified online to effect the change, by incrementally adding and removing match/action tables in a running program based on the "delta" [19, 53]. Compared to the approach of recompiling the changed program and reflashing the data plane from scratch, runtime updates enable rapid deployment of new features. Runtime programmability has become available in a variety of targets [19, 52, 53], including commercial off-the-shelf switch ASICs [3, 5, 9].

However, runtime programmability introduces another layer of correctness concerns. It is known that live network updates come with risks—they result in intermediate states with different behaviors from the initial and final networks [29, 44, 45]. Even in a traditional network, updates must be carefully staged to ensure that the transition is free of error [36]. This risk and the challenges it raises are only magnified in programmable switch updates. A step-by-step update to a deployed P4 program, while the switch is in use, could expose partial and potentially unsafe program snapshots to user traffic. For instance, if not careful, an old ACL table may have been removed before the updated ACL is installed, leading to a transient program snapshot without access control. Although we could attempt to make all changes in a single step and avoid intermediate states, in practice, this is only feasible for minuscule updates. The "delta" corresponding to the update needs to be prepared in the switch scratch memory before activated [6, 53]; thus, this leads to a resource utilization peak that may exceed the available memory. A large delta often needs to be broken down into smaller steps, where each step prepares and applies only a fraction of the change [53].

To safeguard runtime network updates, we must again rely on formal reasoning techniques. Existing work in runtime programmability either asks the user to prescribe step-by-step updates [19] or employs algorithms that do not provide semantic guarantees [53]; thus, they do not offer the same level of assurance. Existing verification work, on the other hand, focuses on certifying the correctness of a single P4 program [18, 38, 51], but cannot help *identify a correct-by-construction sequence* for program transition. Thus, these verification techniques cannot guarantee the correctness of intermediate program snapshots during a transition, and bugs could be unwittingly introduced into the network. We believe that customizing *program synthesis* techniques to this new problem domain will bear much fruit. If we can formally synthesize a *safe* (i.e., conforming to a specification) and *feasible* (i.e., within switch resource limits) transition plan, then we can be confident about its effect on the network.

We develop such a tool called FlexPlan. It takes in a P4 program with annotated changes and a user specification, and produces such a transition plan. FlexPlan draws inspirations from a powerful approach to program synthesis—CEGIS, or counterexample-guided inductive synthesis [47]—and develops a variety of domain-specific techniques for our problem at hand. At a high level, the CEGIS algorithm navigates the search space by iterating between a "proposal" phase, which suggests potentially correct programs (e.g., transition plans), and a "verification" phase, which proves or disproves the

proposals. The verification phase also produces counterexamples (i.e., packets) upon failure, which are learned by the next proposal to guide the search. CEGIS has found success in many synthesis problems [17] and more recently, in networking [13, 22, 59]. However, runtime programmable switch updates raise distinct challenges that require novel designs.

The first challenge stems from the fact that FlexPlan needs to synthesize not just one single program but a *sequence* of program snapshots; further, each snapshot must successively modify the previous one safely and stay within the resource constraints. To cast this into a CEGIS framework, we develop two concise encodings that articulate the essence of this synthesis. A *version sketch* is derived from the switch program with annotated changes, adding version variables at change sites to represent the change progress. This building block provides a uniform representation for any intermediate program state that could appear in the transition. Further, a *sequence sketch* concatenates multiple version sketches, while constraining them to modify each other and make progress toward the final program. The synthesis target, therefore, is correct assignments to the version variables across snapshots—these are the program "holes" in the verbiage of the *program sketching* [47] synthesis framework.

The scalability bottlenecks of this synthesis represent the second challenge. Existing verification efforts are already challenged by the large SMT formulas produced by complex switch programs [38, 50, 51], but FlexPlan needs to reason about stacks of these formulas in each proposal/verification phase. To accelerate the synthesis, we develop two domain-specific techniques to shrink the problem sizes whenever possible—to as close as that of single program snapshots. *Snapshot learning* extracts insights about the synthesis from a single snapshot, and generalizes that knowledge to all subsequent snapshots. *Snapshot verification* shatters a proposed transition sequence into individual snapshots for divide-and-conquer. We arrange these techniques carefully to ensure that reasoning about the safety of a sequence from snapshot-based properties still produces a sound analysis.

Finally, we also leverage a unique property in the FlexPlan synthesis problem—its *diagnosability*—to perform introspection into the synthesis process. For a traditional CEGIS problem, the synthesis tool cannot know beforehand whether or not a correct solution exists. FlexPlan, however, can check the initial and final programs against the safety specification to see whether or not some safe transitions exist. If both programs are correct, then a safe transition must exist; a safe transition may not be feasible, however, due to resource constraints. To check feasibility, FlexPlan incrementally grows the transition sequence length—so that each step in the transition sequence makes smaller and smaller changes to approach feasibility—while performing another introspection to decide when to stop trying longer sequences. Finally, when FlexPlan concludes that no safe transition is feasible under the current switch headroom, it introspects on how much resource release is needed for feasibility, as another assistance to the operator.

We prototype FlexPlan [2] and show that it scales to real-world switch programs and sizable changes, and supports a rich set of safety properties including but going beyond those in existing work [53]. With FlexPlan, operaters can synthesize transition plans quickly and automatically (e.g., within a few minutes for sizable changes to switch.p4), while being assured of the correctness of the transition process.

2 Motivation

Analogues of our motivation can be found in existing work on OpenFlow network updates, summarized as follows: Network changes are a constant [23], but they often come with risks [36] due to intermediate states during transition [21, 45]. Rigorous approaches are needed to safeguard against transient disruption [44] to satisfy security requirements and stringent service-level objectives [16]. Ensuring transactional updates at each step [44], and formally guaranteeing the correctness of an update plan [39] is essential. These arguments hold true still, and are further amplified, for P4 programmable networks.

2.1 Runtime programmable switch updates

Programmable switches enable new network features to be quickly developed in-the-field [24, 27, 28, 35, 48, 56]; however, in earlier designs, *deploying* new features to the switch was an intrusive process. To update the switch program (e.g., add, remove, or modify a feature), the traditional approach was to completely recompile the changed program and reflash the data plane [60]. This results in device disruption, so program updates had to be conducted offline—user traffic is drained from the device and diverted elsewhere in the network, after which the switch is re-imaged, and finally, reactivated again.

Recognizing its cumbersomeness and risk for downtime, researchers and practitioners have made a concerted effort toward *runtime programmability* [4, 19, 52, 53, 57]. That is, switch programs are updated using partial reconfiguration without taking down the device for maintenance. Since P4 programs have modular table boundaries, runtime reconfigurations on specific match/action tables and their control flow logic need not disrupt other parts of the program. A feature update can be decomposed into a series of table and branch changes to transform a deployed program to a desired state.

Runtime programmability is not just an academic ideal. In response to the perennial call for both "feature velocity and cloud availability" [16], major switch vendors have embraced this trend with ASIC support. Nvidia's Spectrum switches [53] and Broadcom's Trident [9] and Jericho [3] switches are commercially available off-the-shelf, and academic prototypes [19, 52] are also exploring this design. Use cases of runtime programmable switches include real-time security defense [53], multi-tenancy [52], adaptive telemetry [19], where live switch updates afford higher flexibility not found in earlier programmable switches.

2.2 A motivating example

We will consider a simple example to illustrate the flexibility of runtime switch updates as well as the challenges they raise. The following program snippet uses `@add` and `@del` annotations to demarcate change boundaries in a control block:

```
1   /* Ex1: ipv4_ipv6 */
2   control ingress {
3       apply {
4           if (ipv4.isValid()) {
5               @del acl_v4.apply();
6               @add nat_acl_v4.apply();
7           } else if (ipv6.isValid()) {
8               @del acl_v6.apply();
9               @add nat_acl_v6.apply();
10          }
11          @add stats.apply();
12  }}
```

We remove two older versions of ACL tables for both IPv4 and IPv6 (Lines 5+8), and add two new tables that perform both NAT and ACL (Lines 6+9); we also add a statistics table for monitoring (Line 11). Since P4 is a target-independent language, operators can specify a desired change with such annotations without worrying about hardware details. Indeed, the mechanisms for implementing a live update depend upon the underlying switch platforms [19, 53]:

- FlexCore [53] relies on pointer swaps to achieve transactions, adding and removing a group of tables atomically. In our example above, we can first add `nat_acl_v4`, `nat_acl_v6`, and `stats` to scratch memory, and then use a transaction to make them visible to network traffic, while deleting the two older tables `acl_v4` and `acl_v6`.
- rP4 [19] also adds and removes MA tables leveraging scratch memory, but it relies on temporarily pausing traffic to achieve transactional effects. Before making the update, packets are paused and stored in a front buffer, and the respective tables are modified to effect the change. Buffered packets are then let out into the pipeline.

The careful reader may have noticed that this update appears to have completed within a single transaction, so no intermediate states are exposed. However, this is because we have yet to consider the resource constraints of the switch hardware. Switches have severe memory capacity bottlenecks, and they are easily packed to the brim with large MA tables [32, 53]. Thus, this logically simple update may be physically infeasible if the switch memory has high utilization.

The potentially infeasible operation is preparing the three new tables (`nat_acl_v4`, `nat_acl_v6`, and `stats`) in scratch memory before their old counterparts have been deleted. Assume without loss of generality that every MA table has the same size (say, of U, one unit of table entries), then the net resource increase after the change is only U. However, preparing the transaction causes a resource peak of $5 \times U$, which might exceed the available switch headroom. One workaround is to

ensure that the switch always has a low utilization [6], but this is obviously undesirable. Thus, recent work [53] proposes to break a larger update into multiple smaller batches to reduce the resource peak. For instance, if we first add `nat_acl_v4` and delete `acl_v4` in a transaction, it only requires $3 \times U$ headroom; the second transaction adds `nat_acl_v6` and deletes `acl_v6`, also within $3 \times U$ headroom; a final transaction adds the `stats` table, again within $3 \times U$ headroom. As tradeoff, after the first transaction, IPv4 traffic is processed with new tables for NAT and ACL, whereas IPv6 traffic is still processed with the old, and it only encounters new tables after the second transaction; further, the statistics table is only applied after the third transaction completes. Nevertheless, this may still be a reasonable sacrifice in order to achieve a feasible update, as long as the intermediate states are "well-behaved."

2.3 Computing a safe and feasible transition

It is far from clear, however, how to conjure up a transition plan for a desired change. rP4 [19] relies on the user to supply this plan, and FlexCore [53] algorithms only analyze whether changes are "reachable" to each other in the table graph. Neither represents a formal approach that can provide semantic guarantees on the transitional behavior. In general, the notion of correctness is scenario-specific and should be encoded in a user specification in a granular manner, going beyond the three fixed definitions in FlexCore [53], summarized below:

- *Program consistency:* Only one-step updates without intermediate states are allowed.
- *Element consistency:* Intermediate states are acceptable as long as reachable regions (e.g., changes that eventually reach the same table) are changed together atomically.
- *Execution consistency:* Reachable tables can be changed independently as long as no packets will mix them.

This cannot, for instance, capture user intention on "traffic classes" (e.g., IPv4 vs. IPv6); nor can it support more granular correctness definitions (e.g., for any intermediate state, packets must go through an ACL table, or packets must be sent to the same outgoing port). For some cases, the three fixed consistency definitions conflate into the same. For the above change, execution and element consistency will find the same plan as program consistency, because all changes eventually reach the `stats` table, forcing a $5 \times U$ peak despite the desire to relax the requirements for a feasible update.

2.4 FlexPlan: A program synthesis perspective

We believe that a principled solution should instead rest upon a firmer foundation, grounded in formal synthesis. Such a solution would satisfy three key goals:

- *Automated:* Beyond expressing a desired property, human reasoning is not required to identify a plan.
- *Completeness:* If a safe and feasible transition exists, we will guarantee to find it.
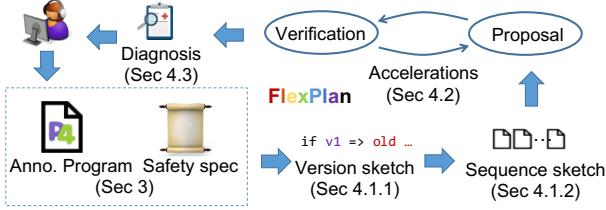
Figure 1: FlexPlan and its key techniques.

- *Soundness:* Once we output a transition, it is guaranteed to be safe and feasible.

The FlexPlan approach. We formulate this problem as follows. Given a P4 program p with a set of change annotations A, a safety specification ϕ, and resource headroom δ, identify a transition sequence $\{p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_t\}$, where p_i's are the intermediate program snapshots, and at the end of the sequence, all changes in A have been applied. Further, we require that a) each p_i satisfies ϕ, and b) each transition step $p_i \rightarrow p_{i+1}$ makes positive progress and stays within the resource headroom, initialized to δ. Our synthesis relies on a CEGIS approach (cf. [46, 47] for more background): In each iteration, the proposal phase generates a candidate transition sequence that satisfies ϕ and δ on the current set of counterexamples (that is, packets), which is initialized to the empty set. The verification phase will try to extract a new counterexample packet that causes violation, which will be consumed by the proposal phase in the next iteration, and so on. This continues until no counterexample packets can be generated—in which case we have a correct sequence—or until no candidate sequences can be generated—in which case no update plan exists. Figure 1 illustrates the workflow of FlexPlan, with several key milestones in Sections 3 and 4.

3 Specifying Safe Updates

Users provide a P4 program with annotated changes, as well as a specification to constrain intermediate states.

Update annotations. FlexPlan provides three intuitive annotation primitives for users to express a desired update: @add, @del, and @mod. A source P4 program can be annotated with these primitives, where each annotation site captures a set of changes. In Section 2.2, we have already provided an annotated program in this syntax, and here we used another example to describe several other aspects of the annotations.

```
1  /* Ex2: acl_ecmp_flowlet */
2  control ingress {
3      apply {
4          //modify acl into nat_acl
5          if (ipv4.isValid()) {
6              @mod acl.apply(), nat_acl.apply();
7          }
8          //add ECMP, delete flowlet switching
9          @del { if (ipv6.isValid()) flowlet.apply(); }
10         @add ecmp.apply();
11     }}
```

First, the syntax is similar to "P4 annotations" [7] in the P4 language standard. Our annotations target the common intersection of the reconfiguration primitives in existing work [19, 53]—an annotation may specify individual table updates (e.g., Line 10) or a code block update (e.g., tables with their control flow, as in Line 9). The hardware mechanisms are abstracted away from the annotations, but are assumed to provide atomicity for each change annotation, as in recent switches [19, 53]. The @mod primitive achieves similar effects as @del+@add, but @mod must complete in a single atomic step whereas the latter could occur in two separate steps. The entire update finishes when all annotations have been applied.

Specification language. Specifying resource constraints is as simple as providing a number δ that denotes the current switch headroom. Thus, our specification language focuses on the *consistency* properties, which constrain the relation between a program snapshot and the initial and final programs. FlexPlan refines the fixed consistency levels in existing work [53] in two ways: (1) a consistency property may refer to specific *traffic classes*, and (2) new consistency levels can be *programmatically* defined. Consider the following examples.

S1: Execution consistency for IPv4 traffic that hits ACL. Any IPv4 packet that hits the acl table must not mix old and new code blocks in any intermediate state. However, two IPv4 packets traversing different execution paths in the program do not need to use the same program version—e.g., TCP traffic may be processed by old code blocks, but UDP traffic by the new. We do not constrain the behaviors of other traffic classes.

```
1  specification {
2      // create new ghost variables for the program
3      // these are used for verification only
4      ghost bit<1> sawOld = false;
5      ghost bit<1> sawNew = false;
6      ghost bit<1> acl_hit = false;
7      // update ghost state when tables are applied
8      @old => { sawOld = true; }
9      @new => { sawNew = true; }
10     @hit('acl') => { acl_hit = true; }
11     // define: no path mixes old and new nodes
12     // $cur: the current/transitional program state
13     execution_consistency_ipv4 = {
14         $cur.in.ipv4.isValid() & $cur.eg.acl_hit =>
15             !($cur.eg.sawOld && $cur.eg.sawNew);
16     }
17     assert execution_consistency_ipv4;
18 }
```

Lines 4-6 define "ghost variables" that track meta-level properties of a program execution—specifically, whether a packet has encountered an old code block, a new block, or the ACL table, respectively. Lines 8-10 describe how ghost variables should be updated for an execution: whenever a packet triggers an old code block, a new block, or a table named acl, assign the respective ghost variable to be true. Lines 13-17 are the consistency assertion, where $cur represents a packet traversing the current program snapshot. If such a packet contains a valid IPv4 header when it arrives at the ingress, and

if it has hit the ACL table before it exits the egress, then we assert that it should not be processed by a mix of code blocks.

S2: Field consistency for egress_spec. We define a new consistency level that only constrains the processing outcomes of specific header fields. The following example specifies that any intermediate states should preserve the processing outcome for the packet's egress port—i.e., a packet should either go to the same port as the old program or as the new one. The `$new` and `$old` variables denote two packets traversing the old and new programs, respectively.

```
1  specification {
2    // preserve processing outcome of egress_spec
3    field_consistency_espec = {
4      $cur.in == $old.in == $new.in =>
5        ($cur.eg.espec == $new.eg.espec ||
6          $cur.eg.espec == $old.eg.espec);
7    }
8    assert field_consistency_espec;
9  }
```

S3: Program consistency for TCP traffic: We require that TCP packets must not encounter any intermediate state. The `all_old` assertion states that if at the ingress a packet carries a valid TCP header, then at the egress it must only have been processed by old tables. Analogously, the `all_new` assertion states the opposite. Their disjunction implies that TCP traffic will only be processed by one version of the program. The primary difference between this specification and S1 is that execution consistency only constrains the behaviors of each individual packet, whereas program consistency constrains the behaviors across all packets of a certain kind (e.g., TCP).

```
1   specification {
2     // same ghost variables as before
3     ghost bit<1> sawOld = false;
4     ghost bit<1> sawNew = false;
5     @old => { sawOld = true; }
6     @new => { sawNew = true; }
7     // define whether all packets use the old program
8     all_old = {
9       $cur.in.tcp.isValid => !$cur.eg.sawNew;
10    }
11    // define whether all packets use the new program
12    all_new = {
13      $cur.in.tcp.isValid => !$cur.eg.sawOld;
14    }
15    // all packets use old program or all use new
16    assert all_old || all_new;
17  }
```

These granular consistency levels require program semantic analysis and cannot be captured by fixed definitions [53]. We describe several more examples in Appendix 9.1 and summarize them in Table 1. Figure 2 presents the grammar of the specification language. Like existing work [18, 50, 51], the specifications are eventually translated into assertions in the source P4 program. Appendix 9.2 shows one such translation.

spec	::=	**specification**{*stmt*∗}	*Specification*
stmt	::=	*gvar*∗	*Ghost vars*
	|	*instr*∗	*Instrumentation*
	|	*property*∗	*Property*
	|	*assert*∗	*Assertion*
gvar	::=	**ghost bit** $< n > gv$	*Ghost vars*
gexpr	::=	**$cur** \| **$old** | **$new**	*Network version*
	|	*gexpr.field*	*Field dereference*
	|	*gexpr* + *gexpr*	*Addition*
	|	...	*Other expr*
instr	::=	*label* => *assignment*∗	*Ghost update*
assignment	::=	*gv* = *gexpr*	*Assignment*
label	::=	**@new** | **@old** | **@hit**	*Annotation*
property	::=	*name* = {*gexpr*∗}	*Consistency*
assert	::=	**assert** *name*	*Assertion*

Figure 2: FlexPlan specification language grammar.

Specifications	LoC
S1. Execution consistency for IPv4 [53]	13
S2. Field consistency for egress_spec	8
S3. Program consistency for TCP [53]	13
S4. Element consistency for ACL [53]	15
S5. Table consistency for ECMP [53]	10
S6. VLAN table access [51]	8
S7. Correct TTL decrement [51]	6

Table 1: FlexPlan supports granular consistency specifications that go beyond existing work [53] (e.g., S1-S5). Although our primary focus is consistency, FlexPlan can also support general program snapshot correctness (e.g., S6-S7) guarantees addressed by existing verification work [18, 51].

4 Update Plan Synthesis

Next, we describe how FlexPlan synthesizes an update plan from the annotated program and specification. We denote these inputs as $\langle p_{[A]}, \phi, \delta \rangle$, where $p_{[A]}$ is an annotated P4 program with a set of change sites $A = \{a_1, \cdots, a_k\}$, and ϕ and δ are the safety and resource constraints, respectively. FlexPlan outputs an update sequence $s = \{p_{old} = p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_t = p_{new}\}$, where two special states p_{old} and p_{new} represent the initial and final programs, respectively. We ask that each intermediate state p_i must be safe (i.e., satisfying ϕ) and each transition from $p_i \rightarrow p_{i+1}$ is feasible (i.e., the resource spike for this transition stays within δ_i, as computed from the initial headroom δ after applying transitions before p_i). If no such s can be found, FlexPlan outputs diagnostic information on whether the safety or feasibility constraints have caused the failure, and in the latter case, it analyzes how much resource release would enable a feasible synthesis.

4.1 Synthesizing a program sequence

Our CEGIS formulation uses *program sketching* [47], a classic framework for synthesis. This formulation views the synthesis task as filling "holes" in an incomplete program (i.e., a "sketch"), and it has been successfully applied to network

```
1   control ingress {
2       apply {
3           /* annotation site 1: acl->nat_acl */
4           if(ipv4.isValid()) {
5               if (! vsk.v1) { // @mod
6                   acl.apply();
7               } else {
8                   nat_acl.apply();
9               }
10          }
11          /* annotation site 2: delete flowlet */
12          if(! vsk.v2) {        // @del
13              if (ipv6.isValid())
14                  flowlet.apply();
15          }
16          /* annotation site 3: add ecmp */
17          if(vsk.v3) {     // @add
18              ecmp.apply();
19          }
20  }}
```

Figure 3: A version sketch that is derived from the `acl_ecmp_flowlet` example in Section 3. Version variables `vsk.v` are the sketch holes. The safety specification will also be instrumented into the version sketch (cf. Appendix 9.2)

programs [13, 22]. However, the objective in FlexPlan differs from existing work as it must identify a correct *sequence* of program snapshots that successively build upon each other toward the final program. To cast this into the CEGIS framework, we develop two novel encodings: a *version sketch* to represent any valid program snapshot during transition, and a *sequence sketch* to string together multiple version sketches and constrain them to make progress toward our final state.

4.1.1 Version sketch: Encoding program snapshots

The version sketch is a uniform and expressive encoding that can capture any transitional program snapshots derived from $p_{[A]}$. As its name suggests, it introduces a "version variable" v_i at the annotation site $a_i \in A$. The annotation is then substituted with two version control branches guarded by v_i. Without loss of generality, consider $a_i = \{@mod\ s_i \rightarrow t_i\}$, a modification from s_i to t_i. It will be transformed into if (v_i) then $\{t_i\}$ else $\{s_i\}$, or simply `ite`(v_i, t_i, s_i). That is, a version sketch with v_i turned on encodes a program snapshot where annotation a_i has been applied; on the other hand, a version sketch with v_i turned off represents a snapshot where the change a_i is yet to be applied. The `@add` and `@del` annotations are handled analogously, with one of the branches controlling an empty statement: `ite`$(v_i, t_i, noop)$ for adding t_i and `ite`$(v_i, noop, s_i)$ for deleting s_i. Across all annotation sites, by turning version variables on or off, the resulting snapshot seamlessly reflects any combination of applied changes.

Figure 3 shows the version sketch for `acl_ecmp_flowlet`, where version variables are added to the input program as instrumentations. From this instrumented program, FlexPlan derives an SMT formula, where ξ represents the unchanged components without annotations, and the i-th `ite` formula

represents annotation a_i. Constraining the version sketch with the safety specification would give an SMT encoding:

$$\mathrm{vsk}\,(\xi, \bigwedge_{i=0}^{k} \mathrm{ite}(v_i, t_i, s_i)) \ \wedge\ \phi$$

FlexPlan obtains SMT formulas in a similar way as existing work [18]. ϕ is instrumented into the version sketch. Then, FlexPlan converts all statements into static single assignment form and SMT formulas. It then computes the weakest preconditions based on the control flow logic.

The version sketch is expressive enough to encode any intermediate snapshot and constrain its safety with ϕ. However, it cannot capture the resource constraint δ, because it is a *sequence* property defined over a series of snapshots and their relations. As a version sketch only represents an individual snapshot, it cannot easily reason about end-to-end feasibility for a snapshot sequence. By itself, it only enables an awkward workaround—start with an empty version sketch, synthesize a safe snapshot as its immediate next step, and iterate based on the new snapshot. More concretely, one could ask that the next snapshot must fill more holes than the current one, while staying safe with respect to ϕ and within the current headroom δ. This would result in a new snapshot where a subset (but likely not all) of version variables have been turned on. This serves as the new "initial" state for another synthesis, until the final state has been reached. However, this results in a difficult search process, as it can only be guided with some greedy heuristics—e.g., maximizing the progress for each step by filling as many holes as possible, or minimizing the resource spike for each step while ensuring some progress. Either way, the lack of a global view could corner the search into a difficult or infeasible state (e.g., no more headroom), where it must backtrack and probe again in the very large search space—all possible permutations of change annotations, together with all possible combinations of adjacent changes in each permutation. This greedy synthesis also cannot easily conclude that no feasible solution exists.

4.1.2 Zooming in on resource constraints

Thus, resource constraints must be encoded explicitly to enable a cross-snapshot, end-to-end synthesis. Recall that δ denotes the initial switch headroom, which is obtained by subtracting the total table sizes of p_{old} from the overall switch memory. From there, each transition from one snapshot `vsk` to the next `vsk`$'$ adds and removes some tables—thus, we must keep track of the changes to δ and two metrics "spike" and "release" at each transition. Consider the version variable v_1 in Figure 3, which modifies an `acl` table (2Mb) into a `nat_acl` table (3Mb). To achieve atomicity, this is done by first adding `nat_acl` in switch memory, resulting in a transient resource spike of 3Mb, and then deleting `acl` and freeing 2Mb in the same transaction. We record this as $\mathrm{add}_1 = 3$ and $\mathrm{del}_1 = 2$ for turning on v_1. Thus, across all v_i we define the *spike*:

$$\mathrm{vsk}'.\mathrm{spike} = \sum_i \mathrm{ite}(\mathrm{vsk}'.v_i \wedge \neg \mathrm{vsk}.v_i,\ \mathrm{add}_i,\ 0)$$

That is, if the transition from vsk to vsk$'$ has turned on v_i, then the transient resource spike increases by add_i; otherwise, if v_i is not changed in this step, it does not contribute to the spike. We can therefore define the resource *release* after the transaction completes and the spike comes down:

$$\texttt{vsk}'.\texttt{rel} = \sum_i \texttt{ite}(\texttt{vsk}'.v_i \wedge \neg\texttt{vsk}.v_i, \texttt{del}_i - \texttt{add}_i, 0)$$

In our example, the net release is $\texttt{del}_1\texttt{-add}_1 = -1$. This reflects in the overall headroom update to δ after this step:

$$\texttt{vsk}'.\texttt{headroom} = \texttt{vsk}.\texttt{headroom} + \texttt{vsk}'.\texttt{rel}$$

The feasibility constraint can therefore be stated as $\texttt{vsk}.\texttt{headroom} \geq \texttt{vsk}'.\texttt{spike}$ for any two adjacent snapshots, across the entire transition sequence from p_{old} to p_{new}.

4.1.3 Sequence sketch: Encoding a transition plan

Based on the above resource constraint analysis, we develop a sequence sketch encoding that enables an end-to-end CEGIS. A sequence sketch ssk is the conjunction of t version sketches $\{vsk_1, vsk_2, \cdots, vsk_t\}$ as well as their sequential relations to each other. $\texttt{vsk}_1$ and $\texttt{vsk}_t$ encode the initial and final programs, respectively, and other states represent the transitions.

$\forall i \ \neg\texttt{vsk}_1.v_i$; initial program p_{old}

$\forall i \ \texttt{vsk}_t.v_i$; final program p_{new}

$\forall j \ (\forall i \ \texttt{vsk}_{j-1}.v_i \implies \texttt{vsk}_j.v_i)$; progress

$\forall j \ \texttt{SpikeAndHeadroom}(\texttt{vsk}_{j-1}, \texttt{vsk}_j)$; feasibility

That is, all version variables in the initial sketch are turned off, equating it to p_{old}; all variables in the final sketch are turned on, resulting in p_{new}. A later sketch in the sequence must monotonically advance the version variables to make progress toward the final state. Furthermore, each transition's spike must be feasible within its current headroom (Section 4.1.2).

Sequence synthesis. This encoding enables an end-to-end CEGIS by filling the ssk holes (i.e., all v variables in all version sketches vsk) in a way that satisfies ϕ for all snapshots and δ across all adjacent snapshots. The sketch holes are $\texttt{H} \in \mathbb{B}^{k \times t}$, a two-dimensional matrix of binary variables.

$$\texttt{H}_{k \times t} = \begin{bmatrix} vsk_1.v_1 & vsk_1.v_2 & \cdots & vsk_1.v_k \\ vsk_2.v_1 & vsk_2.v_2 & \cdots & vsk_2.v_k \\ \vdots & \vdots & \vdots & \vdots \\ vsk_t.v_1 & vsk_t.v_2 & \cdots & vsk_t.v_k \end{bmatrix}$$

In this matrix, k is the number of annotations, and t is the number of transitional states; and we will synthesize all holes in an end-to-end CEGIS, as shown in Figure 4. The proposal phase (Lines 7-11) identifies a potentially correct program (i.e., values in H) by solving for H that exhibits correct behaviors on all counterexamples collected so far (Line 9). The verification phase strengthens the check to test against the full specification (Lines 12-17). If no further violations are

```
1:  function SEQUENCECEGIS(ssk, φ, δ)
2:      for t = 1..k do //Iteratively increase seq length ◁ Opt-Diag
3:          ssk.H ← RAND(𝔹ᵏˣᵗ) //Init w/ random H
4:          ce_set ← ∅   //No counterexample so far
5:          // Next, enter main CEGIS loop          ◁ Opt-SnapL
6:          while ¬ Timeout do
7:              // Proposal: Identify candidate H=h
8:              ssk.H ← SYMBOLIC(𝕏ᵏˣᵗ)
9:              {ssk.H := h} ← SMTSOLVE(ssk, φ, δ, ce_set)
10:             if ssk.H == ∅ then    // No solution exists, t++
11:                 break
12:             // Verification: Verify H=h
13:             ce ← SMTVERIFY(ssk.H, φ, δ)          ◁ Opt-SnapV
14:             if ce ≠ ∅ then // Obtain counterexample
15:                 ce_set ← ce_set ∪ {ce}
16:             else  // Verifies, solution found!
17:                 return ssk.H
```

Figure 4: The end-to-end CEGIS algorithm on the sequence sketch. Later subsections will further develop three optimization techniques, labeled as 'Opt-', to scale this analysis.

found, we have obtained a correct solution; otherwise, counterexamples are added to ce_set and we continue with a new proposal. The power of CEGIS lies in the fact that with more counterexamples, SMT solvers learn from violations and eliminate entire classes of proposals in the search. Notice also that at Line 2, we iteratively deepen the search based on the sequence sketch length, so it does not need to reason about a larger problem instance unless absolutely necessary.

In the acl_ecmp_flowlet example (Figure 3), suppose that we require program consistency for IPv4 and that the current headroom is 1Mb. A correct ssk could give a two-step transition denoted by:

$$\texttt{H}_{3 \times 3} = \begin{bmatrix} \texttt{F} & \texttt{F} & \texttt{F} \\ \texttt{F} & \texttt{T} & \texttt{F} \\ \texttt{T} & \texttt{T} & \texttt{T} \end{bmatrix}$$

The first transition deletes flowlet (3Mb) for IPv6 traffic. This causes a resource spike of 0Mb and a release of 3Mb; and the headroom becomes 4Mb after this transaction. Next, the second transition modifies acl (2Mb) into nat_acl (3Mb) and adds ecmp (1Mb) for IPv4 traffic, which causes a resource spike of 4Mb and a release of -2Mb.

4.2 Accelerating the CEGIS loop

We have now obtained a sequence CEGIS algorithm that is guaranteed to be sound (i.e., a synthesized transition is correct) and complete (i.e., if a correct transition exists, it will be found); it also produces the shortest transition due to the iterative deepening search. (More discussions in Section 4.4) However, in terms of performance, this algorithm has a series of scalability bottlenecks. A traditional CEGIS problem only has to reason about SMT formulas generated from a single program, but FlexPlan produces formulas many times larger as they are derived from program sequences. Thus, we develop

two domain-specific optimizations to accelerate the proposal and verification phases, respectively, based upon a divide-and-conquer approach. Our observation is that, for specific CEGIS steps, we can avoid reasoning about ssk directly but instead reason about its comprising vsk instances individually. Since SMT algorithms tend to grow exponentially with the formula size, dividing a large instance (i.e., ssk) into many smaller ones (i.e., vsk) and reasoning about the smaller formulas individually is more efficient than reasoning about the larger instance in a single shot.

Snapshot learning and generalization. We extract insights from a single snapshot before entering the main CEGIS loop (Line 5, 'Opt-SnapL'; Figure 4). This algorithm learns what a "bad" snapshot might look like, and then generalizes the knowledge for the entire sequence. We observe that, if a snapshot vsk violates the safety property ϕ, then no matter where this snapshot appears in the transition sequence ssk, it still constitute a violation. Thus, there is much to learn from an individual vsk before we have to stitch many such snapshots together. Stated in another way, resource constraints δ force us to perform end-to-end reasoning in general, but the safety aspect of the reasoning is still decomposable to individual snapshots.

This is achieved by operating a loop that extracts as many unsat cores as possible (within timeout threshold) from a single snapshot vsk, but only asserting safety properties ϕ and ignoring resource concerns δ. Each iteration produces one counterexample that witnesses a specific violation for *any* snapshot in the sequence. For instance, a counterexample might say that {v1(T), v2(T), v3(F)} violates IPv4 execution consistency, so this snapshot should never appear anywhere in the sequence. After producing many counterexamples, we aggregate and feed such knowledge into the main CEGIS loop, so that the proposal phase will not err in the same way on the larger sequence sketches. Further, for efficiency, FlexPlan distills counterexamples into "minimum unsatisfiable cores" (unsat cores) [20], which is a subset of assignments to vsk.v as the root cause. In our running example, the unsat core {v1(T), v3(F)} articulates the essence of the violation—the two IPv4 related blocks are not updated together. The main CEGIS loop ingests this condensed knowledge, avoiding the larger formulas from full-blown counterexamples.

Snapshot verification. The 'Opt-SnapV' optimization reduces the task of verifying a proposed ssk against ϕ into smaller tasks of verifying each individual vsk in it. The intuition still stems from the fact that ϕ can be reasoned per snapshot, whereas δ is a sequential property and needs to be synthesized end-to-end. The proposal phase (Line 9; Figure 4) must already ensure end-to-end feasibility in its proposal; so a subsequent verification may only fail due to violation of ϕ. Thus, when verifying ssk, we check individual vsk snapshots separately. If any snapshot produces a violation, its counterexample is used in the next round of synthesis.

Figure 5 shows the pseudocode for both optimizations.

```
1:  function SNAPSHOTLEARN(vsk, φ)
2:      uc_set ← ∅ // Aim to learn unsat cores from vsk
3:      while ¬ Timeout do
4:          // Solve for a new violation by negating φ
5:          {vsk.v, pkt} ← SYMBOLIC(𝔹ᵏ, Packet)
6:          {vsk.v := v, pkt := p} ← SMTSOLVE(vsk, uc_set, ¬φ)
7:          if {v, p} == ∅ then // Exhausted all ce's
8:              return uc_set
9:          else // New violation, extract unsat core
10:             uc_set ← uc_set ∪ EXTRACTUC(v, p, φ)
11:     return uc_set
12: function SNAPSHOTVERIFY(ssk, φ) // Verify a proposed ssk.
13:     ce_set ← ∅ // Counterexample set
14:     for vsk ∈ ssk do
15:         ce ← SMTVERIFY(vsk, φ)
16:         if ce ≠ ∅ then
17:             ce_set ← ce_set ∪ {ce}
18:             return false
19:     return true // All snapshots verify!
```

Figure 5: The snapshot learning (Opt-SnapL) and snapshot verification (Opt-SnapV) algorithms. Note that the snapshot learning algorithm does not need to enumerate all counterexamples or unsat cores, as it serves as an optimization for the main CEGIS loop. If the learning times out (Line 3), the collected uc_set is still useful in the main CEGIS.

4.3 Diagnosing the synthesis

Another domain-specific property of our synthesis lies in its *diagnosability*. In traditional synthesis, even if the tool struggles to find a solution, it may not mean that a valid solution does not exist—unless it has exhausted the search space. Thus, the search in the worst-case scenario may spend a significant amount of time only to conclude at the end with a failure. In contrast, we observe that FlexPlan can obtain three types of conclusive proof early in the game. This helps us to determine whether or not a continued search will be fruitful, and enables further optimizations. We call these techniques *introspection*.

Existence? A basic type of introspection is to determine whether or not a safe transition exists at all, regardless of the resource headroom. We observe that FlexPlan can determine this by checking p_{old} and p_{new} against the safety specification ϕ. If both programs are correct, then some safe transition must exist—the degenerate case is to make a one-step transition {$p_{old} \rightarrow p_{new}$}, exposing no intermediate state (but potentially causing a very large resource spike and thus may not be feasible). However, if even this check fails, FlexPlan aborts with the conclusion that no solution exists.

Deepening the search? Once we pass this smell test, we are faced with a harder introspection task—what should be the upperbound of t, the length of the sequence sketch? The algorithm in Figure 4 uses a naïve upperbound, where t iteratively deepens from one to k, the total number of change annotations. It first searches through all possible t-step transitions; if no such transition is both feasible and safe, it attempts

a longer transition sequence with $t+1$ steps. Failures are only conclusive at $t = k$, where each transition only applies one change thus no further breakdown is possible. However, this deepening search gets significantly more expensive with every increment to t—at each t, we need to perform a full round of CEGIS with a sequence of t sketches. Thus, it is beneficial to reflect on the usefulness of a larger t before we deepen the search, potentially stopping far earlier than the naïve bound. This introspection relies on the following property:

Introspection theorem. *Assume that the switch has infinite resources. Without resource constraints, if there does not exist a t-step safe transition plan, then there cannot exist any safe transition plan with more than t steps.*

Proof sketch: The intuition is that, if a t-step transition exists that satisfies ϕ but not δ, we can potentially make more granular changes in a longer transition sequence to stay within δ; thus, attempting a $(t+1)$-step transition could be fruitful. On the other hand, if a t-step transition that satisfies ϕ does not exist to begin with, then any t'-step transition where $t' > t$ cannot exist either. We can prove this by contradiction: if a t'-step transition exists, repeatedly combine any adjacent transitions into a larger transition until it becomes a t-step transition. This resulting transition must satisfy ϕ as it exposes strictly a subset of the states in the t'-step transition. $\square$

Thus, when FlexPlan concludes that no t-step transitions exist, before it attempts a $(t+1)$-step CEGIS, it performs the above introspection. The introspection may conclude that a) some safe solution exists but b) no safe solution is feasible under the current resource constraints. This may be disappointing, but all is not lost—runtime programmable switches make it possible to deallocate resources to make extra room (e.g., by deleting certain tables or table entries).

Resource release? Whether to deallocate resources and which tables to delete are up to the network operator, but FlexPlan performs a third introspection to diagnose *how much* resource release would be sufficient for a t-step transition (where the search has stopped). This relies on an SMT optimization primitive `max-smt`, which can maximize an objective function while solving for a solution. Recall that each step causes a resource spike during the transition, and a headroom change after it. We track the the minimum headroom across a t-step transition, and ask for a solution that maximizes it:

$$\texttt{min\_headroom} = \min_{\forall j} \texttt{vsk}_j.\texttt{headroom}$$

$$\delta^* = \texttt{max-smt(min\_headroom) s.t. ssk} \wedge \phi$$

δ^* will be the smallest headroom possible to maneuver a t-step transition, and $\delta^* - \delta$ is the amount of resource release that is required to achieve a feasible update.

4.4 Remarks

We discuss several properties of the synthesis techniques.

Introspection. An important property of the introspection algorithms is that they work with sequence sketches of the current length of the search (i.e., t steps). Thus, they do not lead to new scalability bottlenecks. Furthermore, the combination of the second and third introspection techniques also enables a synthesis goal of identifying a safe transition plan with minimized resource spikes—first determine the sequence length upperbound for a safe transition (with the second introspection), and then synthesize a transition while minimizing resource spikes at this length (with the third introspection).

Guarantees. The completeness of the synthesis is derived from the fact that the candidate solution space is finite and that CEGIS will eventually finish an exhaustive search [26]. Concretely, the solution space is defined by the two-dimensional matrix $\texttt{H}_{k \times t}$. k is the number of annotations, and therefore finite. t is the sequence length, initially undetermined, but we know that it is upperbounded by k, because applying one annotation per step will result in the longest possible sequence. This is because we ask that each transition step makes positive progress, so no reverts are allowed once a change has been made. The synthesis is also sound, because FlexPlan always verifies the correctness of a proposed candidate plan.

5 Discussions and Limitations

P4 intermediate states. Intermediate states when the data plane is under change have been considered in the P4Runtime standard (cf. DATAPLANEATOMICS) [6]. However, the current standard focuses on the atomicity and intermediate states when adding or removing a batch of *table entries* for existing MA tables. Runtime table additions and removals, as a recent development, have not yet been captured in P4Runtime. Nevertheless, for table entry changes, P4Runtime describes how atomic pointer swaps can be used for transactional changes (when available in the target), and discusses the headroom requirement for preparing the changes in scratch area. This results in a similar range of considerations as recent designs for runtime programmable switches [19, 53]. We hope that FlexPlan will further the research in handling date plane intermediate states and the standardization process in P4Runtime.

Change annotation primitives. In the spirit of target-independence, our change annotations capture the intersection of hardware reconfiguration primitives between FlexCore [53] and rP4 [19]. Reconfiguration primitives that are not yet fully supported across platforms (e.g., parser changes [19] and table swap operations [53]) are considered out of scope for the current paper. These are interesting avenues for future work.

Switch architectures. Recent designs of runtime programmable switches [19, 53] employ disaggregation to split memory from compute. Thus, FlexPlan models memory resources as a global constraint—e.g., when a table is removed, the released resources can be used anywhere. However, future runtime programmable switches might adopt alternative architectures—e.g., RMT switches [12] with fixed stage boundaries would require a different model on memory reusability. Similarly, for SmartNIC targets with software and hardware pipelines [1], atomic transactions may become

Programs	LoC	Tables	Synthesis results		Programs (α)	Specification	Headroom	# Steps	Time(s)	Greedy
			time(s)	c.e.(u.c.)						
flowlet	216	6	5.61	0(0)	switch (20%)	IPv4 exec. consistency	80%	2 (✓)	110.32	132.37
simple_nat	362	6	6.02	1(1)	switch (20%)	IPv4 exec. consistency	50%	3 (✓)	160.25	timeout
ndp	275	7	5.73	1(1)	switch (20%)	IPv4 exec. consistency	20%	5 (×)	318.95	timeout
beamer	448	7	6.79	2(2)	switch (40%)	IPv4 exec. consistency	50%	3 (✓)	197.53	timeout
vpc	272	10	6.48	2(2)	switch (40%)	Espec field consistency	20%	4 (✓)	263.38	timeout
sai_p4	697	14	7.18	2(2)	switch (40%)	L2/L3 field consistency	20%	4 (✓)	436.44	timeout
linear_road	846	24	13.48	4(4)	switch+meter-stat	L2/L3 field consistency	20%	2 (✓)	186.24	552.82
nethcf	822	30	14.71	6(6)	switch+meter-stat	IPv4 exec. consistency	20%	2 (×)	93.90	105.51
netcache	1845	96	37.59	14(14)	switch+ipv4-ipv6	L2/L3 field consistency	20%	2 (✓)	249.99	749.78
switch	5599	120	199.43	27(24)	switch+ipv4-ipv6	IPv4 exec. consistency	20%	2 (✓)	102.35	124.46

Table 2: FlexPlan scales to real-world programs. The left-hand side uses a range of popular programs, ranked by the number of MA tables they contain. It uses an update ratio of 20% with 50% resource headroom. The right-hand side focuses on case studies with switch.p4, including synthetic and realistic changes. For each change, we denote resource peak needed to atomically update the entire program in a single step as S, and set the headroom to $\beta \times S$ ($\beta = 20\%, 50\%, 80\%$). The greedy synthesis times out in 5 out of 10 cases, and it is much slower than FlexPlan for the rest of the cases.

harder due to the need to synchronize across pipelines. Thus, as alternative architectures become available in the future, FlexPlan may need to incorporate a new set of constraints.

Safety properties. The FlexPlan consistency specifications capture safety (but not liveness) properties that are defined over multiple program executions, or k-safety hyperproperties [15,49]—specifically, 3-safety, as FlexPlan reasons about the old, new, and current snapshots. Among safety properties, it does not analyze stateful packet processing, where program behaviors may mutate based on the input packets [30]. Further, FlexPlan only reasons about a single network device, so extending it for network-wide updates is future work.

Resource utilization. FlexPlan considers switch memory constraints as the main bottleneck resource. In a P4 program, each MA table has a 'size' field that specifies the maximum number of entries it could contain. This provides coarse-grained information as input to FlexPlan. However, the number of entries in a table is not the same as physical memory consumption, which further depends on the match types and their target-specific implementation (e.g., TCAM vs. SRAM). To obtain exact information, FlexPlan would need the compiler to produce such data for the old and new programs. Modeling other types of resource constraints is left to future work.

Synthesis delay. Requiring each runtime update to go through a formal synthesis phase will incur delay to the change. As we will show, the latency is a few minutes across our evaluation. We believe that reliability gains outweigh the resulting delay.

6 Evaluation

Prototype. We have built FlexPlan in ∼5000 lines of code (available at [2]). Our prototype consists of two components: (1) a frontend translator building upon an existing tool [18], which takes in the annotated P4 program and the specification, and converts them into an instrumented sequence sketch and SMT formulas; and (2) the main CEGIS backend which searches for a transition and performs introspection whenever needed. We use Z3 [8] as the SMT solver.

Methodology. Our program corpus is based upon popular

programmable switch applications, representing use cases in monitoring, security, offloading—similar as recent P4 verification projects [18, 38, 50]. It contains real-world P4 programs, with sizes ranging from 200+ to 5000+ LoC. Further, FlexPlan uses two methods to generate program changes. Synthetic changes are generated using a similar strategy as existing work [53], which controls the number of changes with a parameter α. If a program has M tables, an update ratio α will generate $M \times \alpha$ table additions, deletions, or modifications. To test realistic changes, we use switch.p4 as the basis to perform manual modifications based on its control block boundaries (e.g., remove or add back the egress statistics control block `process_egress_bd_stats()`), mimicking feature changes in realistic deployments. To analyze headroom, we assume that each table has the same size, denoted as U.

Evaluation objectives. Our evaluation primarily focuses on various dimensions of scalability: (1) How well does FlexPlan scale to real-world programs and sizable changes? (2) How well do the granular consistency levels work? and (3) How effective are the FlexPlan optimization and introspection techniques? We note that existing P4 verification projects [18, 38] analyze the correctness of single program snapshots, so they are not suitable as baseline solutions for comparison. Thus, we have created several FlexPlan variants as baseline solutions, where specific optimization techniques are disabled.

6.1 Macrobenchmarks

We start with the macrobenchmarks summarized in Table 2.

Scalability. The first macrobenchmark (left four columns) tests the scalability of FlexPlan with popular switch programs. We use "L2/L3 field consistency" (i.e., intermediate snapshots should preserve the same L2/L3 processing outcome) with a fixed update ratio of 20%. Further, we set the headroom to 50% of what would be required for the most straightforward plan that updates the entire program in a single step, which in turn would lead to the highest possible resource peak. There are two high-level takeaways. First, as the program size increases from 200+ to 5000+ LoC, synthesis time also grows

significantly, as larger programs produce bigger SMT formulas. However, even for the largest program, FlexPlan was able to finish within 3.4 minutes. Second, the number of learned counterexamples ('ce') also grows with the program size, and FlexPlan effectively learned from a relatively few number of counterexamples (varying from 0 for the smallest programs to 27 for the largest) before finding a correct transition. We further break it down by showing the number of counterexamples learned from a single snapshot (i.e., in the process of enumerating unsat cores with SnapL, labeled as 'uc'). Except for switch.p4, FlexPlan enumerated all unsat cores in the SnapL phase, so the main CEGIS loop identified a correct solution in the first attempt. For switch.p4, FlexPlan performs 3 CEGIS iterations in the main loop to identify the transition.

Synthesized switch.p4 changes. The second macrobenchmark (right-hand side) zooms in on modifications to switch.p4, a datacenter switch implementation. We first test six synthesized changes with different control parameters—resource headrooms (rows 1-3), update ratios (rows 2+4), specification types (rows 5-6)—and examine their influence on the transition sequence lengths and synthesis time. As we can see, more severe headroom constraints lead to longer synthesis time (rows 1-3). FlexPlan had to try longer transition sequences to find a solution or conclude that no solution exists (shown as ×). Larger update ratios also lead to longer synthesis time (e.g., 23% increase when changing from $\alpha = 20\%$ to $\alpha = 40\%$ in rows 2 and 4). Moreover, the specification type also has a direct influence on synthesis. Field consistency on egress_spec is easier to check compared to L2/L3 header checks. FlexPlan took 1.8-7.3 minutes across all cases.

Hand-crafted switch.p4 changes. Next, we analyze a set of hand-crafted changes to switch.p4, by adding, removing, or modifying well-defined control blocks. We also vary the specifications across data points. The switch+meter-stat case removes all statistics tables (5 tables overall) in switch.p4, and adds in meter related tables (4 tables). For L2/L3 field consistency, FlexPlan identifies an update plan with 2 transitions. A closer look at the update plan shows that FlexPlan first removes all statistics tables, then adds meter tables—this is possible because statistics tables do not manipulate L2/L3 packet headers. On the other hand, IPv4 execution consistency fails to generate a update plan because most of the modified tables share some execution paths, which means they must be updated together. The required headroom goes beyond the available resources (20%). The switch+ipv4-ipv6 update removes IPv6 processing tables (10 tables in total) and adds IPv4 processing tables (8 tables). IPv4 execution consistency, on the other hand, could find a update plan with 20% headroom. This is because IPv4 and IPv6 tables can be updated separately as they do not share execution paths. All experiments with realistic changes finished within 4.2 minutes.

FlexPlan vs. greedy synthesis. We also test the greedy synthesis algorithm (Section 4.1.1), which either maximizes progress ('Greedy MinSeq') or minimizes resource spikes
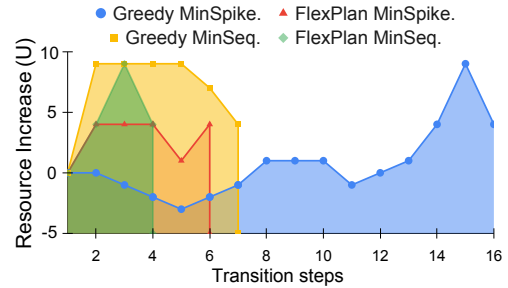

Figure 6: A NetCache case study showing step-by-step transitions and the headroom changes. The greedy synthesis produces suboptimal transitions whtn it is able to finish; it times out under more severe resource constraints.

('Greedy MinSpike') for each step locally. As Table 2 shows, it times out after 30 minutes for five out of ten cases. When it is able to produce a transition, it only finds suboptimal solutions. Figure 6 visualizes a case study on NetCache, where greedily maximizing per-step progress results in a transition in seven steps with 80% headroom, whereas FlexPlan produces a much shorter transition in four steps ('FlexPlan MinSeq'). Similarly, FlexPlan when minimizing resource usage (cf. Section 4.4) leads to much lower peak usage than greedily minimizing resource spikes per step. This demonstrates the benefits of the sequence sketch encoding for end-to-end synthesis.

6.2 Consistency levels vs. headroom

Next, we show that the granular consistency specifications in FlexPlan can lead to lower resource requirements when rolling out an update. We use "program consistency" and "execution consistency" as baselines, which are the strongest and weakest guarantees developed in existing work, respectively [53]. We chose three large switch programs (NetHCF, NetCache, switch.p4), and generated 50 changes with α ranging from 5% to 40%. For each change, we ask FlexPlan to find the transition sequence that minimizes the peak resource usage under each specification. We then averaged across all changes with the same α and show the results in Figure 7.

Our first takeaway is the inflexibility of heuristic-based definitions in FlexCore [53]. Although "execution consistency" is a weaker requirement than "program consistency," it unfortunately does not reduce the resource peak by much and the two corresponding curves are closely coupled together (i.e., 'FC-Prog' vs. 'FC-Exec'). It can reduce the resource peaks for specific cases, but aggregating over all cases the reduction is only 5%. We found that this is highly correlated to the program shapes and where the changes are made. A common root cause can be attributed to the *bottleneck table* problem. If a change modifies some tables that are shared by many or all execution paths, then it forces all changes to be made in the same step, equating "execution consistency" to "program consistency." Since FlexCore [53] only relies on "reachability" information at table level and does not ana-

(a) NetHCF.p4
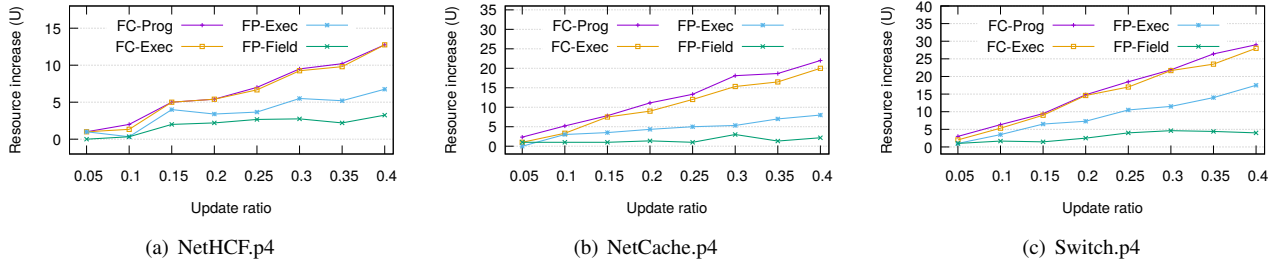
(b) NetCache.p4

(c) Switch.p4

Figure 7: The granular consistency specifications in FlexPlan can effectively reduce peak resource usage. FC-Prog and FC-Exec represent program consistency and execution consistency properties as defined in FlexCore [53], which only analyzes reachability across tables without considering program semantics. FP-Exec refines execution consistency to consider specific traffic classes in FlexPlan. FP-Field is a new consistency definition in FlexPlan that constrains the processing outcomes of specific header fields.
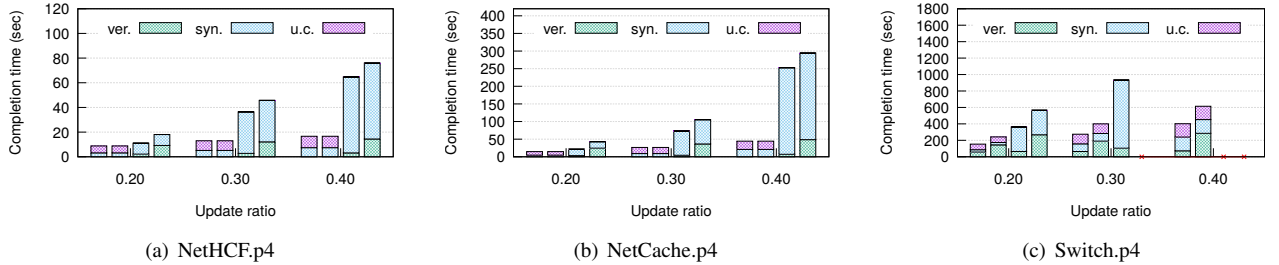

(a) NetHCF.p4

(b) NetCache.p4

(c) Switch.p4

Figure 8: The snapshot-based optimizations are effective. Each group of bars plots the turnaround times, from left to right, for FlexPlan, NoSnapV (no snapshot verification), NoSnapL (no shapshot learning), and NoSnapVL (disabling both optimizations), respectively. On switch.p4, both techniques are necessary for sizable changes; otherwise the analysis will time out after 30 minutes. We further break down each bar by the time the solution spends in verification, synthesis, and unsat core extraction.

lyze program semantics, it cannot distinguish whether or not a change actually affects a particular traffic class, conflating the two guarantees whenever bottleneck tables are modified.

In contrast, the granular consistency levels in FlexPlan are effective in reducing the peak resource usage by capturing program semantics. First, FlexPlan refines "execution consistency" even further by defining it over traffic classes of interest. Specifically, we have tested three variants of "execution consistency" defined over traffic classes (shown as 'FP-Exec'). For switch.p4, we specify it only for IPv4 traffic without tunneling; for NetCache, only for read requests to the cache data structure; and for NetHCF, only packets that establish TCP sessions. Thus, if a table change does not affect how these traffic classes are processed—even at a bottleneck table—FlexPlan is still able to produce granular transitions with lower peak resource usage. The reductions for NetHCF, NetCache, and switch.p4 are 47%, 64%, and 45%, respectively, compared to the fixed execution consistency in FlexCore. Further, we have also tested "field consistency," to showcase FlexPlan's ability to define new consistency levels beyond tuning traffic classes (shown as 'FP-Field'). This states that L2/L3 headers and standard metadata must be processed with the same outcome during transition. This leads to even lower peak usage: the reduction is 46%, 67%, and 68%, respectively, compared to the granular execution consistency levels above. This is because it allows a mix of old and new

tables to co-exist on execution paths, as long as this does not change the processing behaviors for specific header fields.

6.3 Snapshot learning and verification

Next, we evaluate the effectiveness of the snapshot learning (SnapL) and verification (SnapV) optimizations. We create three baseline solutions from FlexPlan where one or both optimizations are turned off: NoSnapV, NoSnapL, and NoSnapVL. Figure 8 compares the four solutions with different update ratios and programs. Across all data points, FlexPlan outperforms NoSnapVL in terms of completion time by 70% on average, demonstrating the effectiveness of the two snapshot-based optimizations. On switch.p4, the NoSnapVL baseline times out when $\alpha > 20\%$. Further decomposition shows that the SnapL and SnapV optimizations lead to 58% and 37% improvements on average, respectively.

Both SnapL and SnapV are more effective with larger programs (which lead to larger SMT formulas per snapshot) and higher update ratios (which lead to a larger update plan search space). For instance, at $\alpha = 40\%$, SnapL reduces the completion time by up to 81% for NetCache. With smaller update ratios, the number of possible update sequences is already small, so the time spent in learning unsat cores with SnapL does not afford as much improvement. The trend for SnapV is similar. For smaller formulas (e.g., NetHCF and NetCache), it is possible for FlexPlan to iterate through all unsat cores,
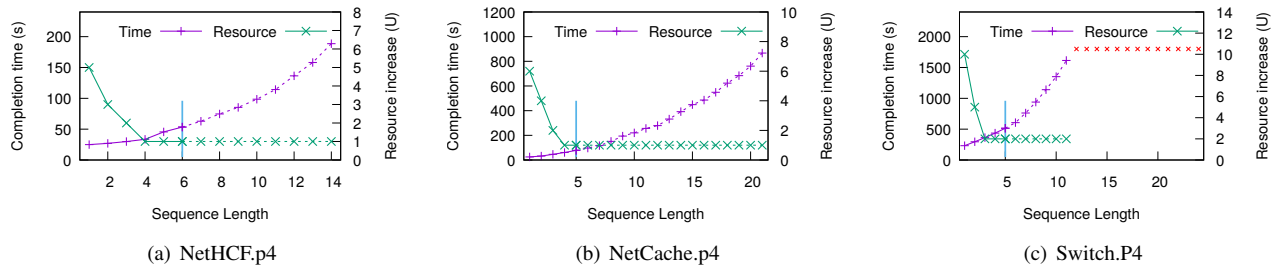
Figure 9: The FlexPlan introspection technique is effective in determining whether the synthesis should continue to try longer sequences. The vertical lines denote the stopping points for FlexPlan, which significantly outperforms a solution with introspection turned off, which naïvely increases the sequence length beyond the vertical lines until the maximal upper bound or timeout.

so the main CEGIS phase could bypass the verification completely since the first proposal attempt will identify a correct plan. Its impact is stronger on larger programs like switch.p4.

6.4   Introspection and diagnosis

We now evaluate the three CEGIS introspection techniques.

Existence. We first generate a set of program modifications that are guaranteed to violate the safety specification. For instance, if the specification requires that egress_spec processing outcomes should be preserved, we ensure that it is modified in a different way. For all cases, FlexPlan correctly rejects the annotated change as unsafe within 90 seconds.

Sequence length. Figure 9 evaluates the FlexPlan introspection for determining whether to attempt a longer sequence. We create cases where the resource constraints will guarantee an eventual failure, and test how soon FlexPlan can detect this inevitability. We also compare against a version of FlexPlan without this introspection. For NetHCF (α=40%), NetCache (α=20%), and switch.p4 (α=20%), FlexPlan concludes that the synthesis will fail when the sequence lengths are six, five, and five, respectively, within several minutes. However, the solution without introspection will keep increasing the sequence lengths (and running time) until the maximal upper bound (i.e., the total number of changes) or timeout, without being able to decrease resource usage further. It took $3\times$ and $12\times$ more time to conclude that the resources are insufficient for NetHCF and NetCache, respectively; for switch.p4, it times out before producing any useful results. Thus, the introspection technique helps determine failures efficiently.

Resource release. After each failure, we further ask FlexPlan to produce diagnostic results on the least amount of resource release that will enable a safe and feasible transition. This is achieved by introspecting the sequence upper bound for a safe transition, and then solving for a transition with minimized resource peak. This diagnosis took less than ten minutes across all update ratios and programs—the longer completion time is due to the need for trying the longest possible safe transition. We then emulated the release by increasing the switch headroom by the suggested amount, and re-attempted another synthesis and verified that it succeeded.

7   Related Work

Runtime programmability. Network switches have become programmable at runtime [3, 4, 9, 19, 52, 53], where switch programs can be modified with partial reconfiguration without downtime. Runtime programmability has also been studied in host networking [40, 42]. FlexPlan develops a formal approach to synthesizing runtime programmable switch updates.

Safe network updates. Ensuring the safety of network updates [34, 43] is a key goal in cloud datacenters [37]. In the context of OpenFlow-based SDN, consistent update algorithms have been extensively studied [21, 44, 45]. FlexPlan considers an analogous problem for programmable data planes, with new definitions on correctness and intermediate states, and uses program synthesis to achieve this goal.

Synthesis and verification. Program synthesis has found many applications in the networking domain [13, 22, 59], including for identifying safe configuration updates in OpenFlow SDN [39]. For programmable switches, existing projects have developed many formal verification techniques for P4 programs [18, 38, 50, 51, 61]. Compared to these lines of work, FlexPlan aims at synthesizing a correct-by-construction update sequence, which in turn requires new techniques.

8   Conclusion

Programmable networks enable the *development* of new features "in the field," without relying on slow-paced vendors. To safeguard network behaviors, formal verification has proven essential. Runtime programmable networks [57], in contrast, emphasize that the *deployment* of these features must also be seamless and "in the field"—without requiring slow-paced maintenance operations. However, live program modifications necessitate new techniques for providing formal assurance. FlexPlan is a synthesis tool that can identify a safe and feasible program transition sequence automatically. It introduces domain-specific techniques for synthesizing switch program updates. With comprehensive evaluation, we demonstrate the scalability of FlexPlan on real-world programs.

References

[1] BlueField SmartNIC Ethernet. https://www.mellanox.com/products/BlueField-SmartNIC-Ethernet.

[2] FlexPlan code repository. https://github.com/824728350/FlexPlan.

[3] Jericho2. https://www.broadcom.com/products/ethernet-connectivity/switching/stratadnx/bcm88850.

[4] The NPL network programming language. https://github.com/nplang.

[5] Nvidia/Mellanox Spectrum Ethernet Switches. https://www.nvidia.com/en-us/networking/ethernet-switching/spectrum-sn4000/.

[6] P4 Runtime Specification: Atomicity, Batch and Ordering of Updates: DataPlaneAtomic. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html#sec-batching-and-ordering-of-updates.

[7] P4 Specification: Annotations. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html#sec-annotations.

[8] The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[9] Trident4 boosts enterprise switch capacity to 12.8 terabit. http://www.gazettabyte.com/home/2019/7/11/trident-4-boosts-enterprise-switch-capacity-to-128-terabit.html.

[10] Wedge 100bf-32x 100gbe data center switch. https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335.

[11] 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. *ACM SIGCOMM CCR*, 44(3), 2014.

[12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[13] E. H. Campbell, W. T. Hallahan, P. Srikumar, C. Cascone, J. Liu, V. Ramamurthy, H. Hojjat, R. Piskac, R. Soulé, and N. Foster. Avenir: Managing data plane diversity with control plane synthesis. In *Proc. NSDI*, 2021.

[14] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, et al. dRMT: Disaggregated programmable switching. In *Proc. SIGCOMM*, 2017.

[15] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Cornell University Tech. Report*, 2009. https://hdl.handle.net/1813/11660.

[16] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. NSDI*, 2018.

[17] C. David and D. Kroening. Program synthesis: challenges and opportunities. *Philosophical Transactions A: Mathematical, Physical and Engineering Sciences*, 2017.

[18] D. Dumitrescu, R. Stoenescu, L. Negreanu, and C. Raiciu. bf4: Towards bug-free P4 programs. In *Proc. SIGCOMM*, 2020.

[19] Y. Feng, Z. Chen, H. Song, W. Xu, J. Li, Z. Zhang, T. Yun, Y. Wan, and B. Liu. Enabling in-situ programmability in network data plane: From architecture to language. In *Proc. NSDI*, 2022.

[20] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In *Proc. PLDI*, 2018.

[21] K.-T. Foerster, S. Schmid, and S. Vissicchio. Survey of consistent software-defined network updates. *IEEE Communications Surveys Tutorials*, 21(2):1435–1461, 2019.

[22] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta. Switch code generation using program synthesis. In *Proc. SIGCOMM*, 2020.

[23] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from Google's network infrastructure. In *Proc. SIGCOMM*, 2016.

[24] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proc. SIGCOMM*, 2018.

[25] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker. Contra: A programmable system for performance-aware routing. In *Proc. NSDI*, 2020.

[26] S. Jha and S. A. Seshia. Are there good mistakes? A theoretical analysis of CEGIS. In *Proc. SYNT*, 2014.

[27] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soule, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. NSDI*, 2018.

[28] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. SOSP*, 2017.

[29] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *Proc. SIGCOMM*, 2014.

[30] Q. Kang, J. Xing, Y. Qiu, and A. Chen. Probabilistic profiling of stateful data planes for adversarial testing. In *Proc. ASPLOS*, 2021.

[31] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. USENIX Security*, 2020.

[32] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. SOSR*, 2016.

[33] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[34] N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *Proc. HotNets*, 2013.

[35] G. Li, M. Zhang, C. Liu, X. Kong, A. Chen, G. Gu, and H. Duan. NetHCF: Enabling line-rate and adaptive spoofed IP traffic filtering. In *Proc. ICNP*, 2019.

[36] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating data center networks with zero loss. In *Proc. SIGCOMM*, 2013.

[37] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: updating data center networks with zero loss. In *Proc. SIGCOMM*, 2013.

[38] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, C. C. Robert Soulé, Han Wang, N. McKeown, and N. Foster. p4v: Practical verification for programmable data planes. In *Proc. SIGCOMM*, 2018.

[39] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. In *Proc. PLDI*, 2015.

[40] S. Miano, A. Sanaee, F. Risso, G. Rétvári, and G. Antichi. Domain specific run time optimization for software data planes. In *Proc. ASPLOS*, 2022.

[41] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. SIGCOMM*, 2017.

[42] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári. Dataplane specialization for high-performance OpenFlow software switching. In *Proc. SIGCOMM*, 2016.

[43] T. D. Nguyen, M. Chiesa, and M. Canini. Decentralized consistent updates in SDN. In *Proc. SOSR*, 2017.

[44] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[45] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proc. HotNets*, 2011.

[46] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[47] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Combinatorial sketching for finite programs. In *Proc. ASPLOS*, 2006.

[48] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. ATC*, 2018.

[49] M. Sousa and I. Dillig. Cartesian hoare logic for verifying k-safety properties. In *Proc. PLDI*, 2016.

[50] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu. Debugging P4 programs with Vera. In *Proc. SIGCOMM*, 2018.

[51] 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 practical usable verification system for production-scale programmable data planes. In *Proc. SIGCOMM*, 2021.

[52] T. Wang, X. Yang, G. Antichi, A. Sivaraman, and A. Panda. Isolation mechanisms for high-speed packet-processing pipelines. In *Proc. NSDI*, 2022.

[53] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen. Runtime programmable switches. In *Proc. NSDI*, 2022.

[54] J. Xing, K.-F. Hsu, Y. Qiu, Z. Yang, H. Liu, and A. Chen. Bedrock: Programmable network support for secure RDMA systems. In *Proc. USENIX Security*, 2022.

[55] J. Xing, Q. Kang, and A. Chen. Netwarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.

[56] J. Xing, A. Morrison, and A. Chen. NetWarden: Mitigating network covert channels without performance loss. In *Proc. HotCloud*, 2019.

[57] J. Xing, Y. Qiu, K.-F. Hsu, H. Liu, M. Kadosh, A. Lo, A. Akella, T. Anderson, A. Krishnamurthy, T. S. E. Ng, and A. Chen. A vision for runtime programmable networks. In *Proc. HotNets*, 2021.

[58] J. Xing, W. Wu, and A. Chen. Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries. In *Proc. USENIX Security*, 2021.

[59] Q. Xu, M. D. Wong, T. Wagle, S. Narayana, and A. Sivaraman. Synthesizing safe and efficient kernel extensions for packet processing. In *Proc. SIGCOMM*, 2021.

[60] L. Yu, J. Sonchack, and V. Liu. Mantis: Reactive programmable switches. In *Proc. SIGCOMM*, 2020.

[61] N. Zheng, M. Liu, E. Zhai, H. H. Liu, Y. Li, K. Yang, X. Liu, and X. Jin. Meissa: scalable network testing for programmable data planes. In *Proc. SIGCOMM*, 2022.

9 Appendix

This appendix includes more details on the safety specifications as summarized in Table 1 in the main paper.

9.1 Specifications

S4: Element consistency for ACL. This specification is similar as program consistency, but only constrains traffic that goes through a particular ACL table. For all packets that have been processed by the ACL table, their execution paths must be of the same version (i.e., either old or new).

```
1  specification {
2    ghost bit<1> sawOld = false;
3    ghost bit<1> sawNew = false;
4    ghost bit<1> acl_hit = false;
5    @old => { sawOld = true; }
6    @new => { sawNew = true; }
7    @hit('acl') => { acl_hit = true; }
8    all_old = {
9      $cur.in.acl_hit => !$cur.eg.sawNew;
10   }
11   all_new = {
12     $cur.in.acl_hit => !$cur.eg.sawOld;
13   }
14   assert all_old || all_new;
15 }
```

S5: Table consistency for ECMP. We introduce a new consistency definition that is not available from existing

work [53], which we call "table consistency." It states that for each packet, the table (in this case ECMP) hit/miss behavior in the intermediate state should be either the same with the old or the new processing logic.

```
1  specification {
2    ghost bit<1> ecmp_hit = false;
3    @hit('ecmp') => { ecmp_hit = true; }
4    // preserve processing behavior across states
5    table_consistency_ecmp = {
6      $cur.in == $old.in == $new.in =>
7        ($cur.eg.ecmp_hit == $new.eg.ecmp_hit ||
8          $cur.eg.ecmp_hit == $old.eg.ecmp_hit);
9    }
10   assert table_consistency_ecmp;
11 }
```

Although FlexPlan's primary focus is consistency guarantees during the program transition, its specification language can naturally support general program correctness properties that are used in P4 program verification [18, 38]. We showcase two of them below. At a high level, general program correctness properties are expressed by constraining a single snapshot using `$cur`, without referring to `$old` or `$new`.

S6: VLAN table access. Packets should go through VLAN table during any intermediate state.

```
1  specification {
2    ghost bit<1> vlan_hit = false;
3    @hit('vlan') => { vlan_hit = true; }
4    table_access_vlan = {
5      $cur.eg.vlan_hit == true;
6    }
7    assert table_access_vlan;
8  }
```

S7: Modification on ipv4.ttl. Any intermediate program snapshot should always decrement the ipv4.ttl header by one when the packet leaves the egress.

```
1  specification {
2    decrement_ipv4_ttl = {
3      $cur.eg.ipv4.ttl == $cur.in.ipv4.ttl - 1;
4    }
5    assert decrement_ipv4_ttl;
6  }
```

9.2 Instrumentations

Figure 10 includes an illustrative example that shows how a specification is translated into instrumentations in the input P4 program. This uses our running example `acl_flowlet_ecmp`, building upon the version sketch in Figure 3 and adding the instrumentations from the specification S1.

As we can see, the `sawOld`, `sawNew` and `acl_hit` variables are directly inserted into source program. We then add instrumentation that checks whether there exists a packet that violates the execution consistency after going through execution path. We then compute the weakest preconditions for

```
1  control ingress {
2    apply {
3      ghost_ipv4_valid = ipv4.isValid();
4      /* annotation site 1: acl->nat_acl */
5      if(ipv4.isValid()) {
6        if (! vsk.v1) { // @mod
7          acl.apply(); ghost_saw_old = 1;
8          ghost_acl_hit = 1;
9        } else {
10         nat_acl.apply(); ghost_saw_new = 1;
11         ghost_acl_hit = 1;
12       }
13     }
14     /* annotation site 2: delete flowlet */
15     if(! vsk.v2) {        // @del
16       if (ipv6.isValid())
17         flowlet.apply(); ghost_saw_old = 1;
18     }
19     /* annotation site 3: add ecmp */
20     if(vsk.v3) {      // @add
21       ecmp.apply(); ghost_saw_new = 1;
22     }
23     if (ghost_ipv4_valid && ghost_acl_hit) {
24       if (ghost_saw_old && ghost_saw_new){
25         violation();
26       }
27     }
28   }
29 }
```

Figure 10: Instrumenting the version sketch in Figure 3 with safety specification, and also adding statements that check the safety properties.

the reachability of the violation nodes. This is achieved by iterating through CFG nodes and propagating stronger conditions to all their neighbors based on the transition relation. We then check whether the predicate is valid using the Z3 theorem prover. In the example, the combined program and safety formula to check would be derived as:

$$
\begin{aligned}
gso &= (\text{ipv4.isValid} \wedge \neg\text{vsk.v1}) \vee (\neg\text{vsk.v2} \wedge \text{ipv6.isValid}) \\
gsn &= (\text{ipv4.isValid} \wedge \text{vsk.v1}) \vee \text{vsk.v3} \\
gah &= \text{ipv4.isValid} \\
check &= \neg(\text{ipv4.isValid} \wedge gah \wedge gso \wedge gsn)
\end{aligned}
$$

where *gso* represents the logical formula for when `ghost_saw_old` is assigned true, and similarly for *gsn* as `ghost_saw_new` and *gah* for `ghost_acl_hit`. The logical variable *gso*, for example, captures the path conditions required to set the ghost variable to true as well as any intermediate assignments on the path to variables that might affect these path conditions/branches.