Automated Attack Discovery in Data Plane Systems

Qiao Kang, Jiarong Xing, Ang Chen
Rice University

Abstract

Recently, researchers have developed a wide range of distributed systems that rely on *programmable data planes* in emerging switch hardware. Unlike traditional SDN switches, these new switches can be reconfigured to support user-defined protocols, customized packet processing, and sophisticated state. However, despite their popularity, one aspect that has received very little attention is their security implications.

This paper describes our ongoing investigation on a new class of attacks to these systems, which we call *sensitivity attacks*. We found that an attacker can generate malicious traffic patterns to "flip" the expected behaviors of a data plane system. We propose an approach to discovering attack vectors in a given data plane system and generating patches, both in an automated manner, and we present a set of preliminary experiments to demonstrate the feasibility of this approach.

1 Introduction

Half a century has passed since the proposal of the first RFC. Over the decades, networks have grown exponentially in size, popularity, and sophistication. One of the "growing pains", however, is the multitude of attacks. As the network infrastructure becomes increasingly critical, it is more urgent than ever to protect them from malicious attackers.

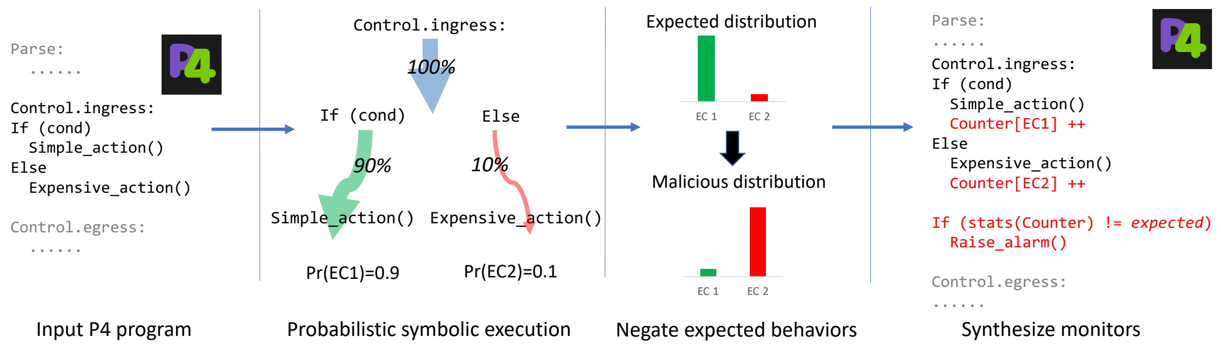
This paper looks at the security implications of one particular dimension of growth: *programmability*. We examine how the rising paradigm of *data plane programmability* can lead to new attacks that are easy to launch and quite damaging, we propose algorithms to systematically discover attack vectors, and, closing the loop, we develop ways to automatically synthesize corresponding defenses.

Programmable data planes. The latest development in the networking community is to make the switch data plane programmable. Unlike traditional SDNs, where the control plane is programmable in software and the switch data plane remains fixed in function, emerging switches have hardware data planes that can be reconfigured in-the-field using high-level languages (e.g., P4 [3]). Programmable data planes can support customized header fields and protocol types, enabling new protocols to be deployed without hardware upgrades. They can perform sophisticated operations (e.g., custom match/actions, arithmetics) over header fields at linespeed (Tbps), enabling non-trivial computation to be offloaded to the switches. Moreover, they can implement stateful data structures in hardware, making it possible for the network to adapt its behavior based on past events.

Leveraging this trend, researchers have built a wide variety of *data plane systems*. Owing to the specialized hardware support and optimized packet processing pipeline, a common feature of these systems is their unprecedented dynamicity. For instance, the Hula [13] system generates probes that carry network path performance information, and propagates them across the network; programmable switches can then extract path metrics from these special packets, and optimize their routing tables on-the-fly. The data plane in the Blink [10] system detects large-scale TCP retransmissions across flows, uses it as a signal for remote link failure, and triggers re-routing to backup paths. Compared to traditional solutions where intelligence resides on the software control plane, data plane systems can respond to network dynamics much faster and enact real-time change. Therefore, more data plane systems will likely emerge in the future.

What about security? A looming concern that has been largely ignored, however, is their security. By definition, data plane systems respond to *data plane signals*, which are simply network packets that can be spoofed and manipulated. The probes in Hula, the retransmissions in Blink, and most other forms of data plane signals, are unauthenticated in nature, so injecting fake signals does not require special privilege. Therefore, adapting the network behavior in response to unauthenticated stimuli seems potentially dangerous. Indeed, we found that an attacker can generate malicious traffic patterns to drive a data plane system from its normal behaviors, abusing the sensitivity of data plane systems to the input traffic. We call this new class of attacks *sensitivity attacks*.

One might wonder whether we can solve this by authenticating data plane signals, e.g., perhaps such systems should always authenticate data plane signals before using them. However, this is not as easy as it might seem. In order to maintain high speed, programmable data planes carefully limit the set of supported operations to those required for packet processing. Explicitly out of scope are loops, recursions, and many other sophisticated primitives for cryptography—at least for today's switch hardware. This means that, if we want to perform authentication, such operations can only happen on the switch control plane, which has general-purpose CPUs, therefore full software programmability. However, invoking this option would effectively downgrade the data plane system to a much slower system, as responses to network dynamics must once again go through software processing. In other words, there exists a tension between the level of dynamicity of a system and the strength of authentication it can afford.



```
Parse:
  ......

Control.ingress:
If (cond)
  Simple_action()
Else
  Expensive_action()

Control.egress:
  ......
```

```
Parse:
......
Control.ingress:
If (cond)
  Simple_action()
  Counter[EC1] ++
Else
  Expensive_action()
  Counter[EC2] ++

If (stats(Counter) != expected)
  Raise_alarm()

Control.egress:
......
```

Figure 1: The workflow of our approach. Our engine a) takes in a data plane program written in P4, b) performs probabilistic symbolic execution to analyze common-case behaviors, c) automatically generates input traces that would trigger corner-case behaviors, and d) synthesizes a set of monitors to detect malicious patterns, and patch the original data plane system with them.

Example attacks. Consider a simple but critical data plane feature: load balancing. One could develop a load balancer in P4 in tens of LoC to split traffic evenly across available paths [2]. A classic approach is to hash the packet header and pick one of the available outgoing paths based on the hash. Since switch hardware cannot support cryptographic hash functions, this "hash" could be as simple as `sport%N`, where `sport` is a packet's source port and `N` is the number of available paths. Though simple, this load balancer will work well as long as incoming traffic is roughly evenly distributed in terms of their source ports. However, consider an adversary that can craft skewed traffic patterns where `sport%N=n` is constant across the trace. The load balancer will simply shunt all traffic along one of the `N` paths, while all other paths remain under-utilized. Such concentration will cause normal flows on the victim path to experience high congestion and loss, creating a denial/degradation-of-service (DoS) attack.

A more sophisticated example is the Hula [13] load balancer, which detects the least-utilized path in the current epoch, and moves traffic from other paths to this path in the next epoch. Using the traffic pattern with constant `sport%N=n`, the attacker is no longer able to cause much harm, since Hula will detect that the `n`-th path has high load and divert traffic to the idlest path (say, `m`) in the next epoch. However, a smart attacker can craft a more advanced pattern. If she could keep in sync with the Hula epochs, and make educated guesses on which paths may be the idlest, she could then launch an attack by concentrating her traffic to the predicted `m` for each epoch. In other words, although Hula is resilient to one malicious pattern, it is vulnerable to a different pattern.

Consider now Blink [10], which is meant to be deployed inside an ISP to monitor retransmissions across TCP flows destined to the same service (e.g., Google). If it detects simultaneous retransmissions in many concurrent flows, it treats them as a signal that some link has failed, and re-routes traffic to pre-configured, alternative paths in a matter of seconds (as opposed to minutes in BGP). Here, a malicious traffic pattern would be a large number of concurrent flows towards the monitored destination that simulate TCP retransmissions, which would create periodic or even persistent routing chaos.

Our goal. Our research question is as follows: *Can we discover all malicious traffic patterns for a given data plane system and synthesize defenses in an automated manner?*

We aim to develop an system that accepts the source code of a data plane system as input, analyzes its expected behaviors, identifies malicious traffic patterns that would cause damage, and generates traffic monitors as "patches" to the system. The monitors would raise alarms when abnormal patterns are detected. Upon an alarm, the system could take further actions to examine, rate limit, or drop packets of a certain kind. Figure 1 shows this workflow.

2 Automated Attack Discovery

We take a three-step approach towards this goal. Given a data plane program, we start by automatically deriving its intended behaviors over normal traffic. Then, we generate traffic patterns that would drive the program away from common-case behavior as much as possible. Finally, we synthesize a set of monitors to detect such malicious traffic patterns.

2.1 Establishing expected behaviors

The first question we need to answer is how to derive "expected behaviors" of a data plane system. One could ask the programmer to supply a manual specification, but this would be labor-intensive and error-prone. One could also feed the system with random traffic traces and observe its outputs, but this may not be comprehensive.

Our first observation is that, obtaining "all possible behaviors" of a system can be achieved by *symbolic execution*, which comprehensively enumerates program execution paths and the kind of inputs that would trigger each path. Symbolic execution engines have already been customized to data plane systems [16, 18]—they can enumerate execution paths of P4 programs, and output constraints on packet headers for a particular execution to be triggered. For instance, they might find that packets that satisfy `(sport<1024)`$\wedge$`(dport==80)`$\vee$`(sip==1.2.3.4)` would be forwarded to the the second outgoing port by a given program.

Challenge #1: Quantifying behavior probabilities. However, one thing is still missing: of all possible executions, how do we know which ones are more "expected" than others? Our solution is to borrow a technique developed by the program analysis community, called *probabilistic symbolic execution* [9]. This is an enhanced version of symbolic execution that not only enumerates all execution paths, but also annotates each execution with a probability to be triggered. Under the hood, this is achieved by coupling symbolic execution with *model counting* [5]: the latter technique takes in a set of constraints, and counts the number of satisfying assignments. Suppose that the number of all possible input packets is K, and that the number of packets that would trigger a particular execution is k, then we can annotate this execution path with a probability k/K, if all input packets are equally likely. If the input packets follow a non-uniform distribution (e.g., TCP traffic is more likely than UDP), one can also compute the probabilities in a *distribution-aware* manner by encoding the actual packet distribution using an *input packet profile* [8].

There are many model counters that we could leverage. For instance, LattE [6] is specifically designed and optimized for handling linear integer arithmetic (LIA) constraints; it works by enumerating points inside a convex polytope as formed by a set of LIA constraints. ApproxMC [5] is an approximate model counter for SAT formulas. We could even use multiple model counters for different kinds of constraints.

Challenge #2: Identifying equivalence classes. We then further group the possible behaviors into *equivalence classes (ECs)*: packets in the same EC are processed in the same way, whereas packets in different ECs are treated differently. Equivalence could be simply defined by the forwarding behavior of packets, e.g., packets are considered to be in the same EC if they are forwarded to the same outgoing port. Or, we could define equivalence based on execution paths, e.g., some packets may trigger link failure alarms, some may be tagged with new headers, and others might simply be forwarded as is. We can then distinguish common cases from corner cases, by a) computing a set of ECs for a given system, EC_0–EC_k, and b) quantifying the probability of every EC: $\Pr[EC_i] = \sum_j \Pr[ep_{ij}]$, where $\Pr[ep_{ij}]$ is the probability of the j-th execution path in EC_i. Determining program equivalence in the most general case is, of course, undecidable, but restricting it to data plane systems makes the problem solvable [7].

2.2 Negating the expected behaviors

Our second step is to obtain input traffic patterns that would drive the system away from the expected behaviors. The statistical distributions obtained above are already a good starting point: an adversary wins if she can generate a trace where the output distribution is distorted from that of the common case as much as possible. In order to do this, she could pick many inputs from low-probability ECs in hope that the corner cases represent some network events that are expensive to handle. For instance, if 0.0001% packets would trigger Blink re-routing in the normal case, then the attack trace might increase this to 1%. Or, she could aim to cause skew in an even distribution: if a load balancer perfectly splits traffic 50% vs. 50%, then the attack trace might drive the distribution to 100% vs. 0%.

More formally, suppose that the expected behavior is that EC_i has a probability of $Pr[EC_i]$, then the attacker's goal is to create a sequence of packets that would lead to a maximally different probability $Pr[EC_i]^*$. The attack strength can then be quantified by $\Delta = \sum_i |Pr[EC_i] - Pr[EC_i]^*|, i \in [0..k]$.

Challenge #3: Handling stateful behaviors. Dealing with stateful data plane systems creates additional challenges, because in order to reach a certain EC, it may be necessary to send a sequence of packets in a particular order. As a concrete example, Blink responds to a sequence of retransmitted packets. One brute-force approach would be to feed the symbolic execution engine with N symbolic packets, ask the engine explore *all* possible execution paths, and pick the sequence that causes the maximum Δ. However, the limitation of this approach is scalability. One common issue in symbolic execution engines is state explosion: it may not be possible to explore all paths in useful time [4]. Exploring paths for a long sequence of packets would further amplify this concern.

Algorithm 1: Pseudocode for handling stateful systems

```
input  : Num. of packets (N); Equivalence classes (EC)
output : A sequence of attack packets (S)
$S \leftarrow \emptyset$;
while $|S| < N$ do
    foreach $EC_i \in EC$ do
        $p_i \leftarrow$ directed_symbex($EC_i$);
        $s_i \leftarrow S||p_i$;
        $\Delta_i \leftarrow$ compute_delta($s_i$);
    $(\Delta_i, s_i) \leftarrow$ max_delta($\Delta, s$);
    $S \leftarrow s_i$;
```

We propose an algorithm that generates attack sequences to maximize Δ greedily. Algorithm 1 takes N as input, and outputs S, which is an attack sequence with (roughly) N packets. In each iteration, the algorithm picks a target EC in a greedy manner, by trying all ECs and picking the one that causes the maximum Δ. To trigger a certain EC, our algorithm uses *directed symbolic execution* [14], which performs a heuristic-based search to explore the shortest path towards a particular line, and prioritizes execution state that is closest to the target. Each step would output a sequence of packets s_i. Then, the algorithm computes the Δ_i for each s_i, and picks the sequence that would cause the most deviation. It then continues the loop until N packets (or more) are generated.

Challenge #4: Handling hash tables. A second challenge in this step is handling hash tables, which are common building blocks in data plane systems. For instance, a hash table may map TCP connections to TCP state variables. Upon collision, a new entry might replace an existing entry, but hits would directly query from the table. Similar as before, keys to a hash table are typically obtained by performing a checksum-like function over packet headers (e.g., `CRC16(flowid)`). Given that hash tables could be large, and that hash functions consist of complex arithmetic operations, symbolically executing hash tables in a brute-force manner would be time-consuming.

Our approach is to treat such data structures as "greyboxes". We do not assume any knowledge about the content of the data structures, and we do not encode the hash functions as part of the path constraints. Rather, we model hash tables as opaque program components with pre-determined collision rates when fed with packets from a certain header space. Symbolically executing a hash table would only produce two execution paths: one for hash collisions and one for the rest, each annotated by its respective probability. We can then derive per-EC probabilities without having to know the state.

2.3 Synthesizing runtime monitors

This final step synthesizes a list of monitors to detect in real time whether the current traffic pattern falls into the common case. For each common-case EC (as defined by the value of $\Pr[EC_i]$), our engine outputs a monitor implemented in P4 to specifically look for this pattern. A monitor consists of a set of counters, one for each EC, and conditional statements inserted in the original program to update the counters per packet. Our engine then patches the original data plane system with these monitors, and inserts a statistical check that is triggered either per packet or periodically. This check compares the current counter values and a set of pre-defined counter values to detect whether the current traffic pattern is roughly within the range of normal behavior. Upon significant deviation, it raises an alarm to the network operator.

Challenge #5: Compressing monitors. The main challenge here lies in optimizing resource usage of these monitors. For a complex data plane system, there could potentially be many ECs. Simply creating a monitor for each common-case EC may incur a high overhead in terms of switch memory. This is because programmable data planes are extremely constrained in resources: their memory is on the order of 10MB and this resource needs to be shared by the original program and the monitors. Our tentative approach here is to use *sketching* to approximate the counts for all ECs in a single data structure. Count-min sketches [19], for instance, have low memory overhead and can provide strong theoretical guarantees in terms of approximation. Approximate counts can be stored in a single sketch and queried out later only with small accuracy loss.

3 Preliminary Experiments

Setup. We have performed an initial experiment to demonstrate the feasibility of our idea, using a load balancer program written in P4 [2], which splits traffic to two outgoing ports based on the hash function `sport%2`. Our experiments was conducted in Mininet `v2.3.0` with one P4 switch connected with three hosts, one as sender and the other two as receivers. We have used p4pktgen [16] as the symbolic execution engine. Since this tool does not support probabilistic symbolic execution, we have written a simple program that acts as a "mini model counter", based on the Python constraint library [1]. It takes the path constraints found by p4pktgen, and counts the number of satisfying assignments.

Attack discovery. The symbolic execution engine has found two ECs in this program: the constraint over packet headers for EC_0 is `sport%2==0`, and that for EC_1 is `sport%2==1`. The model counting program further shows that $\Pr[EC_0] = 0.5$ and $\Pr[EC_1] = 0.5$: that is, assuming the source ports of input packets are evenly distributed, then the expected behavior of the load balancer system should be an even traffic split.

Our attack discovery engine has generated two classes of attacks. (a) The attacker can generate a trace where all packets have `sport%2==0`, resulting in a malicious distribution of $\Pr[EC_0] = 1$ and $\Pr[EC_1] = 0$. (b) She could also generate a trace where `sport%2==1`, and the malicious distribution becomes $\Pr[EC_0] = 0$ and $\Pr[EC_1] = 1$.

Attack feasibility. To demonstrate the feasibility of the attack, we have chosen the attack strategy (a), and injected a crafted traffic trace from the sender to the load balancer. Figure 2 shows the traffic rate of each outgoing path. Initially, the sender uses normal traffic: we can see that the load balancer distributes the load roughly evenly across the two paths. At time $t = 15s$, the sender changes to the malicious traffic pattern. We can see that the traffic rate on port 1 has doubled, whereas port 0 is highly under-utilized.

Monitoring. Currently, we have manually written a monitor and integrated it as part of the load balancer program. The monitor maintains one counter for each EC, two counters overall, and counts the number of packets that trigger the corresponding EC. We then run a control plane program that performs KS-test periodically to check whether the distribution of EC counters conforms to the expected (i.e., uniform) distribution. It raises an alarm if highly skewed distributions are detected. Our experiments show that the monitor can successfully detect the skew, and raise an alarm.

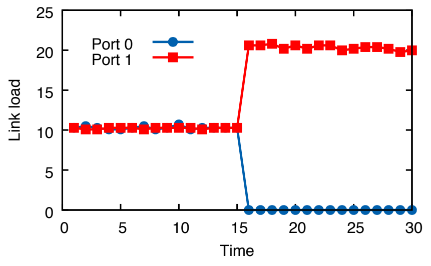


Figure 2: The load balancer splits traffic roughly evenly over normal traces. However, an attacker can generate malicious traffic patterns to drive the system away from its expected behavior. (X-axis unit: seconds; Y-axis unit: KB/s)

Summary. Our preliminary experiments show that our attack generation methodology is able to discover malicious traffic patterns for a simple data plane system, and that the resulting monitors can effectively detect attacks. As ongoing work, we are working on validating our methodology using a more diverse set of data plane systems.

4 Related Work

Automated attack generation. Researchers have developed automated attack discovery techniques for many systems, such as the TCP congestion control protocol [12], OpenFlow-based SDN [11], and software network functions [17]. Our work shares a similar motivation with existing work, but focuses on a different class of targets—data plane systems.

Data plane systems. Programmable data planes in recent switch hardware have enabled a wide variety of data plane systems, such as those for link failure detection [10], transport-layer load balancing [15], and load-sensitive routing [13]. Our work addresses a under-explored aspect: the security risks of data plane systems.

5 Acknowledgments

We thank Rajeev Alur, Ryan Beckett, Kuo-Feng Hsu, Kostas Mamouras, and the anonymous reviewers for their valuable feedback. This work was partially supported by an NSF grant CNS-1801884.

References

[1] Constraint solving problem resolver for Python. `https://labix.org/python-constraint`.

[2] P4 exercises: load balancing. `https://github.com/p4lang/tutorials/blob/master/exercises/load_balance/load_balance.p4`.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. USENIX OSDI*, 2008.

[5] Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A scalable approximate model counter. In *Proc. CP*, 2013.

[6] Jesús A De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of symbolic computation*, 38(4):1273–1302, 2004.

[7] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Dataplane equivalence and its applications. In *Proc. USENIX NSDI*, 2019.

[8] Antonio Filieri, Corina S Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 622–631. IEEE, 2013.

[9] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proc. ACM ISSTA*, 2012.

[10] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *Proc. USENIX NSDI*, 2019.

[11] Samuel Jero, Xiangyu Bu, Cristina Nita-Rotaru, Hamed Okhravi, Richard Skowyra, and Sonia Fahmy. Beads: Automated attack discovery in openflow-based sdn systems. In *Proc. RAID*. Springer, 2017.

[12] Samuel Jero, Md Endadul Hoque, David R Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in tcp congestion control using a model-guided approach. In *Proc. NDSS*, 2018.

[13] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. ACM SOSR*, 2016.

[14] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *International Static Analysis Symposium*, pages 95–111. Springer, 2011.

[15] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. ACM SIGCOMM*, 2017.

[16] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. P4pktgen: Automated test case generation for p4 programs. In *Proc. ACM SOSR*, 2018.

[17] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. Automated synthesis of adversarial workloads for network functions. In *Proc. ACM SIGCOMM*, pages 372–385. ACM, 2018.

[18] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with Vera. In *Proc. ACM SIGCOMM*, 2018.

[19] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. In *Proc. USENIX NSDI*, 2013.