Probabilistic Profiling of Stateful Data Planes for Adversarial Testing

Qiao Kang
Rice University
Houston, TX, USA

Jiarong Xing
Rice University
Houston, TX, USA

Yiming Qiu
Rice University
Houston, TX, USA

Ang Chen
Rice University
Houston, TX, USA

ABSTRACT

Recently, there is a flurry of projects that develop *data plane systems* in programmable switches, and these systems perform far more sophisticated processing than simply deciding a packet's next hop (i.e., traditional forwarding). This presents challenges to existing network program profilers, which are developed primarily to handle stateless forwarding programs.

We develop P4wn, a program profiler that can analyze program behaviors of stateful data plane systems; it captures the fact that these systems process packets differently based on program state, which in turn depends on the underlying stochastic traffic pattern. Whereas existing profilers can only analyze *stateless* network processing, P4wn can analyze *stateful* processing behaviors and their respective *probabilities*. Although program profilers have general applications, we showcase a concrete use case in detail: adversarial testing. Unlike regular program testing, adversarial testing distinguishes and specifically stresses low-probability edge cases in a program. Our evaluation shows that P4wn can analyze complex programs that existing tools cannot handle, and that it can effectively identify edge-case traces.

CCS CONCEPTS

• **Networks** → **Network security**; • **Software and its engineering** → **Software verification and validation**.

KEYWORDS

Programmable data planes, symbolic execution, adversarial testing

ACM Reference Format:
Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. 2021. Probabilistic Profiling of Stateful Data Planes for Adversarial Testing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3445814.3446764

Kang and Xing contributed to this work equally.

ASPLOS '21, April 19–23, 2021, Virtual, USA

ACM ISBN 978-1-4503-8317-2/21/04…$15.00
https://doi.org/10.1145/3445814.3446764

1 INTRODUCTION

One recent advance in networking technology is the design of programmable data planes. Unlike traditional networking devices, emerging hardware can be reprogrammed in P4 for customized packet processing [17]. This has motivated many in-network functions to be developed in the switch data plane, such as caching [44], load balancing [47, 60], link failure detection [38], and security [46, 75]. These *data plane systems* make far more sophisticated decisions in-network than just computing a packet's next hop (i.e., traditional *forwarding*).

Recent projects have developed program profiling support for P4 programs [23, 24, 55, 61, 69, 71] in order to achieve a comprehensive understanding of complex program behaviors; however, existing profilers suffer from two key limitations. First, they only perform *stateless* analysis for simple forwarding programs, but cannot handle *stateful data plane systems* [38, 44, 46, 47, 60, 75]. Stateful P4 programs extract state from historical traffic, and they process packets differently when program state changes. Consider the Blink [38] link failure detector. It randomly samples a set of TCP flows, tracks per-flow retransmissions, keeps a sliding window for monitoring, and activates backup paths in a round-robin manner. Existing profilers [24, 61, 71] cannot track program state, so they are fundamentally handicapped in analyzing fine-grained stateful processing in data plane systems.

Second, network processing is always non-deterministic in nature, e.g., due to traffic changes, load balancing, and failures; for stateful P4 programs, their probabilistic nature is even more fundamental. Their internal state may take a distribution of possible values depending on past traffic patterns, and this will further influence future processing behaviors in a stochastic manner. In recent years, the networking community has been paying increased attention to probabilistic analysis [33, 66, 67, 70, 73], e.g., analyzing network failure probabilities [70] or the likelihood for load violation [73]. However, this line of work so far has only considered network configurations [70, 73] and new probabilistic network languages [33, 66, 67]; neither can analyze today's data plane programs.

We develop P4wn, a program profiler that addresses both limitations. P4wn efficiently analyzes how stateful P4 programs change behaviors over a packet sequence, characterizes the probability for each processing behavior, and generates concrete test traces for validation. This new capability is useful for many applications. *(a) Stateful testing:* Existing tools [24, 71] are restricted to stateless program testing, but P4wn can perform stateful testing of complex P4 programs. *(b) Offloading hints:* Recent systems partition network functions between servers and a P4 switch for accelerated performance [50, 77]; P4wn can pinpoint frequent "hotspots" for offloading, and/or reason about probabilistic SLA properties. *(c)*

Anomaly detection: Operators can profile common- and corner-case network behaviors to catch and guard against unexpected events. We demonstrate a concrete use case—*adversarial testing*—which combines *(a)* and *(c)*, and provide preliminary results on *(b)*.

Unlike basic program testing, adversarial testing distinguishes and specifically focuses on edge cases [10, 18, 37, 39, 41, 42, 49, 52, 53, 63, 74]. Data plane systems—and packet processing programs in general—are particularly vulnerable to adversarial inputs, as even unprivileged network clients can influence the input traffic pattern; as an example, an adversarial trace for Blink might be one that contains high-volume TCP retransmissions. This risk has been recognized by the research community in multiple recent projects [37, 45, 59, 63]; but so far, data plane system designers have resorted to manual analysis for adversarial testing (e.g., in Blink [38] and others [46, 75]). P4wn contributes an automated technique towards this goal—a probabilistic profiler is a natural fit for pinpointing edge cases, and it can further analyze how likely each case would occur, and precisely generate packet sequences to exercise them.

Challenges and techniques. P4wn uses symbolic execution (symbex) as a starting point for program analysis. Unlike program fuzzing or dynamic testing, which generates random or concrete inputs to a program but cannot provide coverage guarantees, symbex is a static analysis technique that enumerates program execution paths exhaustively for high coverage [19, 24, 64, 71]. However, existing P4 symbex tools can only perform stateless analysis [24, 31, 71]. In order to achieve our goal, P4wn needs to take a few more steps forward—it performs *probabilistic* analysis of *stateful* programs with *complex data structures*. This requires a range of new techniques to be developed.

The first challenge we tackle is *computing probabilities*. Existing P4 symbex engines [24, 71] target qualitative analyses, i.e., they check whether a program behavior occurs or not, but cannot analyze how likely a behavior would occur under a certain network scenario. P4wn builds upon an advanced form of symbex, known as *probabilistic symbex* [35], and draws inspiration from two threads of work: header space analysis (HSA) [48], and model counting [20]. P4wn first relies on traditional symbex to collect header constraints for program execution paths. It then analyzes the resulting header space enclosed by these constraints, using model counting to compute the volume of this multi-dimensional polytope. Given a set of constraints, SAT/SMT solvers can produce a concrete set of satisfying assignments; model counting solvers, on the other hand, can *count* the number of all possible satisfying assignments. Dividing this count over the size of the solution space would yield the probability for the given set of constraints to hold. As another twist, the header space for most real-world networks is not evenly distributed (e.g., more TCP traffic than UDP), and the distribution could further vary from one network to another. Since such information cannot be obtained from the solver, P4wn instead formulates *interactive queries* about the target network at runtime and issue these queries to an "oracle" instead of a solver, which could be a simple spec, a human analyst, or a trace she has collected.

The second challenge we address is that data plane programs have very complex *state*. Existing P4 symbex engines, on the other hand, can only perform stateless analysis of simple programs [24, 31, 61, 71], where the processing behaviors do not depend on historical traffic. Since stateful programs may change behaviors per packet, P4wn needs to profile a program using a sequence of symbolic packets, effectively simulating a loop where one symbolic packet is analyzed per iteration [26, 63]. However, loops are notoriously hard to handle, because each iteration could fork multiple paths, and the size of the symbex state would grow exponentially. Since data plane programs are designed to process "infinite" packet sequences, some code blocks require very long sequences to exercise. Moreover, we do not know in advance how long this sequence should be. To address this, P4wn checks in real time whether the current profile has converged, and selectively refines the unconverged portion of the profile further. We also propose a new technique called *telescoping*, which can handle a common class of "deep" code blocks in data plane systems. It can detect stateful but periodic program behaviors using a short sequence, and generalize that behavior to a much longer sequence for program analysis.

P4wn also handles *approximate data structures* common to data plane systems, such as hash tables, Bloom filters, and sketches, none of which is supported by existing tools [24, 71]. These data structures are analogous to arrays or sets, but they use a very large state space to reduce inaccuracy. One solution is to handle them as symbolic arrays, by modeling the uncertainty of the locations and values of reads/writes [19, 32], but this scales very poorly with the array size. Furthermore, approximate data structures heavily rely on hash functions for computing indexes of reads/writes; this creates complicated constraints that cannot be solved efficiently. We develop a novel technique called *greybox analysis*, leveraging the insight that these data structures have well-established statistical properties (e.g., collision rates of CRC functions). This enables P4wn to analyze their probabilities without tracking every state variable and read/write operations, so the resulting profiling algorithm scales independently of the data structure size.

We detail these challenges and solutions in the rest of this paper, and present a comprehensive evaluation.

2 MOTIVATION

Observing that programmable data plane behaviors are governed by complex programs, researchers have taken significant interest in customizing program profiling and analysis techniques for the network data plane [24, 55, 61, 71, 72]. In particular, several tools [24, 61, 71] leverage symbolic execution to perform *stateless* analysis of P4 programs.

2.1 Limitations of existing work

Existing P4 symbex tools suffer from two limitations.

Stateful analysis. We have seen a flurry of *data plane systems* that perform sophisticated *stateful* processing in-network, which are far more complex than stateless forwarding programs. Data plane systems are essentially an infinite loop of the following procedure. They 1) accept a network packet p_i as input, and 2) generate a decision d_i based on the headers of p_i and the current state of the switch s_i. The decision further includes two parts: a) some action on the packet, e.g., forwarding to a certain port, and b) some modification to the switch state. Since the state s_i accumulates over the sequence of packets p_0–p_{i-1}, the decision d_i would

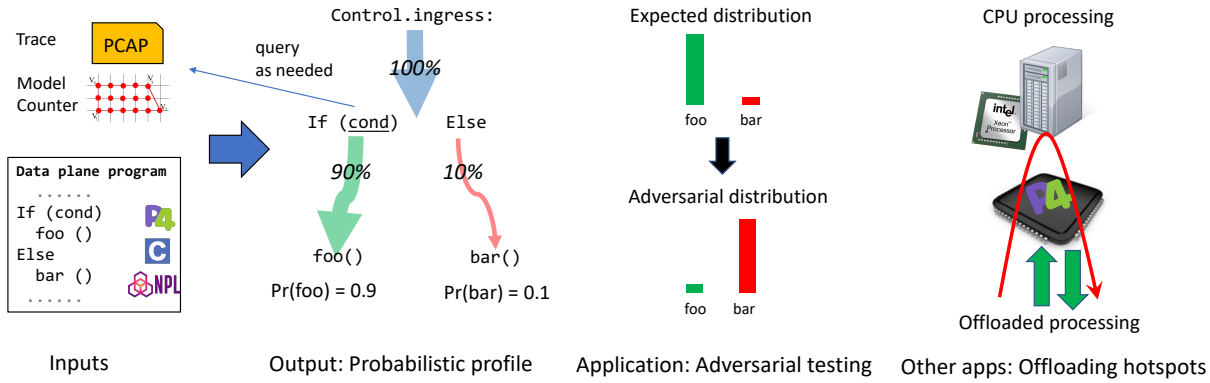


Figure 1: The workflow of our P4wn system and its target applications. P4wn takes a P4 program as input, and outputs its probabilistic profile. When necessary, P4wn performs interactive queries to the operator or a network trace to obtain the target network profile. Our primary application focuses on adversarial testing, but probabilistic profiling can enable other use cases, such as providing offloading hints based on program hotspots. The conceptual ideas behind P4wn are also generalizable to other data plane languages, such as NPL and eBPF/C.

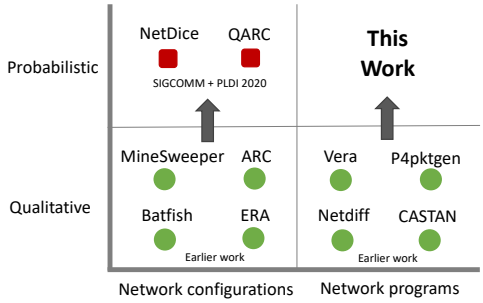


Figure 2: P4wn as compared against existing work.

similarly depend on the history of packets. This stands in stark contrast to stateless forwarding programs, which process every single packet independent of each other—i.e., decision d_i only depends on the current packet p_i; state s_i is non-existent. Existing symbex tools [24, 61, 71] only perform single-packet analysis from p_i to d_i assuming empty state. They cannot analyze program behaviors over a stateful packet sequence.

Probabilistic analysis. Second, while researchers have been actively working towards *probabilistic* program analysis [33, 66, 67, 70, 73], existing work has only considered network configuration analysis [70, 73] and designing new probabilistic network languages from scratch [33, 66, 67]. Network environments are inherently non-deterministic—traffic composition, link failures, random load balancing, and many other factors contribute to their probabilistic behavior. In the case of data plane systems, the most prominent factor is the traffic composition—the packet sequence p_0–p_i is probabilistic in nature, and this will drive s_i to different statistical distributions. However, existing profilers [24, 61, 71] for data plane programs are qualitative, and they cannot capture probabilistic properties.

2.2 Our contributions

P4wn addresses both limitations of existing profilers. It performs *probabilistic profiling* for *stateful* data plane systems. Figure 1 shows its workflow. P4wn takes in the source code of a data plane system as input, and performs program analysis to generate stateful sequences to trigger all program behaviors in a fully automated manner. P4wn can further compute the probability for each behavior, either by querying a model counting solver, or based on a given *traffic profile* that captures the traffic composition of the deployment scenario. Since different programs require knowledge about different aspects of traffic composition—e.g., retransmission ratios for Blink, but TCP ratio for others [8], asking the analyst to provide a complete profile a priori will be burdensome. P4wn draws inspiration from "oracle-guided" program synthesis [43], which obviates the need for a complete specification by allowing runtime queries to an oracle, and enables runtime *interactive queries* to discover relevant traffic composition. These queries are issued to pre-collected traffic traces, e.g., by monitoring systems common in production networks [78].

Figure 2 further positions our work against a set of existing work [13, 24, 25, 30, 36, 61, 63, 70, 71, 73] along two dimensions: a) whether a technique analyzes network configurations or network programs, and b) whether it captures the probabilistic nature of network behaviors. We note that this figure does not comprehensively show all related work: P4 program analysis tools that do not rely on symbex [55, 69], and new probabilistic network programming languages that are designed from the ground up [33, 66, 67], are not shown.

Applications. Program profilers are general utility tools that have many applications. We primarily focus on one use case: adversarial testing, which is an important program testing strategy that specifically stresses edge cases as they may lead to unexpected behaviors [10, 18, 37, 39, 41, 42, 49, 52, 53, 63, 74]. In previous work, adversarial testing of network programs has been performed using machine learning techniques [37] and execution cost aware program analysis [63]. However, for data plane systems, automated

```
function ProbProf(prog, tr)
    (ℕ, μ̂) ← Telescope(prog)
    symb.pktseq = ∅
    while !(isConverged(μ̂, α, ϵ) or isTimeout) do
        symb.pktseq += {symb_pkt}
        ℙ ← symb.EnumPaths(prog)
        (ℕ, μ̂) ← UpdateProb(ℙ, ℕ, tr)
    if !(isConverged(μ̂, α, ϵ)) then
        (ℕ, μ̂) ← SampPaths(prog, tr)
    return Sort(ℕ, μ̂)
function UpdateProb(ℙ, ℕ, tr)
    for p ∈ ℙ, N ∈ ℕ do
        if p triggers N then
            Pr[p] ← ModelCnt(p, tr)
            μ̂ ← Update(N, Pr[p])
    return (ℕ, μ̂)
function SampPaths(prog, tr)
    while !(isConverged(μ̂, α, ϵ)) or isTimeout) do
        symb.pktseq += {symb_pkt×δ}
        ℙ ← symb.InformedSamp(prog, μ̂)
        (ℕ, μ̂) ← UpdateProb(ℙ, ℕ, tr)
    return (ℕ, μ̂)
```

```
function Telescope(prog)
    ℝ ← Registers(prog)
    for r ∈ ℝ do
        if IsPeriodic(r, prog) then
            ℝ_p ← {r}
        if IsGuard(r, prog) then
            ℝ_g ← {r}
    for r ∈ ℝ_p ∩ ℝ_g do
        N_r ← r.guarded
        for p ∈ r.periodic_paths do
            prob ← ModelCnt(p, tr)
            rept ← r.thresh/r.period
            μ̂ ← Update(N_r, prob^rept)
    return (ℕ, μ̂)
function IsGuard(r, prog)
    op ← ">" | ">=" | "=="
    for each br ∈ prog.all_branches do
        if br == "r op const" then
            r.isguard ← True
            r.guarded ← br.code_blk
            r.thresh ← br.const
    return r.isguard
```

```
function IsPeriodic(r, prog)
    symb.pktseq ← {symb_pkt×γ}
    ℙ ← symb.EnumPaths(prog)
    for p ∈ ℙ do
        if p modifies r then
            pc ← symb.ObtainPC(p)
            period ← BinarySearch(pc, γ)
            if period < γ then
                r.periodic_paths ∪ = {p}
    return !isempty(r.periodic_paths)
function BinarySearch(pc, γ)
    for i ∈ [γ/2 ..1] do
        pref ← pc.Prefix(i)
        if pc repeats pref then
            period ← i
    return period
function ModelCnt(p, tr)
    pc← symb.ObtainPC(p)
    if tr ≠ ∅ then
        hdr_dist ← Query(tr, pc)
    vol ← ComputeVol(pc, hdr_dist)
return vol/ComputeVol(True, hdr_dist)
```

Figure 3: The main function ProbProf takes in a P4 program, and optionally a trace, and outputs its probabilistic profile $(\mathbb{N}, \widehat{\mu})$. P4wn performs the greybox analysis before the profiling (not shown). Functions with the prefix "symb._" are invocations to the symbex engine. The initial state of a program is empty.

support for adversarial testing has been lacking; existing work finds adversarial traces via manual analyses [38, 46, 75]. These manual efforts call for a principled approach to adversarial testing of data plane systems, as manually analyzing the systems may not produce a comprehensive result, and it could also be tedious and error-prone. In this regard, P4wn contributes a practical approach to adversarial testing of data plane systems by identifying edge cases based on probabilities. In Section 6, we also provide brief discussions on other use cases.

3 THE P4WN SYSTEM

P4wn uses symbolic execution [51] as the starting point for program analysis. The symbolic execution engine (SEE) is a special program interpreter that runs a program with symbolic input values, representing all possible concrete values that the inputs may take. As the execution proceeds, the SEE updates program variables with symbolic expressions. Unlike a regular program execution, symbolic inputs create uncertainty as to whether a loop or branch condition holds. The SEE handles this uncertainty by forking one execution path for each possible outcome. Consider a branch if (x<=7) then foo() else bar(). The SEE forks two paths and annotates each with a set of *path constraints*, representing the conditions for that path to be exercised: a) the if-branch holds when {x<=7}; and b) the else-branch holds when {x>7}. Symbolic execution finishes when all paths have been explored, or, in the case where a program is very complex, until a timeout threshold. The latter hints at a practical limitation of symbex: its analysis may not be exhaustive, unless complete enumeration is feasible within useful time. Upon exit, the SEE invokes SAT/SMT solvers to solve for concrete inputs that would satisfy the constraints for certain paths. Innovations in symbex techniques center around two fronts: a) developing *domain-specific techniques* that are customized for new problem domains, and b) mitigating the *state explosion* problem as it manifests itself in these domains. The design of P4wn involves both.

The pseudocode in Figure 3 serves as our technical roadmap. P4wn models a program *prog* as a Control Flow Graph (CFG), which is a directed graph where every node represents a code block (i.e., a sequence of program statements without branches), and every directed edge represents a branch. The program is then transformed into a CFG with t nodes, $\mathbb{N} = \{N_1, \cdots, N_t\}$. The ProbProf function takes in *prog* (and optionally, a trace tr) as the input, and outputs its probabilistic profile $(\mathbb{N}, \widehat{\mu})$, where $\widehat{\mu} = \{\Pr[N_1], \cdots, \Pr[N_t]\}$. The main while loop performs an iterative deepening search, where ProbProf uses longer and longer sequences of symbolic packets to exercise *prog*, and updates the distribution $\widehat{\mu}$ in every round of execution until it converges.

3.1 Computing probabilities

The first property that sets P4wn apart from state-of-the-art network symbex engines [24, 26, 63, 71] is its ability to perform *probabilistic* profiling for different network scenarios or traffic distributions.

ProbProf profiles the probability for a CFG node N_i by enumerating all execution paths that exercise N_i, computing the probability for each path p, and then summing them up. Initially, the symbolic packet sequence *pktseq* is empty, and it gets extended with one additional symbolic packet per loop iteration. For each iteration, we enumerate all execution paths that can be exercised by *pktseq*, and invoke UpdateProb on each path to update $\widehat{\mu}$. If a path p can trigger a CFG N_i, then it is added to N_i's pathset. To obtain $\Pr[N_i]$, we compute $\Pr[p]$ for each p and sum them up: $\Pr[N_i] = \sum_{p \in N_i.\text{pathset}} \Pr[p]$. To compute $\Pr[p]$, we rely on the path constraints *pc* collected by the symbex engine—e.g., an execution path that processes TCP SYN packets might yield $pc = \{proto == 6 \wedge syn == 1\}$. More generally, *pc* forms a *polytope* in the header space, and its enclosed points are the satisfying assignments to the constraints.

This formulation is helpful, because it allows us to compute the number of satisfying assignments without having to enumerate them all. Supposing for the moment that we do not consider concrete network traces, then P4wn can compute probabilities by invoking a *model counting solver*, an advanced variant of SAT/SMT solvers. Regular solvers can compute one set of satisfying assignments to a logical formula, but model counting solvers can count or estimate the total number of solutions. Suppose that there are t sets of satisfying assignments to pc, and that the space of all possible assignments has size T, then the probability for pc to hold can be derived as t/T. The ModelCnt algorithm is similar to a program analysis technique called probabilistic symbex [35], which applies model counting to C programs to estimate probabilities.

To handle unevenly distributed header spaces, P4wn needs to understand the specific network under consideration. In this case, ModelCnt would instead issue an interactive Query to the trace tr to discover header distributions. We then compute the polytope volume in a *skewed* multi-dimensional space, by weighing the header subspace and its volume according to the distributions. Alternatively, P4wn could take in a prespecified header distribution, and use a weighted model counting solver to compute the skewed polytope volume. Both would capture network and scenario specificity, but using a trace would enable P4wn to obviate the need for prespecifying header distributions. Of course, users of P4wn can supply an initial profile that encodes well-established facts—e.g., TCP accounts for 90% traffic in Microsoft data centers [8]; P4wn can discover the rest at runtime.

3.2 Checking convergence

The state explosion problem manifests particularly severely for *stateful* programs. A P4 program itself does not contain loops, but a stateful program accumulates state over many packets, and it may behave differently depending on the state. In other words, data plane systems implicitly run the P4 program in an *infinite loop*. Correspondingly, we need to use a sequence of symbolic packets in a loop to explore such a program [26, 63]. Scaling the analysis of a *symbolic sequence* is our focus in this subsection and the next. State-of-the-art P4 symbex engines [24, 71], only perform stateless analysis of simple *forwarding* programs. However, *data plane systems* have very complex state, so P4wn needs to handle new challenges due to stateful processing.

A stateful program may change its behavior per packet, so its probabilistic profile only stabilizes over a sequence of packets; moreover, we do not know in advance how long this sequence is. In ProbProf, the while loop may soon generate an unmanageable amount of state: if the program has k branches, then t symbolic packets would fork $O(k^t)$ paths. To determine how large t should be, P4wn relies on a statistical approach—after obtaining new execution paths in each iteration, it updates the current profile $\widehat{\mu}$ and tests whether it has changed significantly. P4wn continues to update $\widehat{\mu}$, and exits the while loop once the profile has converged.

However, if state size is very large, this loop may never exit. To handle this, P4wn uses a timeout threshold to enter a sampling phase (SampPaths). In this function, the length of $pktseq$ increases much faster (by $\delta > 1$) to trigger deep code blocks. The target of this phase is to use the current profile as a starting point, and refine the unconverged portion further. For each $pktseq$, SampPaths uses a technique called *informed sampling* [28], which leverages Bayesian Inference to draw random samples based on the prior, and then updates the posterior with the new samples. Suppose the ground truth is μ, our goal is to obtain an estimate $\widehat{\mu}$ with a confidence level α and an error bound ϵ, i.e., $\Pr[|\widehat{\mu} - \mu| < \epsilon] \geq \alpha$. ϵ and α can be further tuned for a closer approximation. Of course, if the program is of astronomical size, even this sampling might time out. This is a fundamental challenge in symbolic execution, and in this case a tool could trade off theoretical guarantees for practical execution time.

3.3 Telescoping "deep" code blocks

When designing P4wn, we encountered a class of data plane program elements that are very hard to analyze, because they can only be exercised after a very long packet sequence (i.e., very large t). To see why, consider Blink [38]: it monitors 64 TCP connections and triggers rerouting if more than 32 connections experience retransmission. This code block, N_{reroute}, is guarded by a conditional statement that contains a counter: if (retrans_cnt > 32). To trigger N_{reroute}, we need at least 64 packets (i.e., 32 retransmissions), and this would fork $O(2^{64})$ execution paths. As another example, consider a very simple (but stateful) program that processes every millionth packet specially (e.g., sampling to the CPU); N_{cpu} cannot be exercised unless we symbex one million packets (i.e., $t = 1M$). In other words, data plane systems are designed to process millions of packets, so some code blocks are guarded by extremely large thresholds.

Our key insight for avoiding state explosion is that, fortunately, these deep loops tend to have regularity. For the above examples, the processing of the pkt_i does not directly depend on the individual packets pkt_0–pkt_{i-1}. Rather, it only depends on a succinct history of the previous packets—typically a monitoring state that counts the occurrences of the same event. This results in execution paths with *periodicity*. For Blink, pkt_0 and pkt_1 need to have identical headers (i.e., retransmissions), and the same goes for pkt_2 and pkt_3. In other words, there exists an execution path whose path constraints have repeatable patterns: $pc = \{pkt_0.headers == pkt_1.headers \wedge pkt_2.headers == pkt_3.headers \wedge \cdots\}$.

We develop a technique called *telescoping* (Telescope). At the heart of this algorithm is IsPeriodic that probes the program with a short packet sequence of length γ, and analyzes the resulting paths to detect periodicity in their constraints. It performs a BinarySearch on a path's pc, and identifies the shortest repeatable pattern, or the *period* of the path. If more than one paths are periodic, IsPeriodic identifies all of them. The output of this step is $\mathbb{R}_p$, a set of registers (P4 parlance for state variables) that increment linearly over repeatable packet sequences. Telescope then intersects $\mathbb{R}_p$ with $\mathbb{R}_g$, which is the set of registers used as conditional guards. The guarded code blocks are the target of telescoping, for which we want to obtain a probabilistic profile without a full symbex.

As the second for-loop in Telescope shows, P4wn computes the probability for each periodic path when "stretched" far enough to trigger the target, and uses their sum as the final estimate. We note on the potential accuracy loss: a) if there are periodic paths $path_1, path_2, \cdots$, the telescoping algorithm cannot tell whether

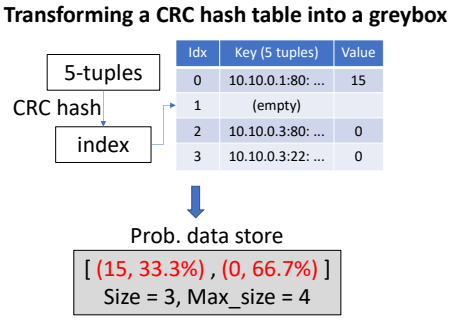


Figure 4: We create a probabilistic data store for each hash table, which encodes the distribution of all possible values in this greybox. This avoids the need for keeping track of a large symbolic state space and analyzing complex CRC functions, improving scalability.

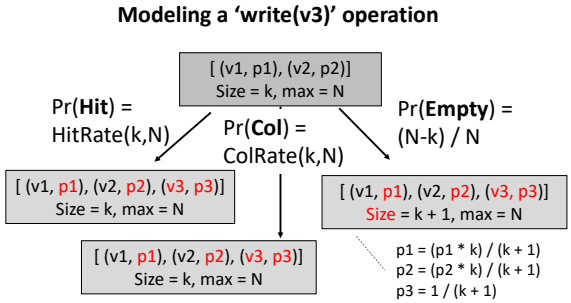


Figure 5: For each access to the probabilistic data store, P4wn forks three paths and updates the probabilities using the current state of the data store. A similar analysis is performed for sketches and Bloom filters.

alternating appearances of $path_1, path_2, \cdots$ can similarly move us towards the target; determining this would again require analysis using a much longer packet sequence, e.g., $path_1||path_2$; b) it also cannot identify aperiodic paths that may also trigger the target. As a result, telescoping may result in an underestimation of the target probability.

3.4 Approximate data structures

The next challenge we address is *approximate data structures*, such as Bloom filters, sketches, and hash tables. These data structures trade off accuracy for space efficiency, and have found widespread use in data plane systems. Under the hood, they rely on a set of arrays and a set of CRC hash functions to compute indexes to these arrays. For instance, a count-min sketch may hash a flow ID using k CRC-16 functions, use the hash values as indexes to retrieve k counters, and return their minimum. The underlying arrays can cause state explosion; the hash functions would produce very complicated constraints to solve for or model count. Existing P4 profilers [24, 61, 71] cannot handle them.

Symbex engines like KLEE symbolically analyze arrays by accounting for the uncertainty of read/write locations and values, e.g., using the theory of arrays or by forking execution paths [19, 32]. This scales very poorly with the array size. In data plane systems, these data structures need to have a large size to reduce inaccuracy—e.g., a CRC-16 hash table has 2^{16} state. On the other hand, hash functions are usually handled using a technique called *havocing* [11, 63], which treats a hash function as a blackbox without collecting its constraints. It creates a fresh symbolic variable as the hash output, and continues the symbex with this new variable without representing the relation between the hash input and output. After the symbex finishes, it uses a rainbow table [62] to connect an input with its output. While havocing is enough for generating a concrete execution, it does not produce path constraints that we need for profiling.

CRC hash functions, moreover, produce non-linear constraints that are notoriously difficult to model count [16]. Unlike SAT/SMT solvers, which are quite mature today (e.g., Z3 [22]), model counters (also known as #SAT/#SMT solvers) are still in their infancy. Today's model counters can only handle simple #SAT problems [20] and a restricted number of #SMT theories [2, 56, 57]. Therefore, even if we have constraints for the CRC functions, efficiently solving them would first require more advances in model counting.

We develop a new technique called *greybox analysis*, leveraging the observation that these data structures have well-established statistical properties. P4wn only tracks necessary state for computing probabilities—i.e., the value *distribution* of the data structure—while abstracting away all possible permutations of these values and their indexes. Consider the hash table in Figure 4. P4wn creates a symbolic representation that we call a *probabilistic data store*. Internally, it contains a set of (v_i, p_i) tuples, where p_i is the probability for the value v_i to appear in the table, and $\sum_i p_i = 1$. The data store also maintains the number of active entries in the table, which P4wn uses to compute the statistical properties of the table, e.g., collision rates. For each access to the table, P4wn forks three paths: a) hitting an empty entry, b) hitting an existing entry, and c) colliding on an existing entry; and it computes the probability for each path using the current distribution of the probabilistic data store. P4wn further updates the distribution and the number of entries after a write: a) would increase the number of existing entries by one, whereas b) and c) would maintain the same number of entries in the table. Figure 5 shows a write operation with a new value v_3.

P4wn handles sketches and Bloom filters using similar techniques. For a count-min sketch with k hash tables, P4wn creates a probabilistic data store for each table, and computes the distribution of the minimum value based on the distributions of the underlying data stores. For Bloom filters, P4wn keeps track of the total number of bits N and the number of insertions k. A membership test on a Bloom filter results in only two paths, one for miss and another for hit—their probabilities are determined mathematically by N and k.

3.5 Generating test packet sequences

So far, we have discussed how P4wn generates a probabilistic profile for a stateful program. Next, we describe how P4wn generates a concrete packet sequence to trigger a desired code block, e.g., when concrete packets are desired for testing. Compared to stateless testing [61, 71], which only needs to generate one single test packet to trigger a program path, P4wn needs to address new challenges

due to stateful sequence generation. P4wn invokes a SAT/SMT solver one more time using the path constraints collected over the symbolic sequence. However, our greybox analysis has abstracted away the internal operations for the corresponding CFGs, so P4wn does not have path constraints to solve for. (In the case where the program is too complex to exhaustively analyze, there may also exist a few CFGs that remain unexplored in the profiling phase.) P4wn recovers the missing path constraints for these CFGs using another round of "lightweight" symbex. Since the goal here is *not* comprehensive enumeration any more, we can afford to drive the direction of exploration in a very precise manner using much longer packet sequences and pruning unfruitful forks aggressively.

P4wn uses two techniques: *directed symbex* [58] and *havocing* [11, 63]. Directed symbex can guide the exploration towards more likely paths towards a target. It measures the "distance" from the current state to the target code block by the number of edges in the CFG. Then, it dives into the shortest-distance path first, and checks whether the target block is reachable via this path. If this exploration fails, it backtracks to the previous state, and picks a slightly longer path to explore. Empirically, this algorithm terminates very fast, as the search space is much smaller than an exhaustive enumeration. Havocing [11, 63], on the other hand, becomes handy in this step, because one concrete sequence will be sufficient.

4 IMPLEMENTATION

We have implemented a prototype of P4wn in 6500 lines of code in C++ as pluggable modules in KLEE [19], an industry-strength symbex engine that has been used in more than 100 projects [5]. The source code is available at: https://github.com/qiaokang92/P4wn.

Our prototype consists of four components: a) a profiler, b) an interactive query processor, c) a test workload generator, and d) a backtesting engine. The *profiler* uses LattE [2] for model counting support, which can compute the volume of multi-dimensional polytopes. It can be configured with a time budget for symbex, and can output a ranked list of CFGs with probabilities. The *query processor* can generate queries on packet (header or timestamp) distributions to human operators or a pcap trace. It loads the trace when P4wn starts, and pins the trace in memory throughout the analysis. It also caches and reuses previous query results. The *workload generator* produces concrete sequences that trigger target code blocks, and converts the resulting KTEST files generated by KLEE to pcap traces. The *backtesting engine* is a script that can replay pcap traces to a DUT (device under test), which can be bmv2 [4] P4 switches or Tofino hardware switches. Our engine can optionally collect the number of bytes received and sent at each port, link utilization over time, and CPU+memory usage at the switch control plane; operators can also easily add metrics of their own. Since P4 is a recent language, all previous P4 symbex projects need to translate P4 into some existing language that today's symbex engines can support [24, 31, 61, 71]; we follow the same approach and translate programs to C [31].

5 EVALUATION

Our evaluation seeks to answer three high-level questions: a) How scalable is P4wn in analyzing *stateful* programs? b) How effectively can P4wn perform *probabilistic* profiling? and c) How effectively can P4wn perform *adversarial testing* for complex, real-world data plane systems?

Programs tested. We will first briefly show that P4wn naturally supports *forwarding* programs, i.e., those used by a state-of-the-art profiler, Vera [71], for evaluation. Our focus, however, will be a set of more complex stateful programs that Vera cannot analyze. Four of them are programs from the P4 repository.

- S1 (lb.p4): A simple load balancer.
- S2 (flowlet.p4): A flowlet switching program that mitigates out-of-order delivery.
- S3 (nat.p4): Network address translator.
- S4 (acl.p4): Network access control program.

Seven programs are from recent research projects. They are among the most complex data plane systems at the time of writing, and cover popular use cases including network monitoring, security, and application-level acceleration. We found that S2, S5, S7, S9, and S10 use CRC hash tables; S6, S8, and S10 contain Bloom filters; S6 and S11 use count-min sketches; and S5, S6, and S11 contain deep state.

- S5 (Blink [38]): Remote link failure detection.
- S6 (NetCache [44]): In-network key/value cache.
- S7 (*Flow [68]): Network telemetry that provides richer information than Netflow.
- S8 (p40f [9]): Passive OS fingerprinting inside the switch for network security.
- S9 (NetHCF [54]): Hopcount-based detection and filtering of spoofed traffic.
- S10 (Poise [46]): Context-aware access control for enterprise networks.
- S11 (NetWarden [75]): A defense system against network covert channels.

We have also created four stateful benchmarks:

- S12 (counter.p4): Counts the number of TCP and UDP packets, sampling each kind periodically.
- S13 (htable.p4): Monitors (exact) flow sizes using a CRC hash table.
- S14 (cmsketch.p4): Monitors (approximate) flow sizes using a count-min sketch.
- S15 (bfilter.p4): Monitors the existence of certain header values using a Bloom filter.

Appendix A.1 includes more description of these systems.

Setup. By default, we have used the CAIDA trace [1] as the traffic profile, with the following exceptions: a) S10 (Poise [46]) uses used-defined protocols that are specific to the access control scenario, b) S11 (NetWarden [75]) specifically targets network covert channels in file transfers, and c) S6 (NetCache [44]) is customized for key/value workloads with a certain skew. We have obtained the original evaluation traces for a) and b), and synthesized the workloads for c). We conducted our experiments on an Ubuntu 18.04 server with six Intel Xeon E5-2643 Quad-core 3.40 GHz CPUs, 128 GB RAM, and 1 TB hard disk. For adversarial testing, we have used a Tofino hardware switch (for Tofino-P4 programs) and a bmv2 software switch (for P4-16 programs) as DUTs.

Vera programs	LoC	Stateful?	Vera (sec)	P4wn (sec)
copy-to-cpu	70	○	0.27	0.07
resubmit	70	○	0.27	0.09
encap	130	○	0.31	0.04
simple_router	145	○	0.28	0.08
NAT (S3)	290	○	0.61	0.25
ACL (S4)	200	○	0.29	0.13
Axon	100	○	0.3	0.08
NDP switch	210	○	0.3	0.12
Beamer mux	340	○	0.35	0.13
P4xos	260	○	0.34	0.08
switch.p4	6000	○	8.5	1.2
New programs	**LoC**	**Stateful?**	**Vera (sec)**	**P4wn (sec)**
lb (S1)	200	△	✗	0.12
flowlet (S2)	250	●♦	✗	0.26
Blink (S5)	928	●♦	✗	0.37
NetCache (S6)	674	●♦	✗	7.50
*Flow (S7)	1728	●♦	✗	2.15
p40f (S8)	884	●♦	✗	5.00
NetHCF (S9)	822	●♦	✗	0.57
Poise (S10)	842	●♦	✗	0.36
NetWarden (S11)	1332	●♦	✗	0.60
counter (S12)	90	●♦	✗	0.1
htable (S13)	160	●♦	✗	0.06
cmsketch (S14)	225	●♦	✗	0.17
bfilter (S15)	185	●♦	✗	0.07

Table 1: Vera, a state-of-the-art P4 profiler, can only perform stateless analysis; P4wn has similar performance on these programs. Only P4wn can analyze stateful data plane systems. Empty circle: stateless; solid circle: stateful; triangle: stateless, with complex hash functions, diamond: approximate data structures.

5.1 Stateful analysis

A key metric for program profilers is scalability. We start by benchmarking against Vera [71], a state-of-the-art P4 symbex engine. As expected, Vera only supports stateless analysis (Table 1). When given a stateful program, Vera sets all state variables to empty, and performs single-packet analysis regardless of state; this does not exercise stateful program behaviors. P4wn performs similarly with Vera on these stateless programs. (Anecdotally, an earlier version of P4wn scaled poorly with switch.p4; we found that this is due to the many branching behaviors of this program. Vera scaled better on this because it specifically optimizes for handling branchy programs. We ported two techniques from Vera to P4wn, namely "drop optimizations" and "concrete packet layouts" [71, 72], to match the performance. The measurement shown for switch.p4 is per packet layout for both Vera and P4wn. Appendix A.2 includes more detailed discussion.) Most of the recent data plane systems are stateful, containing sophisticated data structures like hash tables or sketches; only P4wn can support these programs.

Since Vera cannot handle these stateful programs, the rest of our evaluation uses KLEE as the baseline system for comparison, which is the general-purpose symbex engine that P4wn builds upon. P4wn uses telescoping to explore deep code blocks and greybox analysis to analyze approximate data structures; in contrast, KLEE simply performs an exhaustive search. We further use a timeout threshold of one hour for all executions in order to bound the experiment time. We found that for stateless programs and programs with comparatively simple state, KLEE and P4wn have similar levels of performance. The baseline times out for data plane systems with complex state and deep code blocks (S5, S6, S11). In contrast, P4wn finishes its analysis for all tested programs within one minute.

Telescoping. Figure 6a shows the results for the benchmark S12, which monitors the numbers of TCP and UDP packets, and samples every N-th packet of each kind to the CPU. For a comprehensive evaluation, we further create eight variants of this program with N ranging from $[1, 128]$. As we can see, the baseline times out when $N > 24$. This is because every symbolic packet forks two paths, one for TCP and another for UDP, and $N = 24$ produces 2^{24} execution paths for the baseline. For $N \leq 4$, P4wn takes the same amount of time as the baseline, because it uses a sequence of $\gamma = 4$ packets for telescoping; P4wn exhaustively analyzes all paths for this short sequence to detect periodicity. However, as N grows, P4wn almost stays at a constant time (0.1s), because it can leverage the periodicity to generalize the results.

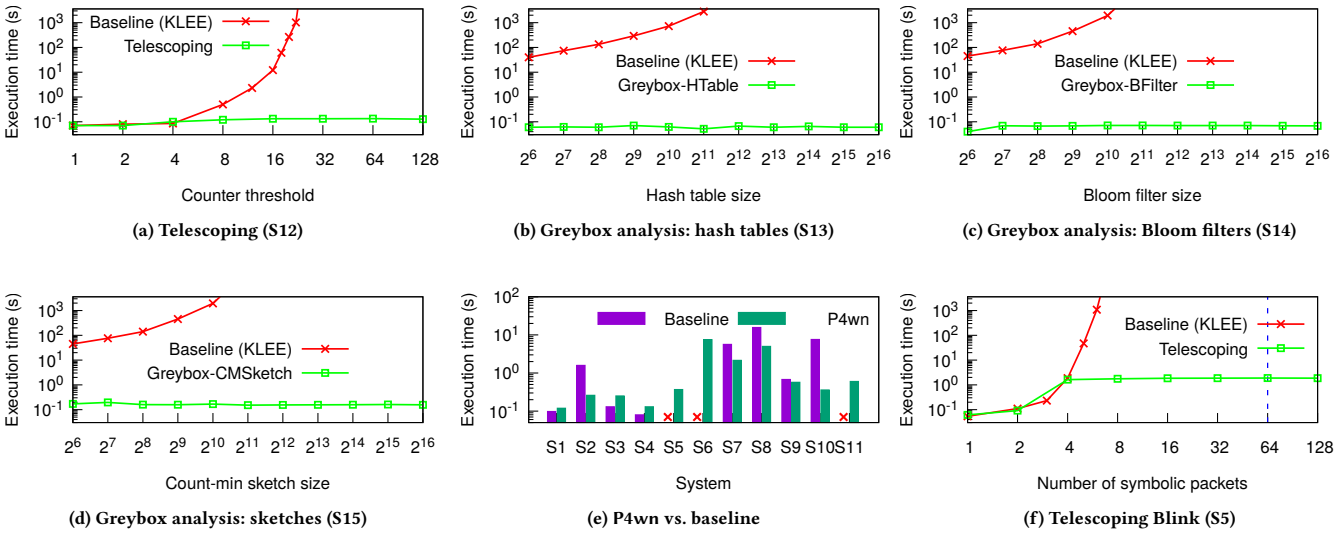
Greybox analysis. Figures 6b, 6c, and 6d present the results for the greybox analysis of hash tables, Bloom filters, and count-min sketches of varying sizes (S13–S15). The baseline KLEE uses the theory of arrays [32] to encode uncertainty of every symbolic access into the path constraints. We fixed the number of symbolic packets to be 5, and tuned the data structure sizes by changing the number of entries (for sketches and hash tables, every entry has four bytes; for Bloom filters, every entry has one single bit). As we can see, the baseline times out on CRC-16 hash tables with more than 2^{11} entries, and on Bloom filters and sketches with more than 2^{10} entries, respectively. P4wn, on the other hand, finishes its execution again in almost constant time for all three systems, This is because the greybox analysis only relies the probabilistic data store, which contains a small number of symbolic state.

Complex data plane systems (S5-S11). We found that the baseline scales poorly on complex stateful systems with deep code blocks. Figure 6e compares P4wn with the baseline, and Figure 6f further shows the detailed results for Blink as the symbolic sequence gets longer. Blink requires 64 packets to trigger rerouting, but the baseline system times out for eight packets, which is far from enough. Moreover, P4wn has achieved 100% code coverage for all tested systems.

Model counting vs. trace queries. The above results were obtained by configuring P4wn to use the LattE model counting solver. We found the running time to be higher if it is configured to query the traffic trace instead. This is because the queries to the solvers are generally simple integer constraints over header fields (e.g., over source ports for load balancing); the query time to the LattE solver is on average 0.02s per query, with 6.7 queries per system on average. In comparison, going through the network trace took more time, and the execution time would further depend on the lengths of the collected traces. Figure 7 shows the results.

5.2 Probabilistic analysis

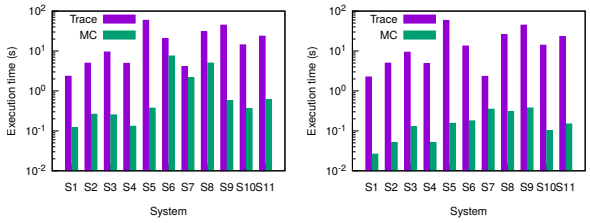
Next, we evaluate P4wn's probabilistic profiles. Since P4wn is the first probabilistic P4 profiler, there do not exist off-the-shelf baseline systems that can provide ground truths to compare against.



(a) Telescoping (S12) (b) Greybox analysis: hash tables (S13) (c) Greybox analysis: Bloom filters (S14)

(d) Greybox analysis: sketches (S15) (e) P4wn vs. baseline (f) Telescoping Blink (S5)

Figure 6: The baseline does not scale for programs with complex state or deep code blocks, whereas P4wn finishes within one minute for all tested systems. The new techniques in P4wn scale well on stateful programs.



(a) End-to-end processing time (b) Time for 'updateProb'

Figure 7: P4wn can be configured to issue queries to model counting solvers or network traces. In both cases, P4wn can finish within minutes. (a) shows the end-to-end execution time for querying the model counting solver vs. the network trace; (b) shows the time for 'updateProb', which performs probability queries.

Therefore, we had to use reasonable approximations to create two baseline systems, and measured their relative differences.

First, we created a baseline called ex, which performs an exhaustive symbex search and then model counts the probabilities; it still relies on greybox analysis for scalability. We use ex's output as the ground truth to measure the accuracy of P4wn. If ex estimates the probability of N to be α_N, and P4wn estimates it to be $\widehat{\alpha_N}$, then we use $\gamma_N = |\widehat{\alpha_N} - \alpha_N|/\alpha_N$ to compute the inaccuracy ratio. We found that P4wn achieves $\gamma \leq 0.04$ for all code blocks. P4wn underestimates the rerouting code block in Blink by 0.04, because this program has multiple aperiodic paths, which P4wn has missed in its periodicity analysis. As another note, this ex baseline timed out on Blink, NetWarden, and NetCache; the above results were obtained using smaller versions of these programs (e.g., Blink uses 4 instead of 64 retransmissions to trigger rerouting).

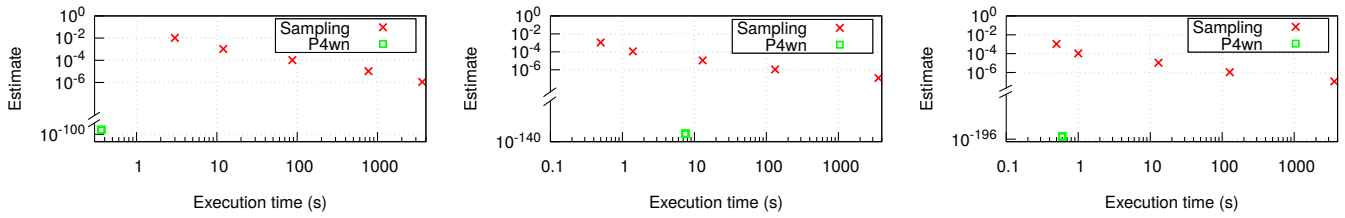
To test the original versions, we created another baseline ps that uses path sampling [28] for scalability. Since ex cannot produce probability estimates to compare against, we focus instead on the trend of profiling *granularity* over time instead of accuracy. For ps, we set the confidence level to 99%, and tuned the sampling error bound to obtain multiple data points. Figures 8a-8c present the results. As we can see, ps can increase its profiling granularity over a longer running time, achieving finer and finer estimates. However, the finest granularity it can achieve is still orders of magnitude coarser than P4wn's estimates. As a concrete example, the exhaustive baseline ex estimates $\Pr[N_{\text{reroute}}]$ in the smaller version of Blink to be $O(10^{-22})$, but the path sampling can only achieve a granularity of $O(10^{-6})$ on the full program. This coarse-grained profiling result is due to the fundamental difficulty in sampling low-probability events [12]—in order to sample rare events, we need a very large sample size. P4wn, on the other hand, can use telescoping to achieve much more fine-grained estimates.

5.3 Adversarial testing

Next, we evaluate how effectively P4wn can perform adversarial testing, by generating concrete packet traces for the top-10 lowest-probability code blocks for each system.

Efficiency. As Figure 9 shows, P4wn takes within one minute to generate traces for each tested system. As the decomposition shows, most of the time was spent in directed symbex, which collects path constraints, and havocing, which generates concrete sequences for greyboxes. Solving the path constraints does not take much time.

Adversarial testing traces. In total, P4wn has identified 13 different adversarial inputs that can cause significant performance disruption. For Blink [38], NetWarden [75], and Poise [46], the



(a) Sampling baseline (S5) (b) Sampling baseline (S6) (c) Sampling baseline (S11)

Figure 8: P4wn can obtain more fine-grained profiling estimates using telescoping.

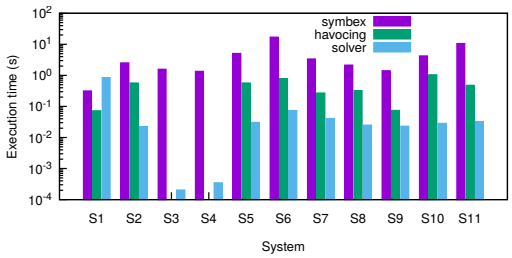


Figure 9: P4wn can generate adversarial testing traces within one minute for all tested systems. Most of the execution time was spent in directed symbex and havocing.

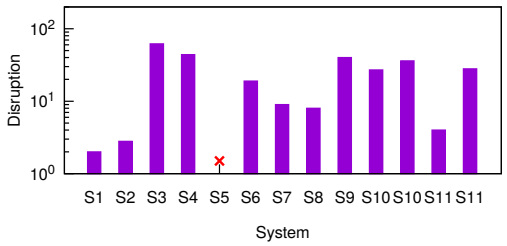


Figure 10: The adversarial testing traces cause significant performance differences than normal traces. The nature of disruption is system-dependent (e.g., throughput, recirculation); for S5 (Blink), the disruption flips the decision on which route to use, so we leave out the ratio for clarity.

authors have manually analyzed adversarial traffic inputs for their systems—P4wn has discovered all of them automatically.

Disrupting traffic forwarding. The first class of adversarial workloads would disrupt normal forwarding decisions, and P4wn discovered them in S1, S2, and S5.

For S1 (lb), the profile obtained by P4wn shows that the expected behavior is that traffic will be roughly evenly split across all ports. On the other hand, the workloads generated by P4wn cause hash collisions, so the flows are hashed to the same slot. This incurs high loads in a victim switch port, whereas other ports remain idle.

For S2 (flowlet), the profile is similar as that for S1. Normally, long flows will be split into flowlets, which are then load balanced across ports. The test trace generates collisions so that the victim port has high utilization, causing load imbalance.

For S5 (Blink), the profile shows that it applies forwarding decisions for most traffic, but only reroutes with a very low probability. The generated trace consists of 32 flows that exhibit retransmission behaviors—i.e., packets in the same flow have the same TCP sequence number. Moreover, the retransmissions happen within the same sliding window of Blink. This causes Blink to mistakenly infer link failures and trigger rerouting to a backup path.

Control plane disruption. The second class of adversarial workloads amount to denial-of-service attacks to the switch control plane. P4wn found these in S3 (nat), S4 (acl), S9 (NetHCF), and S10 (Poise). Whereas most traffic is directly processed in the data plane, P4wn identified one low-probability code block in each system that triggers control plane involvement. Packets that trigger these blocks are sent to the control plane for processing, causing heavy CPU overheads and overloading the data/control plane channel.

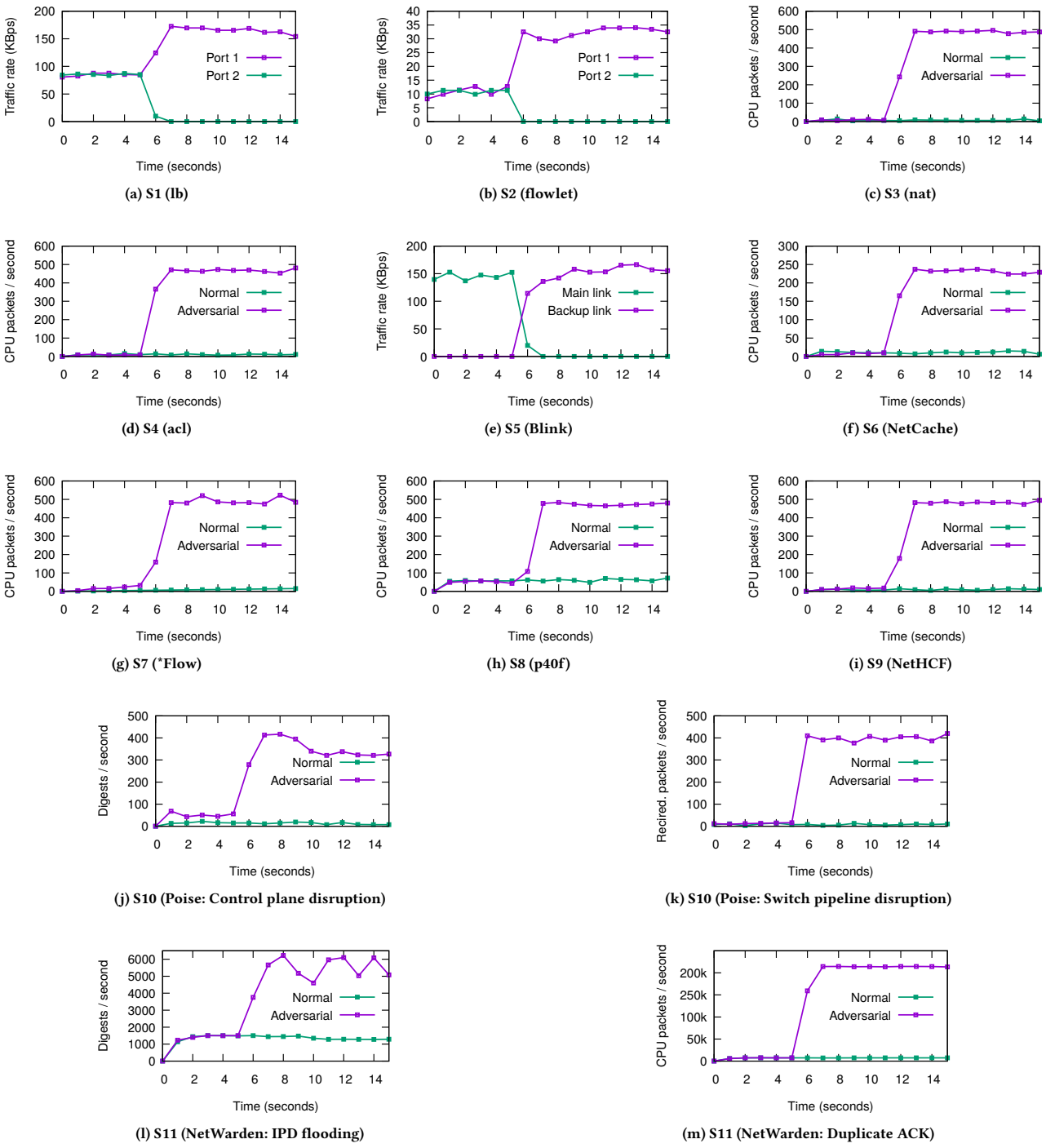
Backend server disruption. The third class of workloads incurs heavy processing at a backend server; as a side effect, it also causes high loads at the switch port that is connected to the server. P4wn generated workloads in S6 (NetCache), S7 (*Flow), S8 (p40f), and S11 (NetWarden).

For S6, the normal behavior is that most key/value requests would hit the in-switch cache, because the "hot keys" are cached inside the network; this is meant to reduce the amount of workloads that reach the backend servers. The adversarial traces cause cache misses and generate workloads to the backend servers. This would also trigger hot key reports from NetCache to the server.

For S7, the telemetry data is evicted typically only when the SRAM buffer is full. The adversarial traces cause hash collisions in the program logic that maintains the SRAM buffers, so the buffer content is frequently evicted to the backend servers.

For S8, the normal behavior is that the switch has the needed OS fingerprints for most traffic, so it rarely contacts the database for further matching. The adversarial traces generate a SYN packet with an unrecognized signature, and then a large number of HTTP packets in the same flow. All such packets are forwarded to the signature database, causing a high load.

For S11, P4wn has discovered two adversarial traces. The first trace contains IPD distributions that are significantly larger than the covert timing channel threshold in S11. This causes such packets to be sent to the "defense slowpath", which runs in software, for



(a) S1 (lb)
(b) S2 (flowlet)
(c) S3 (nat)
(d) S4 (acl)
(e) S5 (Blink)
(f) S6 (NetCache)
(g) S7 (*Flow)
(h) S8 (p40f)
(i) S9 (NetHCF)
(j) S10 (Poise: Control plane disruption)
(k) S10 (Poise: Switch pipeline disruption)
(l) S11 (NetWarden: IPD flooding)
(m) S11 (NetWarden: Duplicate ACK)

Figure 11: The adversarial workloads discovered by P4wn cause significant performance disruption to the tested systems. For adversarial testing, we started each test with normal workloads and then switched to adversarial traces in the middle of the test to show the comparison.

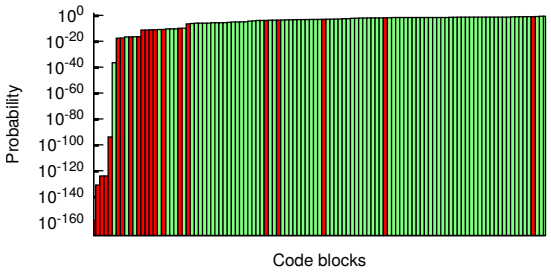


Figure 12: P4wn's probabilistic profiles are a useful guide for adversarial testing, as there is a strong correlation between a code block's probability and the expensiveness of the processing behavior. (Top 110 code blocks out of 220 are shown.)

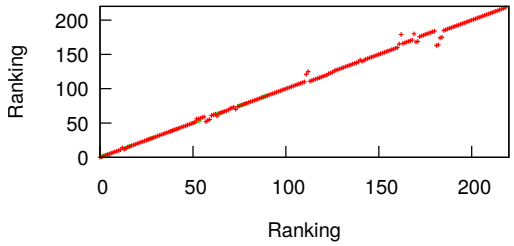


Figure 13: P4wn is robust to natural profile variances.

their IPD patterns to be reshaped. The second contains duplicate ACK packets, which is perceived by S11 as loss signals; this further causes the defense slowpath to buffer a large amount of packets perpetually.

Switch pipeline congestion. For S10 (Poise), P4wn identified a workload that triggers heavy traffic to be recirculated in the switch pipeline. S10 handles hash collisions by recirculating traffic in the data plane until the hash collisions have been resolved by the control plane. The traces generate a large amount of hash collisions to cause a large amount of recirculated traffic.

Adversarial trace impacts. Using the backtesting engine, we have empirically validated that the adversarial testing traces cause severe performance disruption. Figure 10 shows that they lead to 2-64× degradation compared to normal workloads. For different systems, the performance metric could be traffic rate, number of packets sent to the switch control plane or backend server CPUs, or system-specific metrics. Since most of these systems are written in the `bmv2` version of P4 for software targets, the relative comparison between normal and adversarial workloads is more informative than the absolute performance numbers.

Figure 11 shows the impact of the 13 adversarial traces in detail. As we can see, the workloads discovered by P4wn lead to significant performance disruption compared with normal testing traces. For Figures 11a, 11b, and 11e, the main performance metric is the traffic rate per switch port before and after the adversarial workload injection. For Figures 11c, 11d, 11f, 11g, 11h, 11i, 11m, the performance metric is the number of packets that the control plane or backend server CPUs need to process. For Figures 11j and 11l, the performance metric is the number of control plane digests (a hardware mechanism on Tofino switches to generate messages from the hardware data plane to the local switch control plane) that the switch control plane receives. For Figure 11k, the performance metric is the number of packets that need to go through a dedicated switch recirculation pipeline.

Usability. By a manual analysis, we found that one adversarial trace is not discovered from our testing in S7, which also causes data eviction to the backend server. Figure 12 orders the top 110 (50% of the total) code blocks by their probabilities and colors those with expensive processing behaviors in red. Depending on the available time for testing, one could increase the coverage by including more code blocks.

Since a network's traffic composition may naturally vary from time to time, we have additionally tested how robust P4wn's profile is for adversarial testing based on probabilities. To measure this, we have used three different CAIDA traces that are collected in 2016, 2018, and 2019, respectively. We have similarly used three different traces for Poise and NetCache, with different "context packet" frequencies for the former, and different "workload skews" for the latter. To ensure profile diversity, we have compared the interactive query results for the same queries across profiles, and found that the variance for individual queries can be as high as two orders of magnitude. The metric we have used to quantify its robustness is the *rankings* of code blocks based on their probabilities across systems.

Figure 13 plots this result across all traffic profiles for all programs. If a code block has a ranking of a in the first profile, we use this as the baseline and draw a dot at location (a, b) where b is the index in other profiles that has maximum movement. A perfectly robust system would produce results along the the diagonal. As we can see, most dots fall on the diagonal. For the code blocks whose rankings have changed, the average movement distance is 3.23. These results show that prioritizing low-probability edge cases is a robust method for adversarial testing.

6 DISCUSSIONS

We discuss the limitations of P4wn as well as future directions.

Mitigating adversarial disruptions. Although P4wn does not actively generate mitigation strategies, its probabilistic profile serves as a useful guide to craft potential countermeasures. One approach is to add counters to code blocks and measure their distribution at runtime. The runtime profile can be compared against some expected profile for anomaly detection—e.g., excessive occurrences of edge cases might trigger alarms to the operator or rate limiting logic. Moreover, relying on code block probability for adversarial testing does not guarantee a complete discovery of all expensive behaviors in a program. An adversarial behavior might occur, for instance, in common-case code blocks, or by combining multiple code blocks in some nifty manner.

Match/action table entries. As a limitation, P4wn assumes that match/action table entries are known; our prototype currently does not handle "symbolic" entries as in Vera [71], although such integration should be feasible. By making entry fields symbolic and including them in the symbex analysis, P4wn could similarly reason

about their impact on program behaviors and profile their probabilities. Such enhancements would enable P4wn to scale further, as match/action entries introduce n-way conditionals that lead to branchy programs.

Testing targets. P4wn currently does not analyze data plane systems that are distributed across multiple devices, although such systems are also on the rise [40, 47, 76]. One interesting future direction is to handle such systems, e.g., by composing multiple switch programs as one monolithic system to analyze them together. Moreover, although our testing is performed against programmable switches, SmartNICs [3] and software targets [65] are also becoming P4 programmable. P4wn's techniques are applicable to these contexts as well.

Conversion to C. P4wn relies on translating P4 programs to C for symbex. This is an engineering limitation as KLEE does not have direct support for languages like P4. Interestingly, this conversion to C can potentially enable a joint control/data plane analysis, since both components are now in C. This would lead to two interesting topics: a) we could also profile the performance of the control plane and discover expensive traffic patterns for both components; and b) this joint program would essentially be multi-threaded, because the control and data plane components run asynchronously. It would be interesting to investigate multi-threaded symbex techniques [15] to discover race conditions between the two components.

Beyond adversarial testing. As discussed earlier, program profilers have general applications. We have briefly considered another use case: profile-guided NF offloading. The idea is to partially offload NF logic to a programmable switch to alleviate most critical performance bottlenecks, if a full offloading is infeasible due to switch resource constraints.

In this case study, we have used eBPF [7] programs, which are a popular form of host-based packet processing programs. Since these programs are written in C, P4wn can analyze them out of the box. We applied P4wn to an eBPF program in the BEBA project [6] that implements port knocking. This is a firewall-like NF that requires the sender to generate a predefined sequence of port knocks (i.e., packets with certain destination ports) before it can initiate SSH requests. In the initial setup, the middlebox server runs this NF to examine all incoming traffic before the packets are forwarded to the final destinations. We used P4wn to analyze this program, which identified hotspot components that handle non-SSH or port knocking traffic. We then manually offloaded these components to a P4 switch, and validated that this led to an average packet latency improvement of 27% compared to the original eBPF processing. We also compared this with a full offloading of the entire program to the P4 switch by a complete rewrite, but found that it only resulted in <1% additional improvement; on the other hand, full offloading has used 12.5× more SRAM, 2× more VLIW (Very Long Instruction Words), and 3× more stages in the switch than guided offloading. Using P4wn to guide NF offloading and achieve performance/resource tradeoffs is an interesting direction of future work.

7 RELATED WORK

Probabilistic symbolic execution. P4wn borrows from recent developments from the program analysis community, especially probabilistic symbex [35], which can quantify execution probabilities and has found use in reliability analysis [27], measuring software changes [29], and performance profiling [21]. P4wn develops a range of domain-specific techniques on top of this for data plane systems.

Probabilistic network analysis. Observing that networks are governed by complex programs, researchers have applied program analysis techniques for network testing and verification. The first generation of such tools focus on qualitative properties [13, 14, 25, 30, 36], such as whether two nodes can reach each other. However, these tools cannot analyze quantitative (e.g., probabilistic) properties that arise due to network failures, traffic changes, or load balancing. Recognizing that network environments are nondeterministic in nature, as of late, researchers have been actively working towards enabling probabilistic analyses for network programs [33, 66, 67, 70, 73]. Some of these systems enable reasoning of probabilistic programs written in customized languages [33, 34, 67]. Compared to existing work, P4wn is the first probabilistic profiler for data plane systems.

Adversarial testing. Adversarial testing has been demonstrated to be valuable for many scenarios [10, 18, 37, 39, 41, 42, 49, 52, 53, 63, 74]. In this space, CASTAN [63] leverages symbex to identify slow execution paths in software-based network functions for adversarial testing. P4wn contributes a new technique that performs adversarial testing by identifying edge cases, and it looks at a different class of systems. The most related work is a workshop paper [45], but it does not contain a complete design or implementation.

8 CONCLUSION

There is increased interest in developing program profiling and analysis techniques for network data plane programs. However, existing profilers [24, 61, 71] only support stateless programs. P4wn enables *stateful* analysis of data plane systems, and it captures the *probabilistic* nature of network processing. We develop novel symbex techniques in P4wn, and apply it to adversarial testing. Our evaluation shows that P4wn scales significantly better than the baseline, and that it can effectively perform adversarial testing for complex data planes with sophisticated state. The edge-case packet traces cause 2-64× performance disruption to the tested systems.

9 ACKNOWLEDGMENTS

We thank Nate Foster for shepherding our paper and are grateful for the reviewers' insightful feedback. We also thank Bill Hallahan, Kuo-Feng Hsu, Jennifer Rexford, Dingming Wu, Yifei Yuan, and Jialu Zhang for their thoughtful comments on earlier versions of the paper, and Yujian Ou for his contribution in the earlier stage of the project. We gratefully acknowledge Costin Raiciu and Radu Stoenescu for providing the source code of Vera and for their help in the setup. This work was partially supported by NSF grants CNS-1942219 and CNS-1801884.

A APPENDIX

A.1 Description of the tested systems

We include a more comprehensive description of the tested programs below.

- **S1: lb.p4** is a load balancer that computes a hash of a packet's flow ID, and evenly distributes traffic to outgoing ports.
- **S2: flowlet.p4** implements flowlet switching to mitigate out-of-order delivery in TCP. It records the elapsed time between two packets in the same flow, and batches packets into flowlets and forwards them to the same port if they arrive closely together. Otherwise, it computes a new forwarding decision for the new packet based on the current link utilizations.
- **S3: nat.p4** is a network address translator that maps internal and external IP addresses. The first packet in a flow is processed by the control plane to compute a mapping, and subsequent packets will inherit the same mapping.
- **S4: acl.p4** performs network access control. It forwards or drops packets based on the access control list; if packets do not match on any entry, they are sent to the control plane for the final decision.

The seven data plane systems are from recent research projects. **S5, S7, S9, and S10** contain hash tables, **S6, S8, S10** contain Bloom filters, and **S6, S11** contain sketches. All of these systems are stateful.

- **S5: Blink [38]** detects remote link failures by examining TCP retransmission behaviors. It monitors 64 TCP flows and triggers rerouting to a preconfigured backup path if more than 32 monitored flows experience retransmission.
- **S6: NetCache [44]** implements an in-network cache for key/value pairs. Hot items are cached inside the switch and served directly from the network.
- **S7: *Flow [68]** is a network telemetry system. It collects per-packet telemetry data (e.g., packet timing and size) and organizes the data in per-flow hash table. Once the buffer is full, the data is evicted to a backend server.
- **S8: p40f [9]** performs passive OS fingerprinting by observing network traffic. It extracts signatures from SYN packets and matches them against a preloaded table for fingerprinting.
- **S9: NetHCF [54]** identifies spoofed traffic by detecting inconsistent TTL fields in packets with the same source addresses.
- **S10: Poise [46]** is a context-aware access control system for mobile clients. Clients generate context signals (e.g., GPS coordinates) periodically, and the system makes real-time decisions based on the signals.
- **S11: NetWarden [75]** mitigates network covert channels. It detects abnormal packet IPDs (inter-packet delays) for timing channels, and suspicious header values for storage channels; upon detection, it further delays the packets to destroy IPD patterns and rewrite packet headers.

The next four are benchmark programs that we have developed to evaluate the new techniques of P4wn.

- **S12: counter.p4** counts the numbers of TCP and UDP packets using two stateful counters. It mirrors a packet to a remote server for every N packet of each kind. We use this program to measure how well that P4wn can handle deep code blocks.
- **S13: htable.p4** uses a CRC-based hash table of size S for maintaining the number of packets per flow. It mirrors a packet to a remote server for every N packets of each flow. We use this program to measure how well that P4wn can handle complex approximate data structures using greybox analysis. Benchmarks S14 and S15 serve similar purposes.
- **S14: cmsketch.p4** uses a count-min sketch of size S for counting the number of packets per flow. It similarly mirrors a packet to a remote server for every N packets of each flow.
- **S15: bfilter.p4** uses a Bloom filter of size S to encode approximate membership test. It counts the number of packets that hits the bloom filter and mirrors a packet to the controller for every N hits.

A.2 Experience with switch.p4

We report our evaluation experience with switch.p4, which is an implementation of a full stack of protocols for a network switch. The most significant difference between *forwarding* programs (like switch.p4) and *data plane systems* (which we specifically optimize for) is that their scalability bottleneck stems from different sources.

This program has *complex branching behaviors* for handing a wide range of different network prototypes and forwarding decisions (several hundred tables), but it has very *simple state* (tens of stateful registers), In contrast, the *data plane systems* we have focused on have very *complex state*, because their functions are akin to "middleboxes" (on the order of 10k stateful registers per approximate data structure); on the other hand, they have relatively *simple branching behaviors* (tens of tables). A significant portion of these tables and branches are eventually for facilitating state maintenance and sophisticated stateful processing. Consider a CRC-16 hash table as an example: one instance alone would have $2^{16} = 65536$ stateful registers. In order to symbex data plane systems, our techniques are focused on optimizing stateful symbex over a sequence of symbolic packets. Therefore, for all other tested systems that we have presented in the evaluation section, P4wn can successfully profile them using primitive techniques in KLEE for handling branching behaviors, because branching is not the main bottleneck.

As discussed in Section 5.1, our earlier P4wn prototype encountered scalability bottlenecks for switch.p4, due to its complex branching behaviors. KLEE's primitive techniques for cutting branches lead to a coverage of 25.9% in one hour (the default timeout threshold we have used for the experiments), and a coverage of 40.3% for 12 hours. This finding is in line with benchmark results in Symnet (symbex engine for Vera) against KLEE. Symnet/Vera further proposed branch-cutting optimizations. Consider an intuitive example, where a symbolic packet is dropped by an access control rule. A symbex engine aware of the semantics of "dropping a packet", such as Symnet, can immediately prune any further executions, but a general-purpose engine like KLEE will continue forking execution paths until the program calls "free" on the packet buffer. Therefore, such domain-specific optimizations enable the engine to scale better on branchy programs. We have ported two of these techniques for P4wn to match Vera's performance. Using the same setup described in Section 5.1 of Vera [71] with concrete table entries, P4wn took 1-2 seconds for each packet type (with 200+ execution paths explored), which is comparable to Vera.

REFERENCES

[1] CAIDA: Center for Applied Internet Data Analysis. http://www.caida.org/data/.

[2] Latte - computations with polyhedra - uc davis mathematics. https://www.math.ucdavis.edu/~latte/.

[3] Netronome Agilio. https://www.netronome.com/products/agilio-cx/.

[4] P4 behavioral model. https://github.com/p4lang/behavioral-model.

[5] Publications and systems using KLEE. https://klee.github.io/publications/.

[6] The BEBA (Behaviorial Based Forwarding) Project. http://www.beba-project.eu/.

[7] What is eBPF? https://ebpf.io/.

[8] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. SIGCOMM*, 2010.

[9] S. Bai, H. Kim, and J. Rexford. Passive os fingerprinting on commodity switches. 2019.

[10] R. Banabic, G. Candea, and R. Guerraoui. Automated vulnerability discovery in distributed systems. In *Proc. HotDep*, 2011.

[11] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *Proc. FMCO*, 2005.

[12] J. L. Beck and K. M. Zuev. Rare-event simulation. *Handbook of Uncertainty Quantification*, 2015.

[13] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proc. SIGCOMM*, 2017.

[14] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Control plane compression. In *Proc. SIGCOMM*, 2018.

[15] T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *Proc. OOPSLA*, 2014.

[16] M. Borges, Q.-S. Phan, A. Filieri, and C. S. Păsăreanu. Model-counting approaches for nonlinear numerical constraints. In *Proc. NFM*, 2017.

[17] 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.

[18] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Proc. IEEE S&P*, 2014.

[19] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX OSDI*, 2008.

[20] S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Proc. CP*, 2013.

[21] B. Chen, Y. Liu, and W. Le. Generating performance distributions via probabilistic symbolic execution. In *Proc. ICSE*, 2016.

[22] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, 2008.

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

[24] D. Dumitrescu, R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Dataplane equivalence and its applications. In *Proc. USENIX NSDI*, 2019.

[25] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *Proc. OSDI*, 2016.

[26] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. BUZZ: Testing context-dependent policies in stateful networks. In *Proc. NSDI*, 2016.

[27] A. Filieri, C. S. Păsăreanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Proc. ICSE*, 2013.

[28] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys. Statistical symbolic execution with informed sampling. In *Proc. FSE*, 2014.

[29] A. Filieri, C. S. Pasareanu, and G. Yang. Quantification of software changes through probabilistic symbolic execution (n). In *Proc. ASE*, 2015.

[30] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.

[31] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos. Uncovering bugs in P4 programs with assertion-based verification. In *Proc. SOSR*, 2018.

[32] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. CAV*, 2007.

[33] T. Gehr, S. Misailovic, P. Tsankov, L. Vanbever, P. Wiesmann, and M. Vechev. Bayonet: Probabilistic inference for networks. In *Proc. PLDI*, 2018.

[34] T. Gehr, S. Misailovic, and M. Vechev. Psi: Exact symbolic inference for probabilistic programs. In *Proc. CAV*, 2016.

[35] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proc. ISSTA*, 2012.

[36] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *Proc. SIGCOMM*, 2016.

[37] T. Gilad, N. H. Jay, M. Shnaiderman, B. Godfrey, and M. Schapira. Robustifying network protocols with adversarial examples. In *Proc. HotNets*, 2019.

[38] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *Proc. USENIX NSDI*, 2019.

[39] M. E. Hoque, H. Lee, R. Potharaju, C. E. Killian, and C. Nita-Rotaru. Adversarial testing of wireless routing implementations. In *Proc. WiSec*, 2013.

[40] 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.

[41] S. R. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino. LTEinspector: A systematic approach for adversarial testing of 4G LTE. In *Proc. NDSS*, 2018.

[42] S. Jero, X. Bu, H. Okhravi, C. Nita-Rotaru, R. Skowyra, and S. Fahmy. BEADS: Automated attack discovery in OpenFlow-based SDN systems. In *Proc. RAID*, 2017.

[43] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proc. ICSE*, 2010.

[44] 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.

[45] Q. Kang, J. Xing, and A. Chen. Automated attack discovery in data plane systems. In *Proc. USENIX CSET*, 2019.

[46] 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.

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

[48] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. SIGCOMM*, 2012.

[49] C. Killian, K. Nagara, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *Proc. FSE*, 2010.

[50] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proc. SIGCOMM*, 2020.

[51] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[52] H. Lee, J. Seibert, E. Hoque, C. Killian, and C. Nita-Rotaru. Turret: A platform for automated attack finding in unmodified distributed system implementations. In *Proc. ICDCS*, 2014.

[53] H. Lee, J. Seibert, C. Killian, and C. Nita-Rotaru. Gatling: Automatic attack discovery in large-scale distributed systems. In *Proc. NDSS*, 2012.

[54] 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.

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

[56] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *Proc. PLDI*, 2014.

[57] F. Ma, S. Liu, and J. Zhang. Volume computation for boolean combination of linear arithmetic constraints. In *Proc. CADE*, 2009.

[58] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proc. SAS*, 2011.

[59] R. Meier, T. Holterbach, S. Keck, M. Stähli, V. Lenders, A. Singla, and L. Vanbever. (self) driving under the influence: Intoxicating adversarial network inputs. In *Proc. HotNets*, 2019.

[60] 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.

[61] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas. P4pktgen: Automated test case generation for P4 programs. In *Proc. SOSR*, 2018.

[62] P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Proc. Crypto*, 2003.

[63] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. Automated synthesis of adversarial workloads for network functions. In *Proc. SIGCOMM*, 2018.

[64] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki. Automated synthesis of adversarial workloads for network functions. In *Proc. SIGCOMM*, 2018.

[65] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. Pisces: A programmable, protocol-independent software switch. In *Proc. SIGCOMM*, 2016.

[66] S. Smolka, P. Kumar, N. Foster, D. Kozen, and A. Silva. Cantor meets scott: Semantic foundations for probabilistic networks. In *Proc. POPL*, 2017.

[67] S. Smolka, P. Kumar, D. M. Kahn, N. Foster, J. Hsu, D. Kozen, and A. Silva. Scalable verification of probabilistic networks. In *Proc. PLDI*, 2019.

[68] 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 *USENIX ATC*, 2018.

[69] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster. Composing dataplane programs with μp4. In *Proc. SIGCOMM*, 2020.

[70] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev. Probabilistic verification of network configurations. In *Proc. SIGCOMM*, 2020.

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

[72] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: scalable symbolic execution for modern networks. In *Proc. SIGCOMM*, 2016.

[73] K. Subramanian, A. Abhashkumar, L. D'Antoni, and A. Akella. Detecting network load violations for distributed control planes. In *Proc. PLDI*, 2020.

[74] M. von Hippel, C. Vick, S. Tripakis, and C. Nita-Rotaru. Automated attacker synthesis for distributed protocols. In *Proc. SafeComp*, 2020.

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

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

[77] K. Zhang, D. Zhuo, and A. Krishnamurthy. Gallium: Automated software middlebox offloading to programmable switches. In *Proc. SIGCOMM*, 2020.

[78] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, H. Zheng, and B. Zhao. Packet-level telemetry in large datacenter networks. In *Proc. SIGCOMM*, 2015.