Anomaly Detection in Data Plane Systems using Packet Execution Paths

Archit Sanghi
IIT Hyderabad, India

Krishna P. Kadiyala
Texas Christian University

Praveen Tammana
IIT Hyderabad, India

Saurabh Joshi
IIT Hyderabad, India

ABSTRACT

Programmable data planes provide exciting opportunities to realize fast, accurate, and data-driven control-loop decisions. Many data plane systems have been proposed for handling network dynamics (*e.g.*, congestion, failures) in near real-time. The core of these systems has packet-processing data-plane algorithms that continuously monitor traffic and respond automatically. Despite their benefits, automatic response to network events lead to increase in potential sources of inputs, and hence, increase in attack surface.

This paper takes a step towards securing such systems by (1) identifying possible attacks on recently proposed data-driven data-plane systems; and (2) designing a scalable tool for detecting such attacks at run time. Our approach models plausible expected behavior and uses the model as a reference to check whether the system is under attack. We conduct preliminary experiments to demonstrate the feasibility of our detection methodology.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **Network monitoring**; **In-network processing**; • **Security and privacy** → **Intrusion detection systems**; **Denial-of-service attacks**;

KEYWORDS

Programmable networks, In-network computing, Network security, Intrusion Detection Systems, Distributed Denial-of-Service attacks.

ACM Reference Format:
Archit Sanghi, Krishna P. Kadiyala, Praveen Tammana, and Saurabh Joshi. 2021. Anomaly Detection in Data Plane Systems using Packet Execution Paths. In *ACM SIGCOMM Workshop on Secure Programmable network INfrastructure (SPIN '21), August 23, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3472873.3472880

SPIN '21, August 23, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8637-1/21/08…$15.00
https://doi.org/10.1145/3472873.3472880

1 INTRODUCTION

Recent advancements of programmable switches [11] enable programming packet-processing behavior in the data plane using high-level domain-specific networking languages like NPL [4] and P4 [6]. This has opened up a wide range of opportunities to solve network problems considered difficult and complex in traditional fixed pipeline switches.

Recent works such as HULA [19], Contra [15], Blink [14], NetCache [8], leverage these advancements and propose data-driven data-plane systems to achieve better performance in terms of latency and throughput. The core part of these systems has packet-processing algorithms running in the data plane to continuously monitor traffic conditions or listen to data-plane signals and quickly respond to adapt to the network conditions in small timescales (*e.g.*, tens of nanoseconds).

Though such novel data-driven data-plane systems seem promising to achieve better performance, they run the risk of a larger attack surface and hence, may be vulnerable to network attacks not seen before. For example, a network attacker could potentially exploit the semantics of the control logic and craft adversarial network inputs [16, 26]. Such inputs would directly influence the decisions made by the systems and can negatively impact the behavior of a large portion of the traffic, potentially leading to severe performance degradation. Interestingly, to generate fake data-plane signals attackers do not need specific privilege since many of these data-plane systems can be tricked by spoofed and manipulated data-plane signals, or crafted traffic patterns from compromised hosts. To solve this problem, one may want to authenticate traffic or data-plane signals, both arriving at line rate. However, the P4 data plane supports a limited set of operations, with no support for loops and recursions, thus implying we cannot run sophisticated primitives for cryptography in the data plane.

In summary, to reap the performance benefits of data-driven data-plane systems and for such systems to be widely deployed, securing the systems from adversaries becomes crucial and immediate.

Our goal is to secure data-driven systems from adversarial network inputs. In working towards this goal, we propose a system to detect whether data-plane systems and associated packet-processing algorithms are under the influence of adversaries. The key idea is to detect behaviors that are abnormal when compared to some normal standard. Specifically, our approach is to (1) precisely understand the non-adversarial (normal) behavior of a data-plane algorithm written in P4; (2) monitor the actual behavior at run-time; (3) periodically compare the actual behavior with the normal behavior, raising alerts when the actual behavior deviates significantly from the normal behavior.

As an initial step, we identify possible attacks on recently proposed data-driven systems and explain how an attacker can target specific system components and perform attacks (details in Section §2). Next, we propose an anomaly detection system that performs statistical analysis on packet execution paths in the data plane to detect anomalies. The key insight of our approach is that the statistics of packet execution paths in P4 programs can be used to understand non-adversarial behavior (normal) such that a significant variation of these statistics at run-time would hint at abnormal behavior or behavior that is most likely observed if the system is under attack. For example, paths in P4 program that invoke switch CPU are typically expensive and not seen often. However, if we suddenly observe too many packets taking this path, we can consider this abnormal behavior.

The main challenge in implementing our idea is that programmable data planes are resource-constrained with limited per-packet time budget (100's of nano secs), a limited number of per-packet memory accesses (1-2 per-pipeline stage), and limited storage capabilities (order of MBs). We carefully design our system to work under these constraints. To be specific, our system has three main features. First, our approach relies on packet execution path statistics to understand the normal packet-processing behavior of a data-plane system. However, tracking each packet's execution path in the data plane under the constraints is challenging. To this end, a recent work, P4track [20], proposed a technique to track the execution path of every packet using Ball-Larus encoding and has shown that it works well under Intel's barefoot Tofino switch constraints. In this paper, we propose to build our system on top of this work and integrate it with our approach to detect malicious packet-processing behaviors (more details in section §2.2).

Second, it is important to maintain packet execution path statistics with minimal-to-no impact on packet-processing throughput. To achieve this, we maintain statistics in a hash table which can be implemented using stateful registers and can be updated at line-rate in the data plane. Details about the design choice of the hash table are given in section §3.2. Third, to detect abnormal behavior, one has to compare the tracked packet execution path statistics (or observations) with the expected (or normal) behavior. To do so, we use Pearson's chi-squared test [2] to determine whether there is a significant deviation between the expected behavior and the observed behavior (more details in section §3.3 and section §3.4).

Contributions. (1) We categorize possible attacks on existing data-driven data-plane systems built on top of programmable switches. (2) We design an anomaly detection system that helps to determine whether data-plane systems are under the influence of adversaries. (3) We perform preliminary experiments to demonstrate the effectiveness of our system in detecting anomalies in packet-processing behaviors of the NetCache [8] data-plane system.

2 ATTACKS ON DATA-DRIVEN DATA PLANE SYSTEMS

Existing data-driven systems such as HULA [19], Blink [14], Poise [18], Silkroad [27], and NetCache [8] leverage programmable hardware capabilities and significantly reduce either control-loop decision time, end-to-end latency, or telemetry data collection and processing overheads. To achieve their goals, these systems: (i) monitor network traffic, (ii) send feedback-control signals to control plane agents and/or adjacent nodes (*e.g.*, switches), and (iii) analyze the signals either entirely in the data plane or combination of data plane and control plane, and (iv) take appropriate action (*e.g.*, drop packet, add rule, reroute traffic, update register, create packet).

For instance, HULA [19], in each time window, listens to periodic probes carrying path performance metrics, picks the current best path towards a destination, and forwards traffic on the selected path. The Blink [14] system keeps track of the number of flows re-transmitting packets in a particular window on the current path to a destination prefix. If the number of flows exceeds a certain threshold, traffic is rerouted to a backup path. NetCache [8] classifies a key as *hot* if the key is accessed more than a certain number of times (threshold). The control plane then installs this key-value pair in the table, so that the switch would respond to subsequent key requests without having to visit a storage server, hence reducing latency.

2.1 Threat Model

It is important to identify the type of privileges required to attack data-plane systems. In this section, we identify who can send adversarial inputs to attack which parts of the data-plane system.

Attack target. Data-plane systems built on top of programmable switches respond quickly to data-plane signals. There are two kinds of signals: (1) messages generated by switch primitives monitoring incoming traffic; and (2) control packets sent by other switches or hosts in the network.

The attacker and their privileges. We consider two possible scenarios for an attacker to be able to send adversarial inputs — a compromised host, or a Man in The Middle(MiTM) attacker. For both types of attackers, we assume they know which packet header values and the order of packets, would influence the decision making in the data-plane system under attack. Thereby the attackers can trick the system by injecting malicious traffic into the network, or by manipulating the original control packets, or by creating fake control packets.

Note, to influence the decision making, the attacker does not require access to the P4 code, or know the packet-processing behavior, or access to the switch. The attacker can find the semantics of the control logic through other means (*e.g.*, by monitoring end-to-end performance) and then infer the packet header values and their order to influence the decision making. For instance, in the NetCache system, the attacker can deduce the threshold for classifying a key as "hot" by observing the difference in response times before and after a key becomes hot.

2.2 Possible attacks

In this context, we identify attacks on data-plane systems, especially those attacks targeting the behavior of a specific system component that responds to data-plane signals. Table 1 summarizes the possible attacks on data-driven data-plane systems that we present below:

CPU exhaustion. Computational cost represents the amount of work to be performed for a particular input (data/control packet). Typically, not all inputs have the same computational cost — packet-processing in the data plane is very fast (nano to micro-seconds) and packets that invoke the control plane (*e.g.*, copy-to-cpu) are processed slowly (milliseconds to seconds). Hence, it is good practice to invoke the control plane at larger timescales. An attacker could exploit this

Attacks	Goal	Attacker privileges	Data plane systems
CPU exhaustion	DDoS	Compromised hosts, Compromised storage servers	NetHCF [21], ACL, NAT, Poise [18]
Memory saturation	DDoS	Compromised hosts	Poise [18], Silkroad [27], NetCache [8]
Performance degradation	DoS, Poor QoS	Compromised hosts, MiTM	Blink [14], HULA [19]
Corrupt network stats	Evasion (miss attack detection), Poisoning (mislead learning algorithms)	Compromised hosts	Flowradar [22], Lossradar [23], UnivMon [25]

Table 1: Details of possible attacks on data plane systems

and craft inputs exclusively to invoke the control plane continuously, successfully depleting switch CPU resources.

Memory saturation. The data plane has two types of state: stateless match-action table rules and stateful registers. These states are maintained in either SRAM or TCAM or both, where TCAM holds a few thousand rules [12] and the SRAM size is in the order of MBs (50MB-100MB) [27]. In some systems, control-plane agents continuously update match-action rules whenever there is a signal from the data plane. Consider, for example, the NetCache [8] system's key classification mentioned above. An adversary can take advantage of this process and craft traffic with an objective to access distinct keys, each, for more than its threshold. This can eventually lead to rule installation for too many keys, discarding legitimate key requests due to lack of space, thereby increasing end-to-end latency. Similarly, stateful registers are usually pre-allocated and blocks of registers are assigned at run-time whenever certain conditions are met (*e.g.*, allocate blocks on a new flow arrival). An adversary can inject spoofed flows frequently with the goal to saturate stateful registers, thereby leading to a memory saturation attack. The consequences of memory saturation are unpredictable – the switch may behave abnormally (*e.g.*, drop incoming packets), or, the attacker may successfully evade defenders.

Performance degradation. To quickly handle dynamic network events (e.g., congestion, link/switch failure), some systems take decisions immediately in the data plane before the control plane responds. For example, HULA [19] listens to network feedback carried in probes and quickly responds with decisions (e.g., re-route traffic) to avoid performance degradation. However, in such cases, an adversary could potentially exploit the semantics of the control logic to craft network inputs (e.g., modify/delay probes) and influence decisions in the data plane. This will potentially lead to negatively impacting the behavior of a large portion of the traffic and hence degrading aggregate network performance.

Corrupt network statistics. Network management tasks such as traffic engineering, security, and accounting heavily rely on network telemetry data (*e.g.*, avg, min, count estimations) generated by data-plane monitoring primitives. To perform monitoring at line rate, compact data structures like bloom filter and its variants (*e.g.*, counting bloom filters) are commonly used because of space efficiency and low per-packet computation cost. Further, to keep a low false positives rate, using multiple bloom filters indexed by several hash functions (say k) has become common practice. For instance, to check if a packet belongs to an old flow, the packet's 5-tuple flow key is hashed k times and if entries at all k locations are set to 1, then the packet is marked as an old flow. Otherwise, the packet is marked as a new flow. Previous studies on attacking bloom filters [13] have shown that an adversary can pollute bloom filters by generating traffic with an objective to increase the number of 1's. By doing so, the false positives rate increases significantly, thus misleading learning algorithms and successfully evading from being detected. Another possible attempt is to make membership queries expensive by generating traffic in such a way that flow keys evaluate to 0 for the last hash function and 1 for others – thus increasing query execution time.

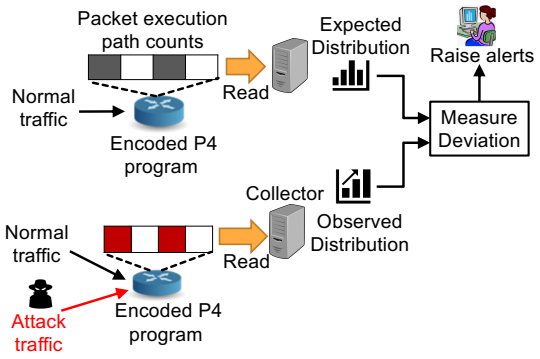


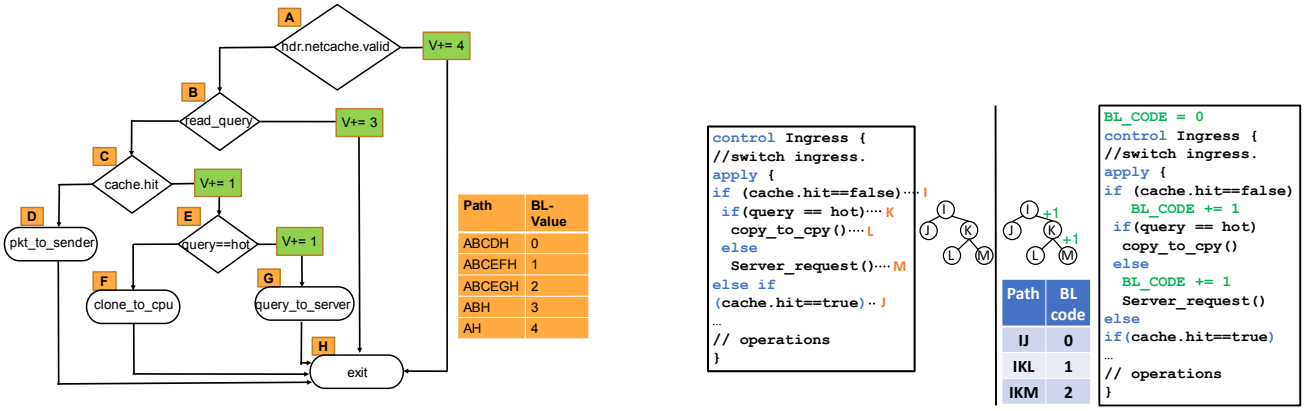
Figure 1: System workflow

3 DESIGN

The main objective of this work is to detect at run-time if a data-plane system is under attack. As a first step, we present an anomaly detection system, which performs statistical analysis on the packet execution path distribution in the switch data plane.

Figure 1 shows our approach to detect anomalies. It can broadly be divided into three phases:

- **Model expected behavior:** Our system monitors traffic under normal conditions where an encoded p4 program running in the data plane tracks and maintains packet execution path statistics. The control plane periodically collects statistics and constructs a packet execution path distribution, which will be the expected distribution.
- **Capture observed behavior:** At run time, we capture packet execution path statistics for each time window, using which, the control plane constructs a packet execution path distribution – the observed distribution.
- **Measure deviation:** We calculate the deviation between the expected and observed distributions using a statistical testing technique called the chi-square test and raise alerts if significant deviation is observed.



(a) **NetCache's [8] read query CFG**

(b) **P4 program augmented using Ball-Larus algorithm**

Figure 2: Tracking packet execution path using Ball-Larus algorithm

In the following sections, we provide details related to (1) tracking packet execution path, (2) maintaining path statistics, and (3) measuring deviation using the chi-square test method.

3.1 Tracking packet execution path

A packet might take any path in a P4 program. To track every path – the tables applied and the actions executed – we need to augment the original p4 program such that per-packet state like the Packet Header Vector (PHV) is updated as the packet travels. However, since the PHV is a scarce resource, the path encoding technique should operate under the resource constraints of programmable data planes. Recently, p4track [20] has shown that the Ball-Larus encoding technique [10], is a promising fit for tracking packet execution paths in P4 programs. This is because P4 programs are loop-free and the encoding does not require sophisticated updates, and the addition operation on the path variable is sufficient. We use this technique for tracking paths and construct the expected and observed distributions atop this technique.

We briefly present the core idea of the Ball-Larus encoding technique here.

Since P4 programs are loop-free, the control flow graph (CFG) of a P4 program is a Directed Acyclic Graph (DAG) where each node represents a program statement such as a table, action, or conditional. The Ball-Larus encoding algorithm performs reverse topological ordering of the DAG and assigns a label to each edge such that, as a given input packet transitions from one program statement to the next, the packet's path variable maintained in the PHV is added to the associated edge label. Finally, at the end of the DAG (or program) processing, the path variable value uniquely represents the path that the packet has taken in the program. Thus, if there are N possible paths, the path variable after program processing has a unique value between 0 to $N-1$.

Running example: Figure 2(a) shows the CFG of a NetCache [8] P4 program. It first checks whether it has received a valid NetCache query packet (node A in the CFG). If yes, it then checks whether the query response is in the cache by applying `read_query` table (node B). If there is a hit, the response packet is generated and sent to the sender (node C and node D). Otherwise, it checks whether the query has been seen many times before. If yes, the query becomes hot and a signal is sent to the local switch CPU, which in turn installs the response in the cache (node E and node F). Otherwise, the query is sent to the destination server (node G). Finally, the exit action marks the end of the read query in the CFG.

When we run the Ball-Larus (BL) algorithm on the CFG, it assigns labels 1, 1, 3, 4 to edges E–>G, C–>E, B–>H and A–>H, respectively. As a packet traverses an edge, the associated edge label is added to the per-packet path variable V. The CFG has 5 different paths from the root node (A) to leaf node (node H). When a program completes packet processing, the value V must be in the range of 0 to 4 as depicted in the table in Figure 2(a).

Multi-variable Ball-Larus encoding. However, a naively adapted Ball-Larus encoding can have significant overhead in terms of action complexity and the number of pipeline stages. More specifically, for large P4 programs such as `switch.p4` [7] the path variable size can go as large as a few hundred bits, making integer arithmetic at line rate a challenge. Another concern is that updating the same path variable by multiple tables would force the compiler to put the tables across multiples stages, which would otherwise be mapped to the same stage: thus increasing the number of stages. P4track [20] handles both these challenges by using multiple variables for Ball-Larus encoding (MVBL). The key idea is to carefully partition the original DAG and assign a variable to each partition (*i.e.*, sub-DAG), and track the packet execution path separately. By doing so, since each sub-DAG has fewer paths compared to the original DAG, the path variable size would stay within the limits of the arithmetic operands. Moreover, it allocates different path variables to the tables mapped to the same stage so that they co-exist in the same stage after Ball-Larus encoding.

3.2 Maintain packet execution path statistics

We use a hash table to maintain per-path statistics. The path variable value (V) is hashed and the hash value is used as a key to lookup an entry in the hash table. If the key is present, we increment the existing counter. Otherwise, we allocate and initialize an entry with the key. We leverage stateful registers in SRAM to store and update

hash table entries at line rate. One might argue that the number of paths would explode in a large P4 program, thus requiring large memory to keep the hash collision probability low. However, in practice, packets traverse a small subset of all possible paths – thus, the memory required is reasonably low compared to the SRAM available in the switches [27]. In our experiments, we observe that packets in Netcache.p4 [8] traverse around 10 paths out of 600 possible paths, and it is 60 out of 10^8 in Blink.p4 [14]. We also observed that the number of paths across different windows remains the same most of the time and if it changes, the percentage change is very small. So, given a hash table of size 500KB, with 100 keys, the expected number of collisions is 0.077 which is in the generally recommended limits of 0.1% of the number of keys (*i.e.*, 100).

Moreover, our control plane agent resets stale entries whose counter value is the same in two subsequent windows; this clean-up process further reduces the collision rate. If MVBL encoding is in place, for multiple path variables, one for each sub-DAG, it requires additional memory to maintain statistics such that the collision rate will be under the limits. One may also consider the count-min sketch data structure [3] to achieve an acceptable trade-off between memory and accuracy at the cost of packet-processing complexity. We will explore this alternative data structure in our future work.

3.3 Model expected behavior

To model expected behavior, we deploy P4 program annotated with Ball-Larus algorithm and hash table on the switch, and collect packet execution path distribution of real traffic over a period of time (training phase). The distribution captures number of packets on each execution path in the P4 program. We build expected distribution, say E, from periodically collected per-path statistics maintained in the hash table in the data plane. Formally, assume there are K paths in a given time window w_i, the expected distribution E_{w_i} for the current window w_i is defined as:

$$E_{w_i} = (E_{w_i}(1), E_{w_i}(2), E_{w_i}(3), ...E_{w_i}(k)) \tag{1}$$

where $E_{w_i}(1)$, $E_{w_i}(2)$, $E_{w_i}(3)$ ... $E_{w_i}(k)$ denote frequencies corresponding to paths 1, 2 ... k, respectively. Let S_i be the total packets seen in this window, given by

$$S_i = \sum_{j=1}^{k} E_{w_i}(j) \tag{2}$$

After capturing E_w and S for each window, we model the probability corresponding to each path in order to define the NULL hypothesis. We define NULL hypothesis in terms of set M given by

$$M = (p_1, p_2, p_3 ... p_k) \tag{3}$$

where p_1, p_2, p_3 ... p_k denotes probability corresponding to paths 1, 2 ... k respectively. To be specific, if there are n windows, the probability p_j corresponding to path j is given by,

$$p_j = \frac{\sum_{i=1}^{n}(E_{w_i}(j))}{\sum_{i=1}^{n}(S_i)} \tag{4}$$

In summary, we represent expected behavior in terms of M and use it as a reference to validate the observed distribution, which we discuss next.

3.4 Validate observed with expected

Given a benchmark for expected traffic distribution, we ought to identify whether observed traffic falls outside a regular pattern. To do so, after the training phase, we collect per-path traffic statistics maintained in the hash table and compare the observed distribution with M. Formally, for a given window w_i, observed distribution O is defined as:

$$O_{w_i} = (O_{w_i}(1), O_{w_i}(2), O_{w_i}(3), ...O_{w_i}(L)) \tag{5}$$

where $O_{w_i}(1), O_{w_i}(2), ... O_{w_i}(L)$ denote number of packets observed on L paths. Let S_i be the total packets seen in w_i, given by the equation:

$$S_i = \sum_{j=1}^{L} O_{w_i}(j) \tag{6}$$

We measure deviation between observed distribution and expected distribution using a chi-squared test [2]. A chi-squared test (χ^2) for independence compares two frequency distributions (M and O) to see if they are related. To be specific, we define (χ^2) for window i as:

$$\chi^2_{w_i} = \sum_{j=1}^{K} \frac{(E_{w_i}(j) - O_{w_i}(j))^2}{E_{w_i}(j)} \tag{7}$$

where expected count of path j, $E_{w_i}(j)$ is defined as:

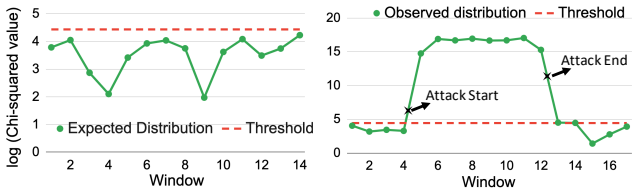
$$E_{w_i}(j) = p_j * S_i \tag{8}$$

A *small* chi-square value means that the observed distribution data fits expected data extremely well. On the other hand, a *large* chi-square value means that the observed data deviates significantly from the expected data.

It is possible that the set of observed paths (L) vary from the expected set of paths (K) in M. To handle this, we consider two scenarios: $L \subseteq K$ and $L \nsubseteq K$. When $L \subseteq K$ the observed packet count is zero for those paths in K-L set. On the other hand, when $L \nsubseteq K$ we add a small threshold t_i to the expected set L-K for the chi-squared value to be infinite. We report that the observed distribution deviates from the expected when the standard chi-square value is improbably large according to the chi-squared p-value table [1].

4 PRELIMINARY EXPERIMENTS

Setup. To demonstrate the feasibility of our idea, we conduct an initial experiment on the NetCache [8] P4 program. We construct the NetCache.p4 program CFG from .json and .dot files generated by the p4c compiler [5]. Next, we run the Ball-Larus algorithm on the CFG and annotate actions and conditional statements in the original P4 program. Currently, the P4 program is annotated manually. We can automate this step by parsing the P4 program line by line and append the action and condition blocks with code that updates BL variable. Finally, we add a hash table at the end of packet processing and the annotated P4 program is deployed on the P4 switch. Our experiments were conducted in Mininet v2.3.0 with one P4 switch, eight key-value storage hosts, one client requesting keys, one controller updating the switch cache with hotkeys, and a collector which reads packet execution path statistics and measure deviation. We run the experiment multiple times using a bash script with each experiment containing around 40000 queries, taking 3 minutes to run. The code for BL encoding, collection, and validation is written in Python.



(a) **With normal traffic** (b) **With normal and attack traffic**

Figure 3: Chi-square test before and during the attack

Memory saturation attack. As mentioned previously, the NetCache system classifies a key as "hot" if the key is requested beyond a certain threshold. Once the attacker discovers the threshold, he/she can craft traffic with distinct hot keys to cause: (1) installation of unwanted keys in cache; and (2) eviction of legitimate keys when the cache is full, thus increasing response time for legitimate traffic.

Experiments. First, we send normal traffic, collect packet execution path counters for multiple windows, and model expected distribution as described in §3.3. The normal traffic is following Zipf distribution on key requests where the frequency of a key request is inversely proportional to its rank such that 10% of the keys account for 60-90% of queries [9]. After building the expected distribution model, we inject attack traffic, collect path statistics, and perform the chi-squared test. Note that the keys in the attack traffic are randomly generated with the intent to create more hotkeys. The cache size is set to 8 *KB*.

Results. To interpret the chi-square test results, we compare the actual chi-squared values with the pre-set chi-squared threshold. The dashed red line in Figure 3(a) and Figure 3(b) shows that the chi-squared p-value is set to 21 — this aligns with the standard threshold [1] when the number of classes (or paths observed) is less than 11. By comparison, if the actual/observed values are below the given chi-square threshold, it denotes that the system behavior is as expected. If the actual/observed values rise above the threshold value, we can conclude that the system is displaying abnormal behavior.

The dotted green line in Figure 3(a) indicates that when there is no attack traffic the chi-squared value is below the threshold. When the system is under attack, as depicted in Figure 3(b), the chi-squared value increases and saturates; here we observe eviction happens to accommodate space for new hotkeys. This experiment shows that our anomaly detection approach is able to detect abnormal behavior in this system.

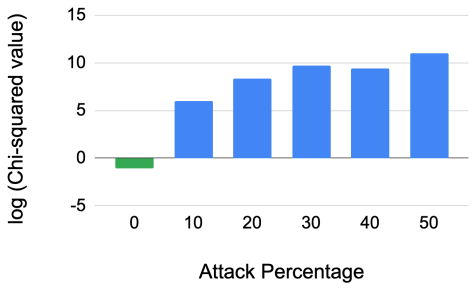


Figure 4: Mean Chi-Squared Value Above Threshold

Figure 4 summarizes the results of experiments carried out over multiple runs for NetCache. The X-axis represents the percentage of attack queries and the Y-axis depicts how far the actual chi-squared value is from the threshold chi-squared value. The difference between the actual and the threshold is averaged over multiple runs. We vary the percentage of attack queries from 0 to 50 and record the mean value for each scenario. When there are no attack queries (*i.e.*, 0 attack percentage), the value turns out to be negative, as depicted by the green bar plot in Figure 4, denoting that there was no attack. When the percentage of attack queries increases, we see a positive mean value, depicted by the blue bar plots here, which indicate our system is successfully able to detect attacks even when the attack traffic percentage is low (*e.g.*, 10).

5 RELATED WORK

A recent work [16, 17] proposed a probabilistic program profiler that uses symbolic execution with model counting to analyze program behaviors for adversarial testing. In contrast, we model path execution probability distribution at run-time from real traces and use the model to check deviation. We believe our work nicely complements offline profilers as it could miss corner cases during testing or expensive to explore all possible paths for large programs in useful time. Another line of work [24, 28, 29] explores automatic verification of various properties about P4 programs using techniques such as static analysis or symbolic execution. However, these tools still operate at the level of the P4 program. That is, they can verify if the software logic is bug-free for a specific set of bugs, but cannot find whether the p4 programs are under the influence of attackers at run-time. Our work is inspired by the security challenges presented in self-driving networks using programmable data planes [26].

6 SUMMARY AND FUTURE WORK.

Our preliminary experiments indicate our methodology is able to detect abnormal behavior due to a memory saturation attack on the NetCache system. As part of ongoing work, we plan to: (1) improve attacker model, especially by distinguishing between data-plane systems designed for data centers and the Internet; (2) validate our detection technique on a wide range of data-plane systems [14, 18, 19, 21, 22, 27] as shown in Table 1; (3) avoid false alerts through continuous learning and adapt our models to the new normal; (4) study attack surface of the proposed system and make the design robust to the possible attacks; (5) study whether an adaptive attacker able to evade detection by generating traffic strategically; (6) handle non-stationary normal events (*e.g.*, traffic spikes, system-specific traffic abnormalities); and (7) evaluate the system using real traces and associated overhead in terms of resources consumed.

7 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful feedback; we also thank Liang Wang, Rinku Shah, Shiv Kumar, and Ranjitha for their valuable comments on the earlier drafts and for their participation in the discussions. This work is supported by a startup grant awarded by IIT Hyderabad.

REFERENCES

[1] 2021. Chi-square p-table. https://people.richland.edu/james/lecture/m170/tbl-chi.html. (2021). [Online; accessed 30-May-2021].

[2] 2021. Chi-squared test — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Pearson%27s_chi-squared_test. (2021). [Online; accessed 30-May-2021].

[3] 2021. Count–min sketch — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Count_min_sketch. (2021). [Online; accessed 30-May-2021].

[4] 2021. Network Programming Language. https://nplang.org/. (2021). [Online; accessed 30-May-2021].

[5] 2021. P4 compiler. https://github.com/p4lang/p4c. (2021). [Online; accessed 30-May-2021].

[6] 2021. P4.org. https://p4.org/. (2021). [Online; accessed 30-May-2021].

[7] 2021. Switch.p4-program. https://github.com/p4lang/switch. (2021). [Online; accessed 30-May-2021].

[8] Xin Jin 0008, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS*.

[10] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *MICRO*.

[11] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM*.

[12] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud Scale Load Balancing with Hardware and Software. In *SIGCOMM*.

[13] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. 2014. The Power of Evil Choices in Bloom Filters. In *IEEE IFIP*.

[14] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *NSDI*.

[15] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A Programmable System for Performance-aware Routing. In *NSDI*.

[16] Qiao Kang, Jiarong Xing, and Ang Chen. 2019. Automated Attack Discovery in Data Plane Systems. In *USENIX CSET*.

[17] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. 2021. Probabilistic Profiling of Stateful Data Planes for Adversarial Testing. In *ACM ASPLOS*.

[18] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable In-Network Security for Context-aware BYOD Policies. In *USENIX Security*.

[19] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM SIGCOMM SOSR*.

[20] Suriya Kodeswaran, Mina Tahmasbi Arashloo, Praveen Tammana, and Jennifer Rexford. 2020. Tracking P4 Program Execution in the Data Plane. In *ACM SIGCOMM SOSR*.

[21] Guanyu Li, Menghao Zhang, Chang Liu, Xiao Kong, Ang Chen, Guofei Gu, and Haixin Duan. 2019. NETHCF: Enabling Line-rate and Adaptive Spoofed IP Traffic Filtering. In *IEEE ICNP*.

[22] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *NSDI*.

[23] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *ACM CoNEXT*.

[24] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4V: Practical Verification for Programmable Data Planes. In *SIGCOMM*.

[25] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*.

[26] Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, and Laurent Vanbever. 2019. (Self) Driving Under the Influence: Intoxicating Adversarial Network Inputs. In *ACM HotNets*.

[27] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *SIGCOMM*.

[28] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. P4pktgen: Automated Test Case Generation for P4 Programs. In *ACM SIGCOMM SOSR*.

[29] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *SIGCOMM*.