# In-Network Probabilistic Monitoring Primitives under the Influence of Adversarial Network Inputs

Harish S A*
IIT Hyderabad
India

K Shiv Kumar†
IIT Hyderabad
India

Anibrata Majee
IIT Hyderabad
India

Amogh Bedarakota
IIT Hyderabad
India

Praveen Tammana
IIT Hyderabad
India

Pravein Govindan Kannan
IBM Research
India

Rinku Shah
IIIT Delhi
India

## ABSTRACT

Network management tasks heavily rely on network telemetry data. Programmable data planes provide novel ways to collect this telemetry data efficiently using probabilistic data structures like bloom filters and their variants. Despite the benefits of the data structures (and associated data plane primitives), their exposure increases the attack surface. That is, they are at risk of adversarial network inputs.

In this work, we examine the effects of adversarial network inputs to bloom filters that are integral to data plane primitives. Bloom filters are probabilistic and inherently susceptible to pollution attacks which increase their false positive rates. To quantify the impact, we demonstrate the feasibility of pollution attacks on FlowRadar, a network monitoring and debugging system that employs a data plane primitive to collect traffic statistics. We observe that the adversary can corrupt traffic statistics with a few well-crafted malicious flows (tens of flows), leading to a 99% drop in the accuracy of the core functionality of the FlowRadar system.

## CCS CONCEPTS

• **Networks** → **In-network processing**; **Network monitoring**; *Programmable networks*; • **Security and privacy** → **Network security**.

## KEYWORDS

Network security, Programmable data planes, Probabilistic data structures, Bloom filters, Adversarial influence

**ACM Reference Format:**
Harish S A, K Shiv Kumar, Anibrata Majee, Amogh Bedarakota, Praveen Tammana, Pravein Govindan Kannan, and Rinku Shah. 2023. In-Network Probabilistic Monitoring Primitives under the Influence of Adversarial Network Inputs. In *7th Asia-Pacific Workshop on Networking (APNET 2023), June 29–30, 2023, Hong Kong, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3600061.3600086

*Both the authors have contributed equally to the work
†Both the authors have contributed equally to the work


*APNET 2023, June 29–30, 2023, Hong Kong, China*

ACM ISBN 979-8-4007-0782-7/23/06…$15.00
https://doi.org/10.1145/3600061.3600086

## 1 INTRODUCTION

The introduction of programmable data planes (*i.e.*, switches, smartNICs, FPGAs) and a high-level language to program them (*i.e.*, P4 [12]) has spurred in-network systems [5, 15, 20, 22, 23, 25, 27–31, 33, 39, 40, 44–46] benefiting a range of management tasks and eliciting interest from both the industry and academia. Network management tasks such as monitoring [20, 29, 30, 40, 44], load balancing [27, 33], routing [7, 22, 23], security [15, 25, 28, 31, 39, 45, 46], and caching [5] heavily rely on network telemetry data (*e.g.*, packet counts, delay) collected by data plane monitoring primitives.

However, the design of these data plane monitoring primitives depends on memory and per-packet processing time constraints imposed by network devices (*i.e.*, network switches [6], smartNICs [26]). More specifically, to comply with the space constraints and perform monitoring at line rates, compact hash-based probabilistic data structures like bloom filters [11] and its variants (count-min-sketches [18], invertible bloom lookup tables [21]) are employed.

Bloom filters are preferred for their space efficiency and low per-packet computation cost. Essentially, it is a lightweight hash-based probabilistic data structure that can determine the membership of an element (*i.e.*, traffic flows in the network context) to a set in constant time. However, they are prone to false positives due to hash collisions [19]. That is, a new flow could be falsely identified as an existing flow. Generally, using multiple bloom filters indexed by several hash functions is a common practice to keep false positive rates low.

Regardless, an adversary (*i.e.*, a malicious actor who intends to cause harm to the system) can aim to increase the false positive probability by polluting the bloom filter [19, 37]. The underlying motive is to trick the bloom filter into incorrectly reporting the presence of non-existent elements, thereby corrupting the collected network statistics. Different network applications can tolerate varied thresholds of false positive rate (FPR) but the general premise holds true: an increase in FPR leads to application misbehavior [13, 19].

To the best of our knowledge, [24, 32] are the first works that study recent data-driven networked systems under varied adversarial network inputs. However, the precise impacts of targeted pollution attacks on these systems' bloom filter-based probabilistic data structures are unclear. Enumerating the impact of such targeted pollution attacks can be rewarding due to its potential applicability across multiple systems that use the same data structures [5, 7, 14, 25, 29, 30, 33, 44, 45].

Essentially we probe the following question throughout the paper: *"What are the negative impacts of polluting probabilistic compact data structures that drive data plane monitoring primitives employed by the data-driven networked systems [5, 7, 14, 25, 29, 30, 33, 44, 45]?"*. With different bloom filter variants employed in these systems, they may be susceptible to pollution attacks. To study such attacks further, we pick FlowRadar [29], a network monitoring system and carefully study and demonstrate the impact of pollution attacks on its bloom filter-based data structures.

Towards this goal, we first elaborate the threat model in §3. The idea is to define the capabilities and influence of the attacker that enable us to work with feasible attack vectors. Next, we briefly explain the role of bloom filters in FlowRadar [29] and subsequently extend the analysis through concrete attacks on its bloom filters. That is, we place FlowRadar in adversarial settings, which generate malicious flows to pollute its bloom filters and analyze its impact on the accuracy of its operations.

The adversarial intent is to either increase the false positive probability or corrupt existing entries in the bloom filters employed. We demonstrate the feasibility of attacks under two adversarial models: (1) Chosen Insertion Adversary (CIA) and (2) Query Only Adversary (QOA) (more in §3). From our preliminary findings, we see that even a few but carefully crafted adversarial flows corrupt a large quantum of network statistics. Colloquially, the attacker seems to get *the bang for his buck*, further emphasizing the need to study such pollution attacks in depth.

We briefly discuss mitigatory strategies and best practices that can detect and defend against such attacks. Further, the scope of our work possibly extends to bloom filters internally employed by network switches [3] and the Linux kernel eBPF code [4, 42]. This work is expected to serve as a template for exploring adversarial influence on other contemporary data plane monitoring primitives that use bloom filter-based probabilistic data structures.

## 2 BLOOM FILTERS

In the context of networking applications, bloom filters are generally used to identify and keep track of traffic flows. Consider a bloom filter of size 12 and two hash functions (*i.e.*, $H_1$ and $H_2$) as shown in Figure 1 which keeps track of traffic flows. A flow is identified by its 5−tuple subset of its header fields (*i.e.*, source IP, destination IP, source port, destination port, and protocol). A hash of the 5−tuple (also called a flowID) is generated and mapped to one of the bloom filter indices which is set to '1' indicating the presence of the flow.

In Figure 1, flows $f_1$, $f_2$, and $f_3$ have already been inserted into the bloom filter as depicted by the corresponding arrows pointing to 'set' indices (*i.e.*, 1). Now, to check the membership of a flow (also called a 'query') in the bloom filter, the indices are calculated

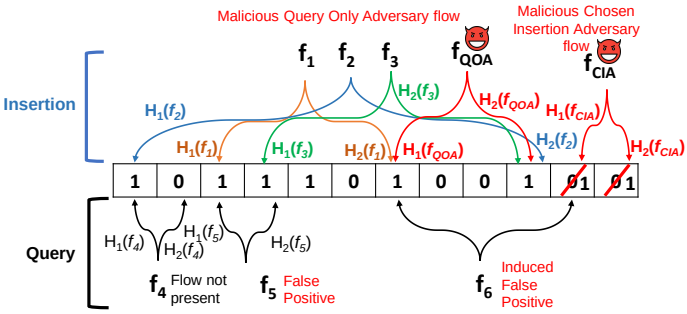



**Figure 1: Bloom filter of size 12 and 2 hash functions: under normal and malicious action**

by using the same two hash functions. In Figure 1, flow $f_4$ is not present in the bloom filter owing to hash $H_2$ pointing to an unset (*i.e.*, 0) index. That is, only if all the calculated indices are set bits (*i.e.*, 1), the flow is considered to be a member. Consider the query for flow $f_5$ in Figure 1. Both the calculated hash indices point to 'set' cells (*i.e.*, 1) but they were set by flows $f_1$ and $f_3$. That is, the flow $f_5$ is not inserted in the bloom filter, but its membership query returns *true*. Such a scenario is a false positive and is unavoidable in a bloom filter due to hash collisions [19]. Also, other bloom filter variants like count-min-sketches [18], invertible bloom lookup tables [21] are susceptible to false positives. Generally, to keep a low false positive rate, using multiple bloom filters indexed by several hash functions is a common practice. The false positive probability $f$ is a function of the expected number of items $n$, the size of the bloom filter $m$ and the number of hash functions $k$ given as:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \tag{1}$$

## 3 THREAT MODEL

Here, we explain who the adversary is, his privileges, his objectives and the adversarial models under which he crafts malicious traffic.

### 3.1 Adversarial privileges

We assume that an adversary knows everything about the system implementation (*i.e.*, parameters and algorithm) except for cryptographic secrets [36]. More specifically, the adversary is knowledgeable of the following details: (1) the size of the bloom filter (2) the number of hash functions, and (3) the type of hash function being used. This is a fair assumption, considering most implementations are open-sourced and publicly available. However, even if not, works like [41] infer the hash function details using collisions. We consider two types of adversaries [19]: (1) Chosen Insertion Adversary (CIA) and (2) Query Only Adversary (QOA).

**Chosen Insertion Adversary (CIA).** The objective of CIA is to increase the number of 'set' bits in the bloom filter. In Figure 1, $f_{CIA}$ is a malicious flow crafted by a CIA. The intent is to maximize the number of 'set' bits, and thus, all of the malicious flows he generates map to different locations in the bloom filter. In the example, $f_{CIA}$ is responsible for $f_6$ being judged as a false positive. That is, the malicious flow has successfully induced a false positive. To do so,

the adversary requires knowledge of the size of the bloom filter and the type and number of hash functions. The adversary could be a malicious node by himself or compromise a benign node and has permission to craft and send malicious traffic to the in-network data plane primitive.

**Query Only Adversary (QOA).** The objective of QOA is to map to bits that are already 'set' in the bloom filter. In this work, we deviate slightly from the classical definition of QOA mentioned in [19]. In Figure 1, $f_{QOA}$ is a malicious flow crafted by a QOA which maps to locations that are already 'set'. Although it may seem counter-intuitive, it is done with the idea of polluting statistics collected behind the bloom filter. For instance, a heavy hitter detector would increment counters when encountering an existing flow. Here, the adversary requires knowledge of the size of the bloom filter, its partial state, the hash type, and the number of hash functions. To know the state of the bloom filter, the QOA may sniff traffic from a compromised node over a period of time (*i.e.*, Man-in-the-Middle). Through this, he maintains a local image of the bloom filter and crafts malicious flows that map to the 'set' bits.

Further nuances concerning malicious flow generation are discussed in §6.3.

## 4 BLOOM FILTERS IN FLOWRADAR

FlowRadar [29] is a network monitoring system that maintains flows and their counters in a data center environment. To realize this, FlowRadar encodes flow information using a data plane primitive that utilizes compact probabilistic data structures (*i.e.*, a bloom filter and a modified inverted bloom lookup table) abstracted as a *flowset* which leverage constant insertion and query time in a programmable switch. A remote collector is then used to aggregate *flowsets* from multiple switches every 10ms to perform network-wide decoding to extract the flow counters.

**Flowset.** The *flowset* ( Figure 2) is composed of two abstract data structures: (1) flow filter and (2) counting table. It uses non-cryptographic hash functions to map incoming flows to both of them. First, the flow filter is a vanilla bloom filter used to register 'new flows' and identify subsequent packets that belong to the registered flow. In Figure 2, flow $f$ is hashed using two hash functions $H_1$ and $H_2$ to set the bits in the flow filter, thus registering it as a 'new flow'. However, if all the hashed bit locations are already set, then the flow is considered as an 'old flow' which will be the case for subsequent packets from the same flow.

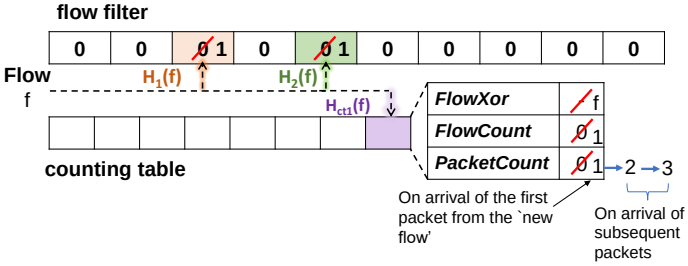



**Figure 2: FlowRadar data structure: *flowset***

The counting table is a modified invertible bloom lookup table [21] used to capture and maintain further fine-grained information about the flows. It is structured as an array of cells where each cell contains three fields: *FlowXor*, *FlowCount* and *PacketCount* as shown in Figure 2. *FlowXor* holds the flowIDs (*i.e.*, XORed 5-tuple values); *FlowCount* holds the number of flows mapped to the same cell; and *PacketCount* holds the number of packets observed. Upon the arrival of a 'new flow', all three fields of the counting table cell (identified by $H_{ct1}$) are updated as shown in Figure 2. *FlowXor* XORs any previous flowIDs and the current flowID. The *FlowCount* and *PacketCount* fields are incremented. Upon arrival of an 'old flow' (*i.e.*, subsequent packets from the flow), only the *PacketCount* field is incremented.

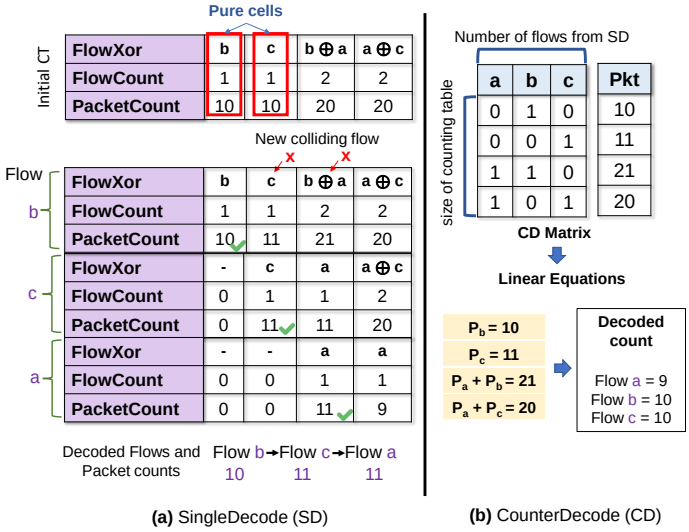



**Figure 3: FlowRadar operations**

**FlowRadar operation: SingleDecode (SD).** SingleDecode takes as input the counting table and outputs a subset of flowIDs that can be used for the CounterDecode operation. Consider the initial state of the counting table in Figure 3 with flows *a*, *b* and *c* having actual packet counts 10, 10, and 10 respectively. Consider that a subsequent flow *x* collides (marked in red color) with two locations that are already 'set'. Thus, this flow is treated as 'old' and only the *PacketCount* is incremented by '1' (*i.e.*, no update to *FlowXor* and *FlowCount*).

The SingleDecode algorithm searches for specific entries in the counting table called 'pure cells' as shown in Figure 3(a) This is characterized by a *FlowCount* field with a value 1. Flows *b* and *c* have 'pure cells' associated with them as shown in Figure 3(a). Upon detecting a 'pure cell', the algorithm subtracts its values from the other locations the flow is mapped to. The subtraction operation causes a cascading effect such that more 'pure cells' emerge as shown in Figure 3(a) and the flowIDs are decoded in this order: b → c → a. The obtained packet counts are discarded as they are inaccurate and only the flowIDs are sent as input to the CounterDecode process to minimize the packet count error.

**FlowRadar operation: CounterDecode (CD).** The CD takes as input the set of SingleDecoded flowIDs and the *PacketCount* field

values of all the counting table entries (*i.e.*, 10, 11, 21, 20) and outputs their packet counts with a reduced error margin. It does so by representing the flows as a CD matrix with dimensions (*counting table size* × *number of SingleDecoded flows*) as shown in Figure 3(b). Each column represents the cells of the counting table to which the flow is mapped to. For example, Flow *a* is mapped to the third and fourth cell and thus has 1 in the corresponding entries. Using the packet counts of the counting table, the matrix is represented as multivariable linear equations which are solved using approximation methods (*e.g.*, method of least squares [34]). The solution in this case reduces the packet count error (*i.e.*, from 2 wrong values to 1).

## 5 QUALITATIVE ANALYSIS

We explore the various scenarios that arise based on the order of malicious and benign flow arrival. In line with the same, we present two scenarios: (1) Malicious flow arrives before benign, and (2) Malicious flow arrives after benign. Each of these scenarios has 4 cases associated with them based on the locations the flows map to in the flowset's filters:

**Case I.** The malicious flow completely collides with only a single benign flow. that is, all their indices coincide.

**Case II.** The malicious flow partially collides with a single benign flow. Only some indices of the malicious flow coincide with a benign flow. The rest or at least one of its indices maps to an 'unset' location.

**Case III.** The malicious flow completely collides with multiple benign flows. All malicious flow indices coincide with distinct indices that belong to many benign flows. None of them map to 'unset' locations.

**Case IV.** The malicious flow partially collides with multiple benign flows. Only some indices of the malicious flow coincide with distinct indices that belong to many benign flows. The rest or at least one of its indices maps to an 'unset' location.

As per scenario 1, a malicious flow arriving first tends to occupy either some or all of the 'pure cells' that would have otherwise been occupied by the benign flow. On the contrary, in scenario 2, when a malicious flow arrives after a benign flow, only the statistics behind the already 'set' locations are affected. All cases in scenario 1 and cases II, IV of scenario 2 always register the malicious flow as a 'new' flow which affects all three fields of a counting table cell thus affecting both the SingleDecode and CounterDecode operation which could cause a harmful effect by rendering the benign flow undecodeable. On the contrary, a malicious flow registering as an 'old' flow (cases I and III of scenario 2) only affects the packet count field of existing benign flows affecting only the CounterDecode operation, which is less severe. Figure 4 summarizes the effects of all the cases across both the scenarios and specify which FlowRadar operation they affect.

## 6 EXPERIMENTS

The key question we investigate through the experiments is: *What is the impact on FlowRadar's ability to decode flow information under adversarial settings?*

| Case | Malicious flow mapping in flow filter | | | Malicious flow treated as | | FlowRadar operations affected | |
|---|---|---|---|---|---|---|---|
| | Single benign flow | Multiple benign flows | Unoccupied location | New flow | Old flow | SingleDecode | CounterDecode |
| I | ✔ | | | ✔ | | ✔ | |
| II | ✔ | | ✔ | ✔ | | ✔ | ✔ |
| III | | ✔ | | ✔ | | ✔ | ✔ |
| IV | | ✔ | ✔ | ✔ | | ✔ | ✔ |
| I | ✔ | | | | ✔ | | ✔ |
| II | ✔ | | ✔ | ✔ | | ✔ | ✔ |
| III | | ✔ | | | ✔ | | ✔ |
| IV | | ✔ | ✔ | ✔ | | ✔ | ✔ |

Legend: Scenario 1: Malicious flow before benign; Scenario 2: Malicious flow after benign

**Figure 4: Malicious flow action**

### 6.1 Experimental setup

We develop the core of FlowRadar logic using python. We do so in order to observe its packet processing behavior and access its bloom filters with ease at runtime. Pybloom, a python library [2] has been used to implement the $flowset$ (*i.e.*, both the flow filter and counting table). We model the $flowset$ parameters based on standard guidelines [21]. That is, given an expected number of incident flows (24k in our case), the size of the flow filter and counting table has been set to 0.24 and 0.03 million cells respectively, based on combined suggestions from [19, 21, 29]. We use non-cryptographic murmur hash functions [8]. Further, we assign 7 and 4 hash functions for the flow filter and counting table respectively. To simulate data center traffic, we perform our experiments only using the Wisconsin datacenter dataset [1], primarily because FlowRadar is targeted at data center environments. Ideally, FlowRadar exports its statistics (*i.e.*, clears *flowset*) every 10ms. We conduct our experiments to influence the bloom filter within this time frame.

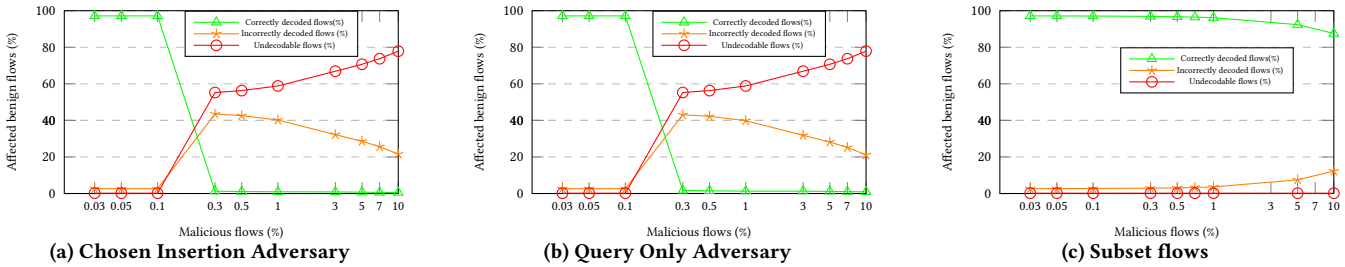
### 6.2 Metrics

To measure FlowRadar's decode accuracy loss under malicious influence, we first define three classifications based on the flowIDs and per-flow packet counts obtained after the decoding process: (1) Correctly decoded flows (2) Incorrectly decoded flows, and (3) Undecodable flows. A correctly decoded flow's packet count is equal to its actual packet count. An incorrectly decoded flow's packet count is not equal to the actual packet count. Further, if the reported count is off even by a single packet, we classify it as an incorrectly decoded flow (*i.e.*, no threshold). However, undecodable flows are those flows whose flowIDs are not decoded via the SingleDecode process and thus their packet counts are unobtainable. Note that the sum of all three flow classifications equals the total number of benign flows (*i.e.*, 24k flows).

It is to be noted that even under completely benign circumstances (*i.e.*, no malicious flows), both the SingleDecode and CounterDecode exhibit loss of flowIDs and per-flow packet counts, respectively. Thereby, both incorrectly decoded and undecodable flows are observed. We define the ground truth as: *"The number of flows reported as correctly decoded, incorrectly decoded, and undecodable by the FlowRadar's SingleDecode and CounterDecode operations under benign conditions"*.

### 6.3 Crafting malicious flows

In order to craft malicious flows, the adversary is assumed to have knowledge of the bloom filter implementation. More specifically, the




(a) Chosen Insertion Adversary (b) Query Only Adversary (c) Subset flows

**Figure 5: Impact of adversarial flows**

size of the bloom filter, the type and number of hash functions used are known by the adversary in order to craft the flows intelligently. Moreover, the methodology for crafting malicious flows differ for each variant of the adversary (as per §3). The crafted malicious flows are inserted at temporally random times, interlacing them with the benign flows of the dataset. Multiple runs of the experiments are performed by gradually increasing the percentage of malicious flows at each run to determine the adverse effects.

**Chosen Insertion Adversary.** The CIA generates random flowIDs (*i.e.*, 5-tuples) for the malicious flows such that they do not collide amongst themselves in the *flowset's* flow filter. By extension, each crafted flow map to new locations in the filter, thus affecting 'pure cells' and increasing the false positive rate. Also, there exists possibilities where the crafted flows can collide with benign flows as the crafting strategy does not factor in the cells occupied by benign flows. Such flows are expected to cause maximum havoc since they potentially affect all the fields of the counting table.

**Query Only Adversary.** The QOA generates random flowIDs (*i.e.*, 5-tuples) such that the malicious flows only map on to already set locations in the flow filter. Such flows are expected to corrupt the packet count statistics of multiple benign flows in the counting table with a higher probability. However, due to the order of flow arrival, it can also occupy new cells and affect 'pure cells', exhibiting effects similar to CIA.

In addition to the above strategies, We also generate malicious flows whose flowIDs are identical to the benign flows. Both CIA and QOA are capable of generating these flows which we call 'Subset'. It is to be noted that we employ brute force search techniques to craft malicious flows in all the cases.

## 6.4 Results and Discussion

We present our empirical analysis on the impact of both adversarial models (CIA & QOA) on FlowRadar in Figure 5. The x-axis denotes the percentage of malicious flows that were introduced with respect to the total number of benign flows (*i.e.*, 24k). We vary the malicious flow percentage till 10% (2472 malicious flows) in all our experiments. Please note that the malicious flows do not replace any existing benign flows but rather are additional. The y-axis denotes the percentage of affected benign flows.

**Observations.** The adverse effect on the decoding accuracy due to malicious flows crafted by CIA is shown in Figure 5 (a). Figure 5 (b) denotes the effect of QOA on decoding accuracy. For both these strategies, all the 8 cases (*i.e.*, 4 cases each under two scenarios in Figure 4) are applicable and thus exhibit similar plots. On the outset, we see that 0.3% malicious flows disrupt almost 99% of benign flows. More specifically, at 10% malicious flows, we see that almost 80% of benign flows are rendered undecodable, making the attack effective. Among both, CIA entails the least effort and thus, we claim it as the most effective strategy for the adversary.

Not surprisingly, for the trivial subset case (Figure 5 (c)), the effects are subdued. As per Figure 4, only case I of the two scenarios applies. However, since its flow IDs are the same as that of benign flows, we can only observe an increase in packet count (*i.e.*, affects CounterDecode). In line with that, we see only a gradual rise in incorrectly decoded flows. This clearly indicates that polluting just the packet counts is not sufficient for the adversary to *get a bang for his buck.*

**Q1. What explains the sharp cliff and rise, at the 0.3% malicious flow mark in Figure 5(a),(b)?**

The nature of bloom filters is such that the false positive rate does not show an exponential increase before a particular threshold. As explained in [19], the *birthday-paradox* renders the initial flows to most likely occupy different cells. Beyond a particular threshold, collisions begin to occur, whose effects we observe as a sharp cliff and rise at 0.3% malicious flows. *Moreover, due to its probabilistic nature, the interdependencies between flows caused by collisions create a cascading effect.* For instance, the SingleDecode operation is affected due to the overlapping of malicious flows with benign flows increasing the number of undecodable flows. Also, CounterDecode is heavily dependent on approximations to solve a large number of multi-variable linear equations. Thus, a few malicious flows that affect a few linear equations potentially cascade to a large number of benign flows observed as incorrectly decoded flows. We leave the theoretical analysis and enumeration of the relationship between the threshold and the bloom filter configuration to future works.

**Q2. How much does the temporal ordering of malicious flows with respect to benign flows matter?**

To find an answer, we performed experiments where the malicious flows were sent at the conclusion of all benign flows for QOA, affecting only packet counts. As expected, the number of undecodable flows remains constant and is equal to the ground truth in spite of the increase in malicious flows. However, we do observe a sharp rise in incorrectly decoded flows (due to cascading effect). Also, in practical scenarios, malicious flows are always interlaced with benign flows as the system is continuously operational in real-time.

**Summary.** From the experiments, we see that it is sufficient for the adversary to put in less effort to corrupt the FlowRadar statistics. Crafting malicious flows intelligently enough (*i.e.*, like CIA and QOA) can substantially compromise FlowRadar. This is a cause for

concern and highlights the glaringly vulnerable state of data plane primitives that use compact probabilistic data structures. We would like to briefly point out that another recent system RouteScout [7] uses a similar 'pure cell' based approach to calculate delay and packet loss completely in the data plane. Our analysis methodology appears to be well-extendable to that system.

## 7 MITIGATORY MEASURES

We brief some detection and defense measures. We leave their concrete analysis and implementation to future work.

**Best practices for system design.** A bloom filter that is directly exposed to raw traffic is at a high risk of adversarial manipulation. However, a well-placed bloom filter as seen in [25] ensures that security policies are applied on raw traffic before reaching the bloom filter, thus making it hard for adversaries. That is, there are layers of other deterministic processing that the traffic has to pass through. The adversarial window for attack reduces substantially. With that being said, the system design is use-case dependent, and therefore in some cases, it is unavoidable to place a bloom filter that is easily accessible to adversarial input.

**Observe traffic response.** One can argue that adversarially crafted malicious flows may not solicit a response from the target server. If there are no reply messages, then the flow is potentially malicious. A monitoring system in place to weed out the offending traffic could do well to stop such attacks. However, the question remains open as to where the monitoring system can be placed. One idea is to place a monitoring mechanism at the data plane primitive itself, but it increases the data plane overhead and could itself be vulnerable. Further compounding the difficulty is line rate observation of traffic, which can be challenging.

**Model benign bloom filter growth.** One idea is to train models to capture expected behavior that is the benign rate of growth of 'set' bits in the bloom filter. Any deviation from the expected behavior could be flagged as potentially malicious. However, constant monitoring of a data plane primitive is challenging in itself. A remote collector would periodically gather data, analyze and send signals for action back to the data plane.

**Ranking the flows.** The malicious flows could be subject to ranking metrics following ideas from works like SurgeProtector [9]. That is, based on some metric (*e.g.*, job size to packet size ratio), the incoming flows could be ranked. Even though this works best for temporal algorithmic complexity attacks, it is a matter of finding the right metric to rank the offending traffic and drop the packets from the least ranked ones to defend against spatial algorithmic complexity attacks like bloom filter pollution.

## 8 RELATED WORK

**Bloom filters in adversarial setting.** The works [16, 17, 19, 35, 37, 38] comprehensively analyze the impact of false positives caused by malicious inputs on bloom filters. They provide provable security treatment of bloom filter variants and focus on securing them cryptographically. Our work follows [19] to empirically analyze the impact of adversarial network inputs on the underlying data plane primitives that employ bloom filter variants.

**Adversarial analysis of data-driven network systems.** The works [10, 24, 32, 43] explore adversarial exploitation of data-driven data plane-based network systems. Our work complements their efforts and extends [32] to analyze data plane primitives that use bloom filter variants.

## 9 CONCLUSION AND FUTURE WORK

Bloom filter-based data plane primitives are integral to network monitoring and management systems. However, such systems are susceptible to adversarial network inputs. In this paper, we study the impact and the feasibility of two types of attacks, chosen insertion adversary and query-only adversary, on a network monitoring and debugging system called FlowRadar. We observe that an adversary can corrupt the traffic statistics collected by FlowRadar by generating a few crafted malicious flows (tens of flows), which would lead up to a 99% drop in accuracy. We analyze various malicious traffic generation scenarios and identify the most effective strategy for the adversary. In our future work, we plan to extend our analysis to other systems using similar data plane primitives (*e.g.*, RouteScout, NetCache) and develop detection and defense mechanisms that aim to protect a wide range of data plane primitives.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful feedback. We also thank Ranjitha for giving valuable feedback on the earlier drafts. This work is supported by National Security Council Secretariat (NSCS), India, and the Prime Minister's Research Fellowship (PMRF) program, India.

## REFERENCES

[1] 2010. *Data Set for IMC 2010 Data Center Measurement.* Retrieved March 2023 from https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html
[2] 2014. *pybloom 1.1.* Retrieved October 2022 from https://github.com/jaybaird/python-bloomfilter/
[3] 2017. *Service discovery optimization in a network based on bloom filter.* Retrieved March 2022 from https://patents.google.com/patent/US20170034285A1/en
[4] 2023. *kernel: missing initialization in bloom filter map in kernel bpf.* Retrieved March 2023 from https://bugzilla.redhat.com/show_bug.cgi?id=2048259
[5] 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*.
[6] Anurag Agrawal and Changhoon Kim. 2020. Intel tofino2 – A 12.9 tbps p4-programmable ethernet switch. In *IEEE HCS*.
[7] Maria Apostolaki, Ankit Singla, and Laurent Vanbever. 2021. Performance-driven internet path selection. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 41–53.
[8] Austin Appleby. 2008. Murmurhash. *URL https://sites. google. com/site/murmurhash* (2008).
[9] Nirav Atre, Hugo Sadok, Erica Chiang, Weina Wang, and Justine Sherry. 2022. SurgeProtector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 723–738.
[10] Conor Black and Sandra Scott-Hayward. 2021. Adversarial Exploitation of P4 Data Planes. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 508–514.
[11] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.
[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[13] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.
[14] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 15–29.

[15] Xiaoqi Chen, Hyojoon Kim, Javed M Aman, Willie Chang, Mack Lee, and Jennifer Rexford. 2020. Measuring TCP round-trip time in the data plane. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*. 35–41.

[16] Ken Christensen, Allen Roginsky, and Miguel Jimeno. 2010. A new analysis of the false positive rate of a bloom filter. *Inform. Process. Lett.* 110, 21 (2010), 944–949.

[17] David Clayton, Christopher Patton, and Thomas Shrimpton. 2019. Probabilistic data structures in adversarial environments. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1317–1334.

[18] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

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

[20] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research*. 61–74.

[21] Michael T Goodrich and Michael Mitzenmacher. 2011. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 792–799.

[22] 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 *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 161–176.

[23] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. 2020. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 701–721.

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

[25] 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*.

[26] Georgios P Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostić, and Gerald Q Maguire. 2021. What you need to know about (smart) network interface cards. In *Passive and Active Measurement: 22nd International Conference, PAM 2021, Virtual Event, March 29–April 1, 2021, Proceedings 22*. Springer, 319–336.

[27] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*. 1–12.

[28] 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 *2019 IEEE 27th international conference on network protocols (ICNP)*. IEEE, 1–12.

[29] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. {FlowRadar}: A Better {NetFlow} for Data Centers. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*. 311–324.

[30] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. Lossradar: Fast detection of lost packets in data center networks. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 481–495.

[31] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches.. In *USENIX Security Symposium*. 3829–3846.

[32] 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 *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 34–42.

[33] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 15–28.

[34] Steven J Miller. 2006. The method of least squares. *Mathematics Department Brown University* 8 (2006), 1–7.

[35] Moni Naor and Yogev Eylon. 2019. Bloom filters in adversarial environments. *ACM Transactions on Algorithms (TALG)* 15, 3 (2019), 1–30.

[36] Fabien AP Petitcolas. 2011. Kerckhoffs' Principle.

[37] Pedro Reviriego and Ori Rottenstreich. 2020. Pollution attacks on counting Bloom filters for black box adversaries. In *2020 16th International Conference on Network and Service Management (CNSM)*. IEEE, 1–7.

[38] Pedro Reviriego, Ori Rottenstreich, Shanshan Liu, and Fabrizio Lombardi. 2021. Analyzing and Assessing Pollution Attacks on Bloom Filters: Some Filters are More Vulnerable than Others. In *2021 17th International Conference on Network and Service Management (CNSM)*. IEEE, 491–499.

[39] Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle. 2020. Me love (SYN-) cookies: SYN flood mitigation in programmable data planes. *arXiv preprint arXiv:2003.03221* (2020).

[40] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous in-network round-trip time monitoring. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 473–485.

[41] R Joshua Tobin and David Malone. 2012. Hash pile ups: Using collisions to identify unknown hash functions. In *2012 7th International Conference on Risks and Security of Internet and Systems (CRiSIS)*. IEEE, 1–6.

[42] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. 2020. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[43] Liang Wang, Prateek Mittal, and Jennifer Rexford. 2022. Data-plane security applications in adversarial settings. *ACM SIGCOMM Computer Communication Review* 52, 2 (2022), 2–9.

[44] Weitao Wang, Praveen Tammana, Ang Chen, and TS Eugene Ng. 2020. Grasp the root causes in the data plane: Diagnosing latency problems with SpiderMon. In *Proceedings of the Symposium on SDN Research*. 55–61.

[45] Jiarong Xing, Qiao Kang, and Ang Chen. 2020. Netwarden: Mitigating network covert channels while preserving performance. In *USENIX Security*.

[46] Eder Ollora Zaballa, David Franco, Zifan Zhou, and Michael S Berger. 2020. P4Knocking: Offloading host-based firewall functionalities to the network. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 7–12.