KeySight: Troubleshooting Programmable Switches via Scalable High-coverage Behavior Tracking

Yu Zhou*†‡, Jun Bi*†‡, Tong Yang§, Kai Gao*†‡, Cheng Zhang*†‡, Jiamin Cao*†‡, Yangyang Wang*†‡
*Institute for Network Sciences and Cyberspace, Tsinghua University
†Department of Computer Science, Tsinghua University
‡Beijing National Research Center for Information Science and Technology (BNRist)
§Peking University

***Abstract*—The rise of programmable switches and P4 brings much flexibility to networks, but this flexibility comes with increased risks of bugs. Diagnosing these bugs is essential for network operation but is non-trivial. A potential approach is to track packet behaviors through postcards, but existing tools either generate substantial postcards (*limited scalability*) or only track a small proportion of packet behaviors (*low coverage*). In this paper, we present KeySight, a platform that troubleshoots programmable switches with high scalability and high coverage. The key idea is based on the Packet Equivalence Class (PEC) abstraction that aggregates packets with identical packet behaviors and generates one postcard per behavior. The PEC abstraction minimizes the number of postcards while tracking all packet behaviors. We design novel algorithms to analyze PECs of P4 programs and to implement the PEC abstraction on programmable switches. We deploy KeySight on Tofino and SmartNIC, and evaluate it against 80 P4 programs and real packet traces of over 5TB. Results show that in the premise of overseeing 99.9% packet behaviors, KeySight reduces the number of postcards by one to two orders of magnitude when comparing with NetSight.**

I. Introduction

Programmable switches [1–3] and P4 oblige operators with much flexibility to define novel data plane functions and protocols, motivating fast innovations on networked systems. Recent studies develop various P4 programs to enhance networks concerning performance [4, 5], measurement [6, 7], and so on. Although providing flourish functions and significant benefits, this flexibility comes at a price of increased chances and diversity of subtle bugs [8–10], which potentially causes severe performance degradation and network outages [11, 12].

Thus, detecting and locating bugs (*e.g.,* P4 program bugs, incorrect policies, and hardware faults) in programmable switches are of great importance. A general approach is to directly track packet behaviors through *postcards* [13]. Postcards are mirrored packets accommodating a set of fields which faithfully record the changes taking place on output ports, packet headers, and switch states (*e.g.,* counters) when a switch processes packets. These changes are often referred to as *packet behaviors*. Packet behavior tracking can provide visibility of how switches process packets and is very useful in network troubleshooting.

However, troubleshooting programmable switches via packet behavior tracking is non-trivial and should satisfy two essential requirements. (1) *High scalability* denotes that *postcard load* (*i.e.,* the number of generated postcards) on every switch should be considerably small. Otherwise, if postcard load is close to normal traffic load, every switch has to reserve a large amount of bandwidth and switching capacity to transmit postcards, which compromises the feasibility. (2) *High coverage* requires overseeing all packet behaviors to detect abnormalities completely and to locate bugs quickly. Otherwise, some abnormalities that bring severe network outages may be left out, undermining the troubleshooting accuracy.

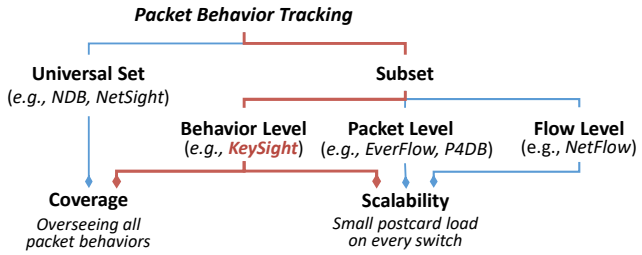


Figure 1. **KeySight vs. other troubleshooting techniques.**

No existing tool, to the best of our knowledge, can satisfy the two requirements simultaneously that they have to make different trade-off decisions. First, NetSight [13] generates postcards for the *universal set* of packets to acquire all packet behaviors in a network. NetSight has to generate massive postcards and is not scalable. Second, some tools generate postcards for a *subset* of packets. NetFlow [14] samples packets periodically to report five-tuples and other performance information at the *flow level*. EverFlow [15] and P4DB [10] introduce a match-mirror design, *i.e.,* employing a Match-Action Table (MAT) to generate postcards at the *packet level*. NetFlow, EverFlow, and P4DB inevitably encounter the coverage issue, as they can only track a subset of packet behaviors. In summary, existing tools either sacrifice scalability for overseeing all packet behaviors or sacrifice coverage for keeping postcard load under control. A fundamental reason is that they use inappropriate postcard generation granularities, constrained by limited support in legacy switches.

To achieve efficient programmable switch troubleshooting, we present KeySight, a platform that supports scalable high-coverage behavior tracking. KeySight is based on an observation that in a switch, many packets have the same behaviors, and the number of packet behaviors is relatively predictable [16, 17] and much smaller than that of packets. This observation identifies an opportunity that *we can generate postcards for a representative subset of packets to track all packet behaviors*. Unlike the state of the arts, KeySight takes

a different abstraction called *Packet Equivalent Class* (*PEC*) to generate probes. The PEC abstraction aggregates packets with identical behaviors and generates postcards at a new granularity, namely the *behavior level* which generates one postcard per packet behavior. With this abstraction, KeySight remarkably decreases postcard load on each switch (by one to two orders of magnitude). Moreover, postcard load of KeySight increases linearly with the number of packet behaviors instead of packets, thus KeySight embraces higher scalability than existing tools. Meanwhile, behaviors of PECs can equivalently represent behaviors of all packets. Thus, this abstraction does not undermine coverage. Note that VeriFlow formally proposes PEC to optimize the efficiency of verifying MAT rules issued by control planes, while KeySight employs PEC to track packet behaviors on data planes. Moreover, to make the PEC abstraction a reality, KeySight comes up with the following two techniques.

$\mathcal{D}_1$: Automatic PEC Extraction from P4 Programs. As P4 enables operators to customize packet processing logic of programmable switches, packets in switches configured with different P4 programs expose different behaviors. We propose to use PEC representations to model packet behaviors defined by P4 programs, and the PEC representation identifies which fields a postcard should carry. However, *manually* extracting PEC representations of P4 programs comprising dozens of or even hundreds of MATs (*e.g.,* NetCache.P4 [18] has 96 MATs and Switch.P4 [19] has 129 MATs) is *cumbersome* and *error-prone*. Moreover, dependencies [20] between MATs further increase this complexity. To this end, we design an analyzer that performs static analysis on P4 programs and automatically extracts PEC representations.

$\mathcal{D}_2$: Efficient PEC Implementation on Programmable Switches. To generate postcards at the behavior level, the PEC abstraction needs to store all packet behaviors and de-duplicate postcards carrying identical behaviors, which requires a large amount of memory space and complex filtering operations. Thus, fully implementing the PEC abstraction on programmable switches whose programmability and memory are constrained is almost impossible. To bound the memory usage and simplify filtering logic, we propose to *conduct approximate postcard de-duplication over recently-arrived packets*. This idea comes with issues of false positives (cannot guarantee generating postcards for all behaviors) and false negatives (repeatedly generate postcards for some behavior) in postcard de-duplication. To implement the PEC abstraction while keeping the overheads low, we design a new algorithm, named *Ring Bloom Filter* (*RBF*). RBF is based on a ring composed of multiple Bloom filters. The ring records the appeared behaviors of recently-arrived packets, and RBF supports clearing expired packet behaviors from the ring and reserving space in the ring for new packet behaviors. Moreover, RBF is highly compatible with P4 and can be entirely implemented on data planes with bound memory usage. It is worth noting that RBF could be flexibly tuned and has low false positive rates (less than 0.1%) even when the available memory is limited.

In this paper, we make the following contributions:

- We propose KeySight which exploits the PEC abstraction to troubleshoot programmable switches while satisfying high scalability and high coverage simultaneously.
- We design an analyzer to extract PEC representations from P4 programs (§III). We design a novel algorithm, RBF, to implement the PEC abstraction on programmable switches, and a suite of APIs to conduct troubleshooting tasks specified by operators (§IV).
- We implement a prototype of KeySight on Tofino [3] and SmartNIC [21], and publish the code at [22]. Evaluation against 80 P4 programs and over 5TB packet traces shows that in the premise of overseeing more than 99.9% packet behaviors, KeySight can reduce postcard load by one to two orders of magnitude (§V).

II. Background and Overview

A. Preliminary Knowledge of P4

As a language for programming switches, P4 offers a lot of reconfigurable elements to implement rich packet processing functions. (1) Operators can customize *parsers* to extract headers with arbitrary protocol formats. (2) Operators can construct *compound actions* with a variety of *primitive actions* (*e.g., modify_filed*). (3) Operators can specify match fields (*e.g.,* IPv4 destination address) with match types (*e.g., exact* and *range*) and multiple *compound actions* in one Match-Action Table (MAT). A MAT can execute specified actions on packets according to the matching results of packet headers. (4) Operators can organize multiple MATs as a consolidated Directed Acyclic Graph with the *control flow*. (5) Operators can declare variables (*e.g., metadata*) and stateful elements (*e.g., counters* and *registers*) to store flow states transiently or consistently. With above elements, operators can develop various data plane functions with P4 programs efficiently.

The lifecycle of P4 programs is composed of two phases. At *compile time*, the P4 compiler takes the responsibility of compiling P4 programs to executable code and generating control APIs. Then, at *runtime*, the P4 pipeline, which accommodates parsers, MATs, and so on, is configured with the executable code and processes packets according to P4 programs. Meanwhile, the centralized controller populates MAT entries through the control API. Notably, a MAT entry comprises match fields, the compound action to be executed, and action parameters.

Regarding programmability, there are also some limitations in P4 and programmable switches. For example, P4 does not allow multiple reads to the same MAT by a packet. Some programmable switches only support atomic access to stateful elements in consideration of implementation complexity and performance guarantee [23].

B. Design Overview

As is shown in Figure 2, KeySight comprises three components that work in the two phases of the P4 program lifecycle. (1) At compile time, ***KeyAnalyzer*** takes a P4 program as input and generates the *PEC* representation. (2) The P4 compiler generates executable code for the P4 program and

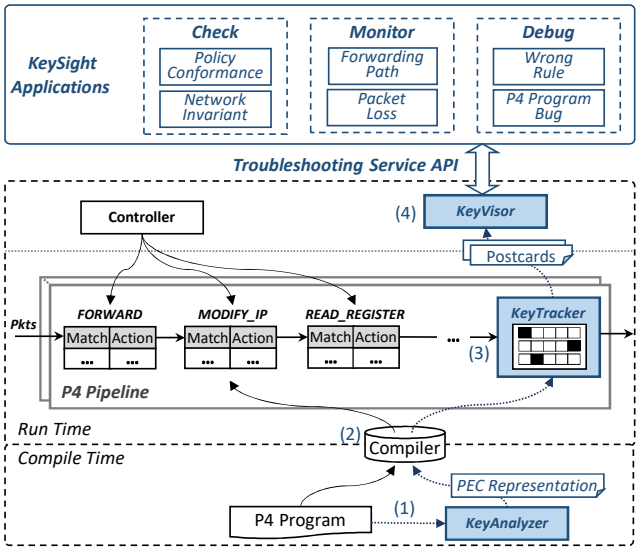


Figure 2. **Architectural overview of KeySight. KeySight is composed of KeyAnalyzer (§III), KeyTracker (§IV-A), and KeyVisor (§IV-B).**

deploys the code into the P4 pipeline. (3) At runtime, every packet traversing the P4 pipeline gets a postcard according to the PEC representation. Then, ***KeyTracker*** checks postcards and determines whether to report the postcard based on the following principle: Only report postcards that have never been seen by KeyTracker before. (4) After collecting postcards from every switch, ***KeyVisor*** conducts troubleshooting tasks and makes the other components transparent to operators with *troubleshooting service API*s.

III. KeyAnalyzer at Compile Time

In the context of programmable switches, P4 programs and MAT entries jointly determine PEC. Especially, P4 programs specify which fields are essential to represent PEC, and values of these fields depend on MAT entries. Thus, as P4 programs vary and may be too complicated to be manually analyzed, an automatic tool to extract PEC representations from P4 programs is well needed.

A. Defining PEC in the Context of Programmable Switches

Previous tools [16, 17] have defined PEC for legacy switches and apply it to accelerate network verification, but there is no researching effort to model PEC of programmable switches and to exploit PEC to track packet behaviors. We propose a PEC definition in the context of programmable switches, and the definition guides systematic extraction of PEC representations. We concentrate on defining PEC for a single switch, while we can easily deduce the network-wide PEC definition [24].

Definition (Packet Equivalence Class): In a programmable switch, a Packet Equivalence Class is a set $\mathcal{C}$ of packets such that any packet $pkt_1, pkt_2 \in \mathcal{C}$ satisfy:

- $\boldsymbol{\mathcal{P}_1}$: pkt_1 and pkt_2 traverse the same MATs, and those MATs compose a MAT set;
- $\boldsymbol{\mathcal{P}_2}$: For each MAT in the MAT set, pkt_1 and pkt_2 hit the identical MAT entry whose compound action is $\mathcal{A}$;
- $\boldsymbol{\mathcal{P}_3}$: For each primitive action invoked in $\mathcal{A}$, pkt_1 and pkt_2 have the same deterministic inputs and outputs.

This definition stresses on deterministic inputs and outputs of every primitive action. P4 also has some non-deterministic objects, such as time stamps and queue lengths, whose values are determined by random variables [25]. The non-determinacy in P4 programs can impact traversed MATs and MAT entries of packets through conditional statements and range matching. $\mathcal{P}_1$ and $\mathcal{P}_2$ can equivalently express how the non-determinacy impacts packet behaviors. Thus, in this paper, we only consider deterministic objects by default.

The above PEC definition differs from the previous PEC definition for legacy switches in two perspectives. (1) For $\mathcal{P}_1$, the previous one always assumes that packets only traverse a single MAT in a switch, while the new PEC definition should consider multiple MATs composing a MAT path in the control flow. The MAT path introduces dependencies between MATs, which needs careful treatment. (2) For $\mathcal{P}_3$, operators construct a compound action with multiple primitive actions. Thus, the correctness of compound actions relies on not only the action parameters specified by MAT entries but also the used primitive actions. Besides, P4 primitive actions are more complicated than those supported by legacy switches. Therefore, exiting tools based on the previous PEC definition cannot fully model PEC in programmable switches. We need a program-independent approach that can attain PEC representations from different P4 programs.

B. KeyAnalyzer to Extract PEC Representations

To attain PEC of P4 programs based on the above definition, we propose a notion called the PEC representation. A PEC representation is a list of fields that identify inputs and outputs of each MAT in a P4 program. Take a P4 program with a MAT called *ip_forward* as an example. The destination IPv4 address (ipv4.dip) is the input of *ip_forward*, and the egress port (port) is the output. Thus, the PEC representation for this program is [ipv4.dip, port]. The PEC representation correspondingly identifies which fields postcards should include. After traversing the P4 pipeline, each packet gets a postcard based on the PEC representation. For the above example, a postcard can be ['1.0.0.1', 1]. Furthermore, the PEC representation and postcards should satisfy that *for any packet pkt_1 and pkt_2, they belong to a PEC if and only if postcard p_1 of pkt_1 equals to postcard p_2 of pkt_2.*

To extract PEC representations, KeyAnalyzer should answer three questions: (1) How to extract inputs and outputs of compound actions comprising various primitive actions? (2) How to extract inputs and outputs of MATs accommodating various matching fields and multiple compound actions? (3) How to extract inputs and outputs of the P4 pipeline while tackling the complexity derived from MAT dependencies (*i.e., match dependency*, *action dependency*, and *reversed-match dependency* [20])? To answer these questions, KeyAnalyzer is designed with three steps which hierarchically analyze P4 programs. Figure 3 shows the basic workflow of KeyAnalyzer based on a typical P4 program, Router.P4 [26].

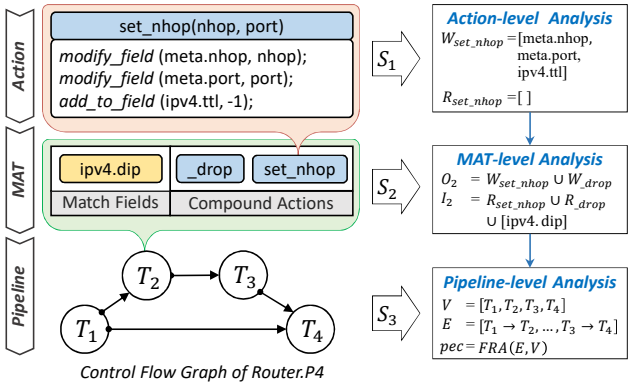


Figure 3. **PEC representation extraction workflow of KeyAnalyzer.**

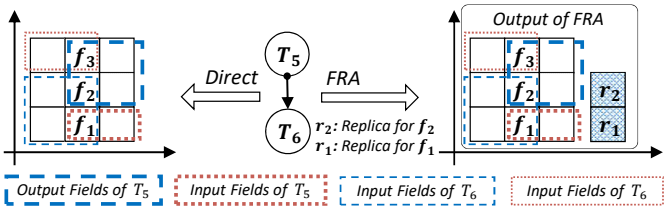


Figure 4. **The HSA-style illustration for a MAT dependency example. Field f_1 triggers reversed-match dependency. Field f_2 triggers action dependency. Field f_3 triggers match dependency.**

$\mathcal{S}_1$: Action-level Analysis. The goal of $\mathcal{S}_1$ is to extract written fields and read fields of every compound action. First, we have thoroughly checked all 31 primitive actions supported by P4, and we specify written fields and read fields of every primitive action according to the P4 specification. For example, for *modify_field(dst, src)*, the *dst* field is written by this primitive action, while the *src* field is read. Notably, KeyAnalyzer cannot analyze all primitive actions, because some primitive actions could modify switch states that are opaque to P4 programs. For example, the *count* primitive action could count packets with a counter, while P4 programs cannot know the exact value of the counter. There are five primitive actions that cannot be fully analyzed, including *push*, *pop*, *count*, *execute_meter*, and *generate_digest*. Second, we can get written and read fields of a compound action by combining those of every primitive action invoked in the compound action. For example, in Figure 3, *set_nhop* has three written fields (W_{set_nhop}) and has no read field (R_{set_nhop}).

$\mathcal{S}_2$: MAT-level Analysis. We design $\mathcal{S}_2$ to extract input fields and output fields of every MAT. As packets exclusively execute one compound action in every traversed MAT, we could attain input fields and output fields of a MAT by merging the read field set and the written field set of every compound action. Further, as match fields are also read by the MAT, inputs of a MAT should also include match fields.

$\mathcal{S}_3$: Pipeline-level Analysis. $\mathcal{S}_3$ aims at resolving complexity derived for MAT dependencies and extracting PEC representations on the basis of $\mathcal{S}_1$ and $\mathcal{S}_2$. To illuminate this step, we present a simple example composed of two MATs, *i.e.,* T_5 and T_6, in Figure 4. We employ the geometric approach of HSA [27] to demonstrate MAT dependencies as well as the relevant problem. A box in the coordinate axis denotes a

Algorithm 1: Field Replication Algorithm

Input: A control flow graph $G = (V, E)$
Output: A PEC representation (*pec*)

1 $pec \leftarrow \emptyset$;
2 $V^* \leftarrow$ Reverse(TopologicalSort(V, E)) ;
3 **foreach** v *in* V^* **do**
4 **foreach** *child vertex* v' *of* v **do**
5 $O_v^c \leftarrow O_v^c \cup O_{v'}^c$;
6 $R_v \leftarrow R_v \cup (O_v \cap O_{v'}^c)$;
7 $R_v \leftarrow R_v \cup (O_v \cap R_{v'}^a)$;
8 $R_v^a \leftarrow R_v^a \cup R_{v'}^a$;
9 **end**
10 $O_v^c \leftarrow O_v^c \cup O_v$;
11 $R_v^a \leftarrow (R_v^a \setminus R_v) \cup (I_v \cap O_v^c)$;
12 $pec \leftarrow pec \cup O_v \cup I_v \cup \{r |\ r$ *is a replica of* $x, x \in R_v\}$;
13 **if** v *is the last of* V^* **then**
14 $pec \leftarrow pec \cup \{r |\ r$ *is a replica of* $x, x \in R_v^a\}$;
15 **end**
16 **end**
17 **return** *pec*;

particular field, such as TTL of IPv4. MAT dependencies cause that input fields, and output fields of T_6 overlap those of T_5. KeyTracker can only observe the field after T_6. A strawman approach is to directly use the existing fields to construct a PEC representation for the program, but this approach does not work well due to the following observations. f_1 and f_2 will be modified by T_6, KeyTracker cannot get the value of f_1 and f_2 when a packet is processed by T_5. For f_3 which triggers the match dependency and is read by T_6, KeyTracker could receive its value got from T_5 without any ambiguity. The above statement implies that the reversed-match dependency and action dependency could undermine the correctness of attaining inputs and outputs of every MAT.

Based on the above analysis, we design a *Field Replication Algorithm* (*FRA*), in Algorithm 1, to overcome the problem caused by the dependencies. As is shown in Figure 4, the key idea of FRA is to create replicas for fields triggering the *reversed-match dependency* and *action dependency*. Besides, FRA is able to identify the fields triggering dependencies as well as to decide which fields to be replicated and which MAT to conduct field replications.

FRA models the control flow of a P4 program as a directed acyclic graph G. G has a set of vertices (V) that are MATs and if-else expressions, and a set of directed edges (E) that denotes execution sequences of vertices. A vertex v in G has five field sets. First, O_v and I_v denote the inputs and outputs of v, and are extracted at $\mathcal{S}_2$. Second, O_v^c denotes the accumulative outputs of v and its children. Third, R_v denotes the fields that should be replicated at v. Lastly, R_v^a denotes the fields that should be replicated at some ancestor of v. Except for O_v and I_v, the other three field sets are initialized as $\emptyset$ and will be populated in FRA.

FRA executes the sort-of-reversed topological sort on G to get V^* (line 2). Sequentially iterating through V^* could guarantee that for a specific vertex, all of its children have been processed at the time of processing itself. Then, FRA gets the cumulative outputs (line 5) and checks action dependency (line 6). For R^a, FRA validates whether the vertex could perform

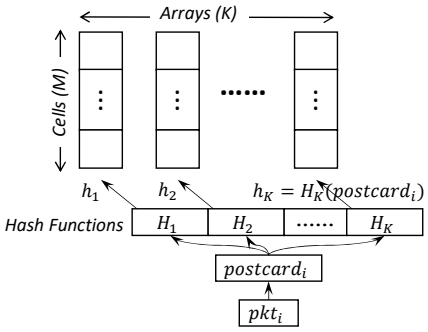


Figure 5. **Process a packet with a BF.**

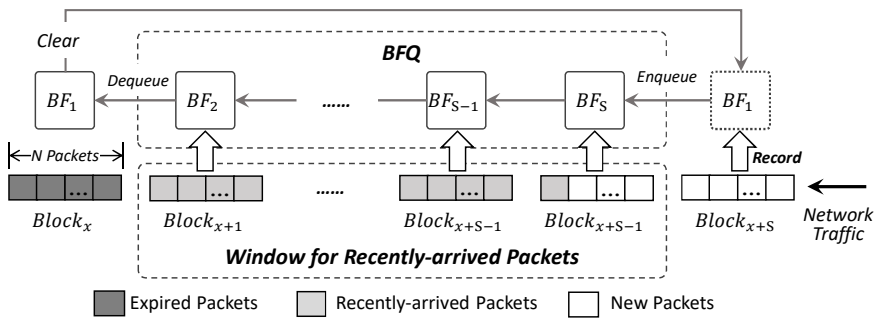


Figure 6. **BFQ design to track behaviors of recently-arrived packets.**

field replication (line 7), and add the fields triggering reversed-match dependency into R^a of its children (line 11). The PEC representation *pec*, initialized as $\emptyset$ (line 1), comprises inputs, outputs, and replicated fields of every vertex (line 12). Note that as the starting vertex of G has no ancestor, fields in R^a of the starting vertex should be replicated before packets enter into G, so *pec* also includes these fields (line 14).

FRA specifies which fields should be replicated and which MATs replicate these fields through R and R^a. To implement R and R^a in a P4 program, KeyAnalyzer modifies original compound actions to replicate the fields through *modify_field* and to store replicated fields in metadata.

Summary. We use N_F to denote the number of header fields and metadata fields in a P4 program. The time complexity of FRA is $O(N_F(|V|+|E|))$. Further, $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ work jointly to extract PEC representations satisfying $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$. The detailed complexity analysis and the proof for KeyAnalyzer are listed at [24]. In P4, there are some special components, such as *action profile* and *field list calculation*, which should also be considered. Fortunately, the above methods can support most of these components. After getting the PEC representation, KeyAnalyzer automatically incorporates the representation, replication actions, and the code of KeyTracker into the original P4 program.

IV. KeyTracker and KeyVisor at Runtime

In this section, we present how to track packet behaviors with *KeyTracker* and how to provide troubleshooting services based on postcards with *KeyVisor*.

A. KeyTracker Design to Track Packet Behaviors

After the P4 program is deployed on programmable switches, the P4 pipeline processes packets as specified by the original P4 program and produces postcards for every incoming packet according to the PEC representation. In other words, these postcards instantiate the PEC representation. Then, KeyTracker takes postcards as input. Through checking whether a postcard has been previously seen, KeyTracker de-duplicates postcards with the same packet behavior.

Considering large quantities of packet behaviors [16], we argue that in terms of checking duplicated postcards, KeyTracker should satisfy the following statement: *KeyTracker should be able to narrow the checking scope down to recently-arrived packets. In other words, KeyTracker should only conduct postcard de-duplication over recently-arrived packets, instead of all packets.* We use $\mathcal{X}$ to represent the above statement. As storing all behaviors for infinite packets requires substantial and unpredictable memory space, it is reasonable to bound the required memory through shrinking the checking scope.

Specifically, the reasons that $\mathcal{X}$ holds for the troubleshooting scenario are as follows. (1) Compared with past packets, recently-arrived packets are more important for real-time troubleshooting. (2) As switch configurations are dynamic (*e.g.,* network updates), some packet behaviors may expire, and some new ones will be enabled. Without $\mathcal{X}$, expired behaviors of old packets could take up an amount of memory space for a long time, which inevitably increase the FPR and undermines troubleshooting coverage. (3) As most flows have anticipated duration, carefully removing recorded behaviors of completed flows does not increase the number of generated postcards, which has a small impact on scalability.

In this section, we first present a strawman solution that satisfies $\mathcal{X}$. Then, we demonstrate why this strawman solution fails for network troubleshooting and come up with a novel algorithm which is compatible with programmable switches while satisfying $\mathcal{X}$.

Bloom Filter Queue. We propose a strawman design, named *Bloom Filter Queue* (BFQ), which is a FIFO queue composed of multiple Bloom Filters (BF) and can satisfy $\mathcal{X}$. BFQ is not a practical algorithm for *KeyTracker*, as its implementation on programmable switches introduces large overheads.

Before delving into the design of BFQ, we show how a BF checks an individual packet. As is shown in Figure 5, a BF comprises K arrays, each of which comprises M cells. Packet pkt_i gets $postcard_i$ according to the PEC representation when traversing the P4 program. Then, $postcard_i$ will be used to generate K positions $(h_1, ..., h_K)$ through K different hash functions $(H_1, ..., H_K)$. Afterwards, BFQ can get K cells for pkt_i. In the BF, a cell contains only one bit. If all values of the K cells are 1, the BF marks pkt_i as positive (the packet behavior has been seen before). Otherwise, the BF marks pkt_i as negative (*i.e.,* the packet behavior has never been seen).

After presenting how a packet is statically checked, we illuminate how packets sequentially traverse BFQ. As is shown in Figure 6, BFQ is composed of S BFs. BFQ sequentially divides packets into fixed-size blocks, each of which has N packets. Each BF exclusively records postcards for a block of packets. For example, BF_1 records postcards of packets

in $Block_x$. To record postcards of a particular packet, the attained cells via K hash functions in the corresponding BF should be set to 1. Every packet queries recent $S-1$ BFs to check whether its postcard has been recorded. If all BFs mark this packet as negative, BFQ has never seen the postcard of the packet in recently-arrived packets. Then, BFQ marks this packet as negative and report the postcard of this packet. Otherwise, BFQ marks the packet as positive and does report its postcard. Meanwhile, to prevent influences from expired packets, BFQ should clear all cells in the dequeued BF, *i.e.,* BF_1. When all packets in the current block ($Block_{x+S-1}$) complete, the new block ($Block_{x+S}$) will be shifted into the window, and its BF (cleared BF_1) will be enqueued at the same time. Similar to the above procedure, BFQ dequeues BF_2 whose block becomes expired.

However, it is hard for programmable switches to entirely implement BFQ due to the following concerns. (1) BFQ needs to implement multiple BFs and requires many querying operations, *i.e.,* $(S-1)K$ per packet, which occupy many resources (*e.g.,* ALU and SRAM). (2) BFQ requires multiple writes on registers to clear the dequeued BF, which is not supported by P4 (§II-A). There is a workaround, *i.e.,* exploiting the controller to clear the dequeued BF. For every block, the P4 pipeline can inform the controller to clear the dequeued BF through the control channel. This approach imposes large load between the controller and switches. Moreover, the latency of the control channel leads to inconsistency between the controller and switches, so it is hard to guarantee the BF to be reset is the oldest one. Besides, there are some existing algorithms [28, 29]) that can de-duplicate redundant elements in an infinite stream, but they do not satisfy $\mathcal{X}$ and encounter the same issues with BFQ.

Ring Bloom Filter. To overcome the practicality issue of BFQ while satisfying $\mathcal{X}$, we propose a novel algorithm, namely *Ring Bloom Filter* (RBF). As is shown in Figure 7, RBF improves BFQ with two aspects of design.

(1) Merging BFs into one special BF with multiple-bit cells. First, assume that all BFs in BFQ use the same set of hash functions. This assumption is fine for BFQ, as different BFs independently record postcards of different packets. With this assumption, the same postcard will be mapped to the same position in different BFs. Then, we could merge BFs into a new BF, the cell of which has S bits and constructs a bit ring. More specifically, as is shown in the left part of Figure 7, we merge the first bit arrays of $BF_1, ..., BF_S$ to form the first ring array of RBF. After that, RBF has K arrays, each of which has M cells. Rings in RBF cells can record postcards for recently-arrived packets, just like BFQ. This design reduces the number of querying operations per packet to K.

(2) Clearing the dequeued BF entirely in programmable switches. In a ring, we can reset the oldest bit (the bit from the expired BF) to 0 at the same time of setting the current bit (the bit from the current BF) to 1. A concern raises that we cannot guarantee clearing all oldest bits in RBF, as the packets could only clear the oldest bit in the same ring due to limitations of P4 (§II-A). We make the improvement in two

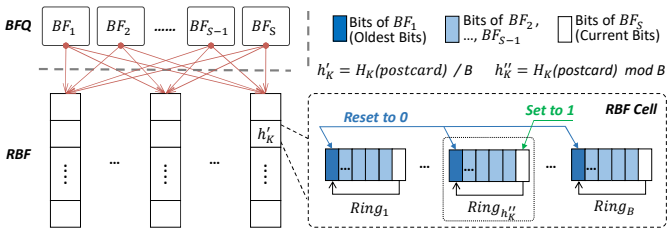


Figure 7. **Design of RBF: BF merging and BF clearing.**

perspectives to alleviate this concern, as is shown in the right part of Figure 7. First, we incorporate B ($B > 1$) rings into one cell. We still need to map a packet to a ring, RBF attains the ring position via two steps: The first one gets a cell position by $H_K(postcard)\ /\ B$. The second step gets a ring position by $H_K(postcard)\ mod\ B$. Second, packets will erase all oldest bits of rings in the cell when setting the current bit to 1. RBF could clear most oldest bits and can be fully implemented in switches. Although RBF cannot guarantee clearing all oldest bits, experiments show that the FNR and FPR of RBF are close to BFQ (§V-C).

To show how RBF works, we present an example with concrete numbers. Assume $K=3$, $S=4$, $B=4$, $M=4$, and each block has 10 packets. For the 41^{st} packet, we assume its postcard is $postcard_{41}$ and $H_1(postcard_{41}) = 5$. Thus, the index of the corresponding cell is $5/B = 1$, and the index of the ring in the cell is $5\ mod\ B = 1$. We can get a ring for $postcard_{41}$ in the first array and record $postcard_{41}$ in the 1^{st} bit of ring. If all bits except the 2^{nd} bit in the ring are 0, the first array does not record $postcard_{41}$. Then, in the ring, we need to set the 1^{st} bit to 1 and set the 2^{nd} bits of all rings in the same cell to 0.

Postcard generation in programmable switches. Programmable switches have multiple choices to generate postcards. (1) We can efficiently implement the same postcard generation function with EverFlow and NetSight in programmable switches through packet cloning actions, such as *clone_ingress_to_egress*. (2) Except for cloning actions, we can also use the *generate_digest* action to report postcards through the control channel. Furthermore, we need to add a MAT at the end of KeyTracker to execute the above actions.

B. KeyVisor Design to Supply Troubleshooting Services

With the support of KeyAnalyzer and KeyTracker, KeyVisor could get a real-time fine-grained view into programmable switches, which helps solve various challenging problems, such as stateful behavior verification and MAT-level bug location. Firstly, we present a suite of APIs facilitating operators to specify troubleshooting tasks. Then, we identify two noteworthy features of KeyVisor and present sample applications enabled by the features. We mainly focus on device-level troubleshooting in this paper, while discussing network-wide troubleshooting briefly in §VI.

Troubleshooting service API. KeyVisor provides three troubleshooting service APIs which have the same input parameters, *i.e.,* a field-value list (FVL) and a position specification. Figure 8 lists their syntax. An FVL comprises multiple field-

Syntax	
fvl	::= "[" fv {, fv} "]"
fv	::= "<" name, value, op ">"
op	::= > | >= | == | <= | < | !=
pos	::= "<" *sid*, *mid*, *aid*, *pt* ">"
sid	::= Switch id | *
mid	::= MAT name | *
aid	::= Compound action name | *
pt	::= BEFORE | AFTER | *

Applications	Troubleshooting Service API	Description
Dynamic ASSERT-P4	D1 = **Assert**([< std.egress_port, 0 , != >] , <*, *, Drop, AFTER>)	Implement an example of ASSET-P4 at run time. 0 is assumed to be the default drop port.
Blackhole Detector	L1 = **Track**([< std.egress_port, 0 , == >] , < *, LastMAT, *, AFTER >) L2 = **Track**([< std.egress_port, 0 , != >] , < *, FirstMAT, *, BEFORE >) L3 = **Track**([< std.egress_port, 0 , != >] , < *, LastMAT, *, AFTER >) ... // Detect blackholes in switches with L1 .. // Detect blackholes between switches with L2 and L3	Assume all switches maintain packet counters for ports. For blackholes in switches, we could find postcards whose egress post is 0. For blackholes between switches, we can calculate counter differences of adjacent ports in different switches.

TCP State Tracer	T1 = **Query**([< ipv4.dst, '10.0.0.3', == >, < ipv4.src, '10.0.0.4', == >, < tcp.src, 1080, == >, < tcp.dst, 80, == >] , < S2, *, *, AFTER >)
	... // Get the register index IDX from S1
	T2 = **Query**([< state_reg.index, IDX, == >], < S2, *, *, AFTER>)
		Assume S2 runs a connection tracking function. Get the index IDX for a particular flow with T1. As the reversed flow (from 10.0.0.3:80 to 10.0.0.4:1080) also modifies the state, we get all the postcards T2 of the register and trace state transitions.

Figure 8. **Syntax and sample applications of the troubleshooting service API.**

value tuples, each of which specifies the name of a field, its value, and the operation between the field and the value. For example, the FVL [< ipv4.src, '1.0.0.1', == >, < ipv4.dst, '1.0.0.2', != >] specifies the packets whose IPv4 source address is '1.0.0.1' and destination address is not '1.0.0.2'. The second parameter is a four-tuple denoting a position to monitor the FVL. The position type (*pt*) denotes the relative position. All components in the position specification can be a wild-card, *i.e.,* *. For example, $< s_1$, ip_forward, *, AFTER> denotes that the behavior should be monitored after *ip_forward* in the switch s_1. We briefly introduce the APIs as follows.

- **Track(**fvl**,** pos**)** returns the postcards that satisfy fvl at pos and are collected after this API is invoked.
- **Assert(**fvl**,** pos**)** returns the postcards that do not satisfy fvl at pos and are collected after this API is invoked.
- **Query(**fvl**,** pos**)** returns the previously-collected postcards that satisfy fvl at pos.

Features and applications of KeyVisor. With the support of KeyTracker and KeyAnalyzer, KeyVisor is able to supply two features, *i.e.,* fine-grained visibility and high coverage, which are useful for various troubleshooting tasks. We will introduce these features as well as three sample applications.

(1) Fine-grained visibility. For KeyVisor, the fine-grained visibility denotes that operators can get the direct observation on what happens to packets at each MAT of a switch. This visibility comes with various benefits, such as finding more hard-to-catch abnormalities and fast bug location. Based on the visibility, we present two sample applications. The first one is a general verification tool similar to ASSERT-P4 [9]. ASSERT-P4 inserts assert annotations anywhere in P4 programs and exploits Symbolic Execution [30] to conduct static verification on P4 programs. The assertion-based approach is suitable for detecting network misbehaviors. However, it is hard to use ASSERT-P4 at runtime, as it cannot provide any visibility into real switches. KeySight makes *Dynamic ASSERT-P4* available, which could attribute the success to the fine-grained visibility. The second one, *Blackhole Detector*, tries to solve a wildly-concerned problem, *i.e.,* packet loss. KeyVisor can detect the packets that are proactively and consistently dropped by switches due to security policies or queue overflow, as *KeyTracker* also generates postcards for the lost packets. KeyVisor is good at detecting packet loss that can be sensed by switches. However, as KeyTracker does not generate postcards for every packet. Thus it is hard for KeyVisor to detect transient silent packet drops, which might be caused by faulty links and are almost impossible to be sensed by switches.

(2) High coverage. KeyVisor could oversee almost all packet behaviors. With this feature, KeyVisor can solve some long-standing problems, such as *stateful behavior verification*. We employ *TCP State Tracer* to exemplify this feature. As KeyTracker could generate postcards for packets that belong to the same TCP flow but are processed under different connection states, *TCP State Tracer* can faithfully record TCP connection states in stateful packet processing functions, such as stateful TCP firewall [31]. Then, the connection state records in *TCP State Tracer* enable verifying the correctness of state transitions.

V. Evaluation

A. Evaluation Overview

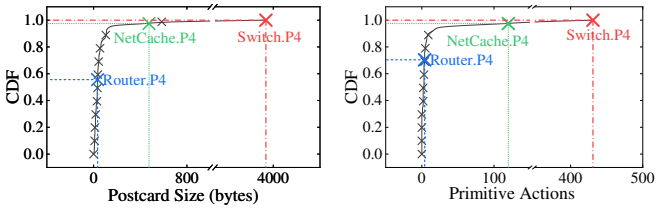
Implementation. We implement KeyAnalyzer with HLIR [32] which provides intermediate representations of P4 programs. KeyAnalyzer has 1000 lines of python code and supports both $P4_{14}$ and $P4_{16}$. KeyAnalyzer automatically generates P4 code for KeyTracker and postcard parsing code for KeyVisor. We implement a simulation model of KeyTracker with 3000 lines of C code to facilitate testing KeyTracker. KeyVisor is implemented with 3000 lines of C++ code.

Evaluation metrics and setup. We evaluate KeyAnalyzer and KeyTracker with different metrics. (1) For KeyAnalyzer, we measure postcard sizes and field replication actions of different P4 programs. (2) For KeyTracker, we measure false positive rates (FPR) and false negative rates (FNR) of RBF and other algorithms. Then, we test KeyTracker in terms of coverage, scalability, and performance overhead. We deploy KeyTracker on a 3.2T Tofino switch and a P4 SmartNIC [21], and use MoonGen [33] as the packet generator. We collect 80 open source P4 programs from Github to test KeyAnalyzer. We employ over 5 TB packet traces from CAIDA [34] and MAWI [35] to show feasibility and generality of KeyTracker.

The code of KeySight, a list of tested P4 programs, and relevant packet traces are published at [22].

B. Evaluation of KeyAnalyzer

KeyAnalyzer takes P4 programs as input and generates the PEC representation as well as the modified P4 program (to execute field replication). We evaluate KeyAnalyzer against 80 P4 programs that are open-sourced at GitHub and present two outputs of KeyAnalyzer in Figure 9, *i.e.,* postcard sizes and field replication actions.



(a) Postcard Size Distribution (b) Replication Action Distribution

Figure 9. **Distribution of postcard sizes and field replication action numbers.**

Postcard size. Figure 9(a) shows the cumulative distribution function (CDF) of postcard sizes. Postcard sizes of over 90% programs are smaller than 96 bytes. As is shown by the red line, the largest postcard comes from Switch.P4 which is the most complex P4 program in our knowledge, and the postcard of Switch.P4 has 3937 bytes, while the second biggest one only has 585 bytes. *Along with the large postcard is the ability of KeySight to finely monitor how each of the 129 MATs processes packets in Switch.P4*. The largest postcard can be carried by jumbo frames (up to 9K bytes), which has been widely supported by the state-of-the-art switches and NICs. This implies that we do not need to care about fragmenting postcards until now. Further, the postcard of Router.P4 has 35 bytes, and the postcard of NetCache.P4 has 476 bytes.

Field replication. Figure 9(b) shows the CDF of field replication actions that are added into original P4 programs. Field replication actions require additional ALU resources and modification of P4 programs. Over 90% programs need no more than nine field replication actions, and only three programs need to add over one hundred actions. Switch.P4 needs 431 actions, while Switch.P4 itself has 1693 primitive actions. These field replication actions bring about additional 25% ALU resource usage. Further, KeyAnalyzer is implemented to automate modification of P4 programs, which prevents field replication actions to be a worry for operators.

Summary. KeySight encounters issues of flatted postcard sizes, which inevitably increases postcard load between KeyTracker and KeyVisor. An efficient network troubleshooting platform should not only tell which packets are wrongly processed but also reveal what causes these faults. The two goals are hard to achieve with little information of packet behaviors, especially in programmable switches whose packet processing logic is mutable and potentially complicated. There is a trade-off between debugging information and postcard load. We give preference to debugging information to handle the complexity of programmable switches and to aim at powerful network troubleshooting. Fortunately, postcard sizes of most P4 programs are acceptable, and we optimize postcard load via decreasing the numbers of postcards.

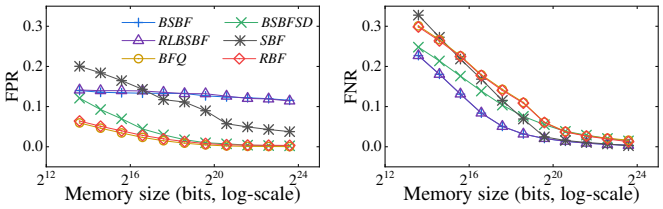
C. FNR and FPR of RBF

We compare RBF against BFQ and other four widely-used algorithms, including SBF [28], RSBF, BSBFSD, and RLBSBF [29]. Figure 10 shows the parameters of the six algorithms. All algorithms are tested against the CAIDA trace to attain distinct five-tuples (over a billion packets with about

	Parameters	BSBF/BSBFSD/RLBSBF	SBF	BFQ	RBF
×	# of BFs (S)	1	1	4	1
×	# of Arrays per BF (K)	3	3	3	3
×	# of Bits per Array (A)	From 2^{12} to 2^{22}			
×	# of Bits per Cell	1	2	1	64
✓	# of Rings per Cell (B)	-	-	-	16
✓	# of Bits per Ring (S)	-	-	-	4
×	# of Cell per Array (M)	A	$\frac{A}{2}$	$\frac{A}{S}$	$\frac{A}{B \times S}$
✓	# of Packets per Block (N)	-	-	$\frac{A}{8}$	$\frac{A}{8}$
✓	# of Recently-arrived Packets	-	-	$N \times (S-1)$	$N \times (S-1)$

✓: Dynamically-reconfigurable ×: Not dynamically-reconfigurable

Figure 10. **Parameters of different algorithms.**



(a) Fasle Positive Rate (b) Fasle Negative Rate

Figure 11. **Comparison of different BF algorithms.**

2% distinct five-tuples). We use the TCP five-tuples instead of postcards generated by KeyAnalyzer to test the algorithms. In the experiment, we vary the memory sizes occupied by these algorithms through changing M.

FPR. Figure 11(a) shows that BFQ and RBF have much smaller FPR than the other algorithms. Even for the smallest memory size, the FPR of BFQ can be 5.9%, and the FPR of RBF is smaller than 6.4%. This property is important for KeyTracker, because programmable switches have limited memory space, and KeyTracker should reserve enough memory for original P4 programs. As the memory size increases, FPRs of RBF and BFQ can decrease to less than 0.1%. As RBF cannot clear all oldest bits, RBF has a larger FPR than BFQ. Furthermore, the other algorithms have larger FPRs, implying that they might leave out a large ratio of packet behaviors when the available memory is limited.

FNR. As is shown in Figure 11(b), FNRs of RBF and BFQ are larger than the other algorithms, and RBF presents a smaller FNR than BFQ. For the smallest memory size, the FNR of BFQ can be 30.0%, and the FPR of RBF is about 29.8%. FNRs of RBF and BFQ decrease fast with the memory size increasing and are smaller than 1.63% which is close to FNRs of other algorithms when the memory size is larger than 1.5Mb. As the packet block size (N) and the packet window size ($N \times S - N$) grow, more flows can complete in the window, leading to smaller FNRs of BFQ and RBF.

Summary. As S, B, and N can be dynamically reconfigured at runtime, RBF can be adjusted to have different FNRs and FPRs. Among these algorithms, only RBF can entirely run on programmable switches without additional overheads, and the low FPR of RBF enables high coverage of packet behaviors.

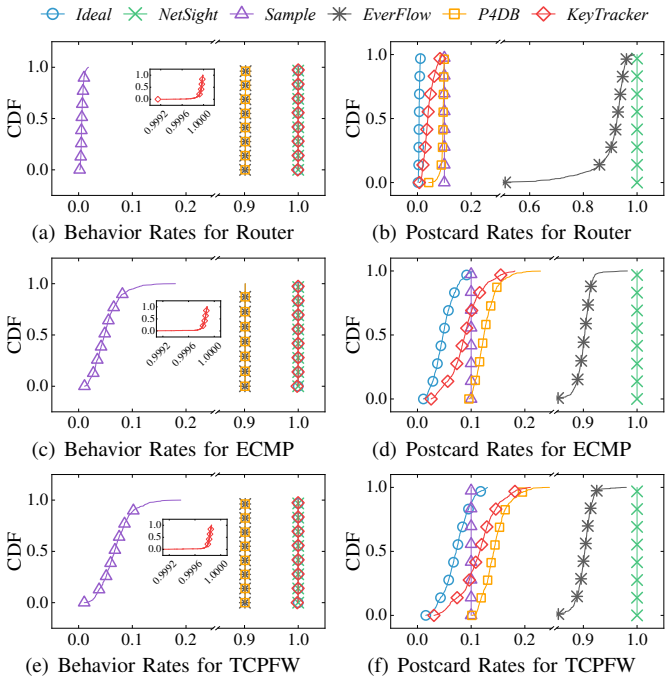


Figure 12. **Behavior rates (larger is better) and postcard rates (smaller is better) of different cases.**

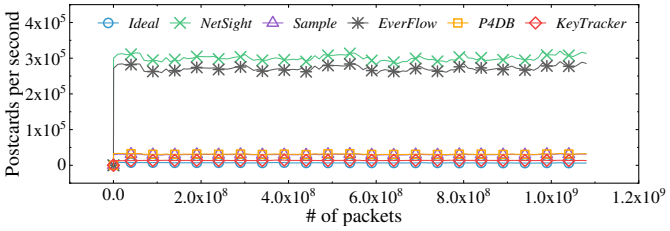


Figure 13. **Postcard generation speed of the CAIDA packet trace.**

D. Scalability and Coverage of KeyTracker

In this section, we measure postcard rates ($\frac{\text{\# of postcards}}{\text{\# of packets}}$) and behavior rates ($\frac{\text{\# of tracked behaviors}}{\text{\# of behaviors}}$) with six cases. (1) The *Ideal* baseline generates one postcard for every behavior and covers all behaviors. (2) *NetSight* generates postcards for all packets. (3) The *Sample* (stand for NetFlow) case randomly samples all packets with a fixed period to generate postcards, and we set the sampling period to 10 packets. (4) We set *EverFlow* to randomly monitor 90% of behaviors. (5) Based on *EverFlow*, *P4DB* further samples packets at the flow level to generate postcards, and the sampling period is also set to 10 packets. (6) *KeySight* occupies 600KB SRAM with $K = 3$, $B = 32$, $S = 2$, and $N = 65536$, which is modest for programmable switches owning dozens of MB SRAM. We respectively test all the cases under three functions, *i.e., Router*, *ECMP*, and *Stateful TCP Firewall* (*TCPFW*)[1]. Over 1K packet traces (each has about tens of millions of packets) from MAWI are tested, and we show CDFs for postcard rates and behavior rates of all these traces. Further, we exploit a CAIDA packet trace to test postcard generation speed of the six cases.

[1]As we have no access to the data set of real P4 MAT entries, we assume that packets with the same destination IPv4 address have same packet behaviors for *Router*; packets with the same five-tuple have same behaviors for *ECMP*; packets with the same five-tuple and TCP control flag have same behaviors for *TCPFW*.

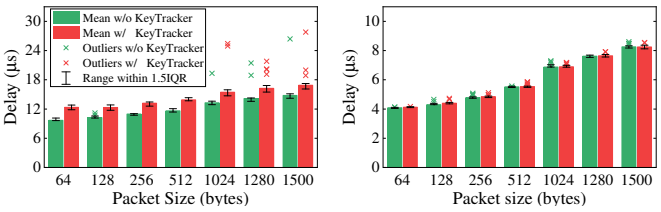


Figure 14. **Performance overhead introduced by KeyTracker.**

Coverage. We employ the rate of tracked behaviors to quantize coverage. More tracked behaviors denote better coverage. As is shown in Figure 12(a), 12(c), and 12(e), for *Ideal* and *NetSight*, the coverage is 100%. For *EverFlow* and *P4DB*, the coverage is 90%. In *Router*, behavior rates of all traces are larger than 99.915%. In the experiments of *ECMP*, *KeyTracker* keeps its behavior rate larger than 99.933% for 99% packets traces. As for *Sample*, behavior rates of all traces are lower than 20%, which is far from satisfactory.

Scalability. We employ the rate of generated postcards to quantize scalability, as more postcards require more servers and take up more bandwidth. As is shown in Figure 12(b), 12(d), and 12(f), *Ideal* shows minimal numbers of generated postcards to track all behaviors. *NetSight* needs to generate postcards for all packets, thus is the least scalable. Postcard rates of *Sample* can be subjectively adjusted, and are constantly 10%. For *EverFlow*, postcard rates of most traces are larger than 90%, which is close to *NetSight*. *P4DB* improves the scalability of *EverFlow*. For most traces, *KeyTracker* poses better scalability than the other cases, excluding the *Ideal* baseline. Further, compared with *KeyTracker*, *P4DB* comes with other significant overheads discussed in §VII.

Postcard generation speed. As is shown in Figure 13, we measure the postcard generation speed (postcards per second) of a long packet trace. The speed of *NetSight* stands for the raw packet throughput in switches, *i.e.,* about 0.3M packets per second. While the postcard load produced by *KeyTracker* is smaller than the other cases and can be less than 4% of *NetSight*. Meanwhile, *KeyTracker* can keep tracking 99.9% PECs during the experiment, which is much higher than those of *Sample*, *EverFlow*, and *P4DB*.

Summary. Based on the above analysis, we can reasonably argue that *KeySight* is closest to the *Ideal* baseline and remarkably outperforms the other cases. Existing approaches fail in either coverage or scalability, while *KeyTracker* could achieve high scalability and high coverage simultaneously.

E. Performance Overhead of KeyTracker

As KeyTracker needs to add additional MATs at the end of the P4 pipeline to generate postcards, it raises a concern that KeyTracker could bring performance overheads. To this end, we deploy KeyTracker on two P4 targets, *i.e.,* Tofino and SmartNIC, and we test the forwarding delay of *ECMP* with (w/) or without (w/o) KeyTracker against packets of varied sizes. Figure 14 shows the evaluation results. We present not

only mean delay but also interquartile range (IQR) and outliers to show whether KeyTracker affects delay jitters.

Delay. As is shown in Figure 14(a), KeyTracker introduces a modest delay increase on SmartNIC. For packets of all sizes, the delay increases by about 2μs. In most cases, packets in a network need to traverse NIC for two times, so KeyTracker, if being deployed on SmartNIC, might increase the end-to-end delay by about 4μs. We claim that this overhead is not unacceptable, even for high-performance data center networks, where the round-trip delay could be hundreds of μs to multiple ms. As is shown in Figure 14(b), KeyTracker introduces a minor delay increase of dozens of nanoseconds on Tofino.

VI. Discussion

Network-wide KeySight. In this paper, we concentrate on how KeySight performs troubleshooting tasks at the device level, while a concern on network-wide troubleshooting may raise. As postcards of KeySight take much more information about packet behaviors than those of NetSight, KeySight can adopt the similar approach to assemble network-wide packet histories with NetSight. Another concern may be that false positives of RBF could lead to incomplete network-wide packet histories. First, we can tune RBF to have a low FPR, which can mostly alleviate this concern. Further, we can also introduce probing techniques [16, 36] to remove ambiguity caused by false positives, which will be studied in the future.

Performance metrics. As KeySight generates postcards at the behavior level, some performance metrics that require monitoring packets continuously are not supported, *e.g.,* end-to-end packet-level delay. We argue that in terms of attaining these metrics, it is less economical to consistently capture every packet than other performance-centric approaches, such as network telemetry [37]. Orthogonal to the measurement tools, KeySight focuses on tracking packet behaviors and can seamlessly cooperate with them.

VII. Related Work

A. Network Troubleshooting

Network troubleshooting has long been a challenging task drawing intensive researching interests. The researching works that rely on passive packet behavior tracking, such as NetSight, EverFlow, and NetFlow, have been briefly elaborated in the previous section. Furthermore, ATPG sends test packets to proactively detect network bugs. ATPG comes with overheads of injecting massive probes into networks. The computation time for probe generation in ATPG increases exponentially with network sizes, and the coverage of ATPG is constrained by the number of servers that inject and collect probes.

With the advance of programmable switches and P4, troubleshooting programmable switches is vital for building reliable networks. Aiming at debugging programmable data planes, P4DB is the closest work to KeySight, but there are two main differences. (1) P4DB inserts debugging snippets into P4 programs to enable on-the-fly debugging, but it requires dynamic updating of P4 programs at runtime, which is unpractical for most networks. KeyTracker is deployed with the P4 programs and requires no further updating. (2) P4DB relies on the match-mirror design to track packet behaviors and uses flow-level counters to reduce the postcard load, which triggers the coverage issue and requires a large amount of memory, while KeySight intelligently tracks packet behaviors with RBF whose memory usage is bounded.

B. Network Verification

Many tools perform verification on network beliefs based on a snapshot of network states. SymNet [38] and HSA [27] perform static verification over the network-wide configurations. VeriFlow [17] and NetPlumber [39] achieve real-time verification through various optimization approaches, such as incremental updating. Most of these tools assume the functions of switches are fixed and limited. Thus, it is hard for them to model packet behaviors in programmable switches, which disables them to verify programmable switches directly.

Recognizing challenges brought by P4, the research community proposes several dedicated tools to conduct verification on P4 programs. These tools convert P4 programs to existing models and exploit formal methods to check the correctness of P4 programs at compile time. Nick *et al.* [40] propose to automatically convert P4 programs to NoD [41] programs and to verify reachability and packet well-formedness. ASSERT-P4 [9] inserts assert annotations in P4 programs and generates a C language model, then Symbolic Execution is used to verify specified properties of P4 programs. P4Pktgen [8] also uses Symbolic Execution to test P4 programs and tool-chains. These tools symbolically run P4 programs on the dedicated models and are unable to reveal how switches process packets. Thus, they cannot deal with the bugs such as hardware failures (*e.g.,* bit flips [42]) and software bugs, which are also common concerns in real networks [16]. KeySight brings fine-grained visibility into the switches at runtime, posing a powerful ability to diagnose bugs.

VIII. Conclusion

This paper presents KeySight, an efficient troubleshooting platform based on packet behavior tracking. KeySight employs the PEC abstraction to generate postcards at the behavior level, which brings a remarkable decrease of postcard load while keeping high coverage of packet behaviors. KeySight provides an automatic analyzer to extract PEC representations from P4 programs and RBF to implement the PEC abstraction entirely on programmable switches. Evaluation with real packet traces and P4 programs shows that KeySight can keep overseeing over 99.9% packet behaviors and reduces postcard load by one to two orders of magnitude.

Acknowledgement

This research is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (No.61472213). Jun Bi is the corresponding author. We thank Chen Sun, Zhilong Zheng, Yiran Zhang, Yunsenxiao Lin, and Heng Yu for their insightful suggestions. We gratefully thank our shepherd, Prof. Xin Jin, and all anonymous reviewers for their constructive comments.

REFERENCES

[1] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 99–110.

[2] Intel, "Intel flexpipe," Website, https://goo.gl/oJjkki.

[3] Barefoot Networks, "Barefoot tofino switch," Website, https://barefootnetworks.com/technology/.

[4] X. Jin *et al.*, "Netchain: Scale-free sub-rtt coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018, pp. 35–49.

[5] X. Jin *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 121–136.

[6] S. Narayana *et al.*, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 85–98.

[7] A. Gupta *et al.*, "Sonata: Query-driven network telemetry," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, 2018.

[8] A. Nötzli *et al.*, "P4pktgen: Automated test case generation for p4 programs," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 5:1–5:7.

[9] L. Freire *et al.*, "Uncovering bugs in p4 programs with assertion-based verification," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 4:1–4:7.

[10] C. Zhang *et al.*, "P4db: On-the-fly debugging of the programmable data plane," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, Oct. 2017, pp. 1–10.

[11] Google, "Google compute engine incident no.16007. connectivity issues in all regions," Website, https://goo.gl/rNW5Mr.

[12] A. Roy *et al.*, "Inside the social network's (datacenter) network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 123–137, Aug. 2015.

[13] N. Handigol *et al.*, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, 2014.

[14] B. Claise, "Cisco systems netflow services export version 9," Website, http://www.rfc-editor.org/rfc/rfc3954.txt.

[15] Y. Zhu *et al.*, "Packet-level telemetry in large datacenter networks," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 479–491, Aug. 2015.

[16] H. Zeng *et al.*, "Automatic test packet generation," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 554–566, Apr. 2014.

[17] A. Khurshid *et al.*, "Veriflow: Verifying network-wide invariants in real time," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 49–54.

[18] X. Jin, "Netcache in p4," Website, https://github.com/netx-repo/netcache-p4.

[19] The P4 Language Consortium, "Consolidated switch repo (api, sai and nettlink)," Website, https://github.com/p4lang/switch.

[20] L. Jose *et al.*, "Compiling packet programs to reconfigurable switches," in *Usenix Conference on Networked Systems Design and Implementation*, 2015, pp. 103–115.

[21] Netronome., "Agilio cx 2x10gbe," Website, https://www.netronome.com/products/agilio-cx/.

[22] "The code of keysight," Website, https://github.com/KeySight-P4.

[23] V. Sivaraman *et al.*, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 164–176.

[24] "A technique report for keysight," Website, https://github.com/KeySight-P4/keysight-report.

[25] S. Luo *et al.*, "Swing state: Consistent updates for stateful and programmable data planes," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 115–121.

[26] The P4 Language Consortium, "Router in p4," Website, https://github.com/p4lang/behavioral-model/tree/master/targets/simple_router.

[27] P. Kazemian *et al.*, "Header space analysis: Static checking for networks," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 113–126.

[28] F. Deng *et al.*, "Approximately detecting duplicates for streaming data using stable bloom filters," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, USA: ACM Press, 2006, p. 25.

[29] S. K. Bera *et al.*, "Advanced bloom filter based algorithms for efficient approximate data de-duplication in streams," *CoRR*, vol. abs/1212.3, p. 41, dec 2012.

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

[31] C. Sun *et al.*, "Sdpa: Toward a stateful data plane in software-defined networking," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3294–3308, Dec 2017.

[32] The P4 Language Consortium, "P4 high level intermediate representation," Website, https://github.com/p4lang/p4-hlir.git.

[33] P. Emmerich *et al.*, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: ACM, 2015, pp. 275–287.

[34] "The caida ucsd anonymized internet traces - chicago 2014-03-20," Website, http://www.caida.org/data/passive/passive_2014_dataset.xml.

[35] WIDE Project, "Mawi working group traffic archive," Website, http://mawi.wide.ad.jp/mawi/.

[36] F. Aubry *et al.*, "Scmon: Leveraging segment routing to improve network monitoring," in *IEEE INFOCOM 2016 - the IEEE International Conference on Computer Communications*, 2016, pp. 1–9.

[37] C. Kim *et al.*, "In-band network telemetry via programmable dataplanes," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15, 2015, pp. 2–3.

[38] R. Stoenescu *et al.*, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 314–327.

[39] P. Kazemian *et al.*, "Real time network policy checking using header space analysis," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 99–112.

[40] N. Mckeown *et al.*, "Automatically verifying reachability and well-formedness in p4 networks," 2016.

[41] N. P. Lopes *et al.*, "Checking beliefs in dynamic networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 499–512.

[42] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 139–152.