ListGuard: Mitigating Pollution Attacks on In-Network Invertible Bloom Lookup Tables

Harish S A*, Vignesh S*, Sathwik Kodamarthi*, Bikraj Shrestha*, Gollapudi Sasank*, Mahanth Kumar Valluri*, Pravein G Kannan† and Praveen Tammana*
*IIT Hyderabad, India. †IBM Research, India.

***Abstract*—Compact probabilistic data structures (CPDS) such as Bloom filters, Invertible Bloom Lookup Tables (IBLTs), and sketches are widely used in network telemetry systems to collect traffic statistics directly in resource constrained, high-speed programmable data planes (*e.g.,* smartNICs and Intel Tofino switches). While CPDS enable scalable and efficient monitoring, they also increase the attack surface area. Adversaries can launch pollution attacks that corrupt stored statistics, leading to incorrect network management decisions.**

In this paper, we present *ListGuard*, a lightweight system that detects and recovers flow statistics in IBLT-based telemetry primitives affected by pollution attacks. Our key idea is to selectively sample additional flows and use them to resume the stalled decoding process without modifying the IBLT structure. Using realistic traces, we demonstrate attacks and show that *ListGuard* restores affected statistics with low memory overhead.

I. Introduction

High-speed programmable data planes like modern switch ASICs (*e.g.,* Intel Tofino [1]) and smartNICs (*e.g.,* Netronome Agilio [2]) have fundamentally transformed network monitoring and management. They enable line-rate packet processing while maintaining fine-grained state directly in the data plane. This capability has enabled a broad range of in-network systems for telemetry [3]–[7], load balancing [8], [9], caching [10], smart rerouting [11]–[13], and security [14]–[16]. At their core are monitoring primitives that continuously track traffic statistics (*e.g.,* flow IDs, packet counts, delays) in the data plane, which are analyzed to drive network decisions [17].

These primitives operate under tight memory and per-packet processing constraints imposed by programmable targets. To meet these constraints, systems rely on hash-based compact probabilistic data structures (CPDS) such as Bloom filters [18], Count-Min sketches [19], and Invertible Bloom Lookup Tables (IBLTs) [20]. We focus on IBLTs, which provide a space-efficient set representation while supporting a listing (decode) operation. IBLTs are widely used in prior systems [3], [4], [11], [21]–[24]. For instance, FlowRadar [3] encodes flow IDs and their packet counts using an IBLT.

An IBLT employs multiple hash functions to allocate incoming elements within a limited memory space. Inevitably, hash collisions occur, causing multiple elements to occupy the same memory locations [20], [25]. To recover stored elements, the listing (or decoding) procedure iteratively "peels" pure cells (cells containing exactly one item) until all elements are extracted. However, this process is fragile. An adversary can inject crafted items that pollute pure cells, reduce their count, and stall decoding. Even a small number of malicious insertions can trigger cascading failures that leave many elements undecodable [26], [27].

Such pollution attacks target listing failures, impacting systems like FlowRadar [3] and LossRadar [4], which results in loss of crucial flow and packet-level details necessary for network-wide decisions. Essentially, by disrupting flow statistics, attackers can cause load balancing or rerouting strategies to fail, and lead to misdiagnosis of network failures, thereby wasting resources on troubleshooting nonexistent issues. For example, an pollution attack on the IBLT of FlowRadar [3] might falsely indicate transient loops and black holes, throwing the network into a state of chaos. Moreover, fine-grained traffic analysis that require flow level details for anomaly detection like [28]–[30] will be compromised, receiving only partial flow and packet-level information.

Existing defenses against such attacks fall into three broad categories, each with limitations in high-speed programmable data planes: (1) Structural modifications to IBLTs aim to strengthen decoding guarantees, but typically incur substantial memory and computational overhead, making them impractical for resource-constrained targets [31]–[34]. (2) Traffic-level restrictions such as rate limiting reduce attacker influence but at the cost of visibility, as they may discard legitimate flow information especially since pollution attacks can operate at low volume. (3) Cryptographic hardening attempts to prevent collisions. However, even secure hash functions like SHA-256 [35] cannot eliminate collisions due to the pigeonhole principle [36]. Moreover, IBLTs intentionally apply multiple hash functions over a limited but shared memory space, meaning collisions are inherent to their design [20]. More advanced authentication mechanisms require operations (*e.g.,* complex hashing, key management, iterative computations) beyond the capabilities of programmable data planes [37].

Prior works [27], [38]–[40] have examined pollution attacks on other compact probabilistic data structures, including Bloom filters [18], Counting Bloom filters [41], sketches [19], and quotient filters [42]. However, the impact of such attacks on IBLT-based telemetry systems remains underexplored. In this work, we empirically extend pollution attacks to IBLTs under an adversary model (described in §III-A), demonstrating

ISBN 978-3-903176-82-9 © 2026 IFIP

that crafted insertions can render flow statistics in IBLT-based monitoring primitives undecodable. To safeguard its integrity, we present *ListGuard*, a lightweight system that detects pollution attacks and mitigates their impact by *recovering undecodable flows* from the IBLT's stalled state.

We tackle two challenges while designing *ListGuard*. First, it is difficult to ascertain whether the IBLT state has been altered by malicious traffic or updated by unusual traffic (*e.g.,* sudden burst). Second, there are memory constraints[1] and per-packet processing time constraints in high-speed programmable data planes. Therefore, it is crucial to minimize memory overheads while maximizing recovery accuracy and being amenable to implementation on the data plane under per-packet processing constraints. We address these challenges through a carefully designed sampling strategy with two complementary effects. (1) It samples a small fraction of strategically selected flows that help resume the stalled decoding process of an IBLT and recover flow statistics. (2) Since adversarial insertions share high collision characteristics, the same sampling mechanism naturally increases the likelihood that malicious flows are included in the sampled set. These can aid fine-grained security systems like deep packet inspectors, anomaly detection systems to identify the malicious flows from the sampled flows (more details in §VI and §VII).

The main contributions of this paper are:

- We demonstrate attacks on IBLT based telemetry system FlowRadar [3]. Due to the lack of attack datasets, we generate realistic attack traces by interlacing attack traffic with CAIDA traces so that traffic characteristics observed in a real network are preserved (§III).
- We design and develop *ListGuard*, a system to recover statistics from the affected IBLT in the event of a pollution attack (§V).
- We develop a prototype of *ListGuard* on the Netronome smartNIC [2] and conduct experiments with realistic traces [43]. We demonstrate the effectiveness of *ListGuard* in recovering statistics from polluted IBLTs (§X).

II. Background

Invertible Bloom Lookup Tables (IBLTs). An Invertible Bloom Lookup Table (IBLT) maintains a compact representation of a set while supporting efficient extraction (listing) operations [20]. An IBLT consists of an array of cells, each typically maintaining three fields: *KeySum*, *Count*, and *ValueSum* (Fig. 1). Incoming elements are mapped to multiple indices (cells) using independent hash functions. For each mapped cell, the element's identifier is XORed into *KeySum*, the *Count* is incremented, and the associated value (*e.g.,* packet count) is aggregated into *ValueSum*.

As an example, consider an IBLT of size 4 with two hash functions H_1 and H_2. Fig. 1 illustrates how three flows: a (1 packet), b (2 packets), and c (3 packets) update the structure. Due to hash collisions, multiple elements may contribute to the same cell which the XOR-based encoding compactly encodes.

[1] fast memory is a scarce resource shared with other core network functions.

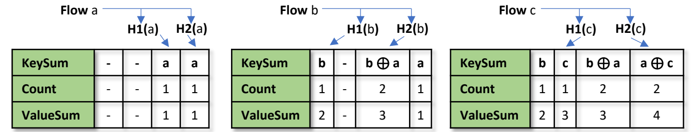


Fig. 1: **Inserting flows (elements) into an IBLT**

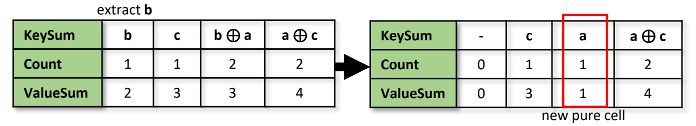


Fig. 2: **Decoding flows (or items) from an IBLT**

To retrieve stored elements, the IBLT performs a *listing* (or decoding) operation based on iterative peeling. Cells with *Count* $= 1$, referred to as *pure cells*, contain exactly one element and can be decoded directly. Once an element is extracted from a pure cell, its contribution is subtracted (via XOR) from its other hashed locations, potentially revealing new pure cells. For instance, in Fig. 2, extracting b or c exposes a new pure cell for a. This process continues until all elements are recovered or no pure cells remain. The success of decoding therefore critically depends on the availability of pure cells.

IBLTs in monitoring primitives. Table I summarizes systems that rely on IBLTs as a core design component. Owing to their space efficiency and constant time updates, IBLTs are well suited for resource constrained programmable data planes where memory and per-packet processing budgets are tightly bounded. In network telemetry systems such as FlowRadar [3] and LossRadar [4], IBLTs are embedded directly within data plane monitoring primitives and deployed on high-speed targets (*e.g.,* Intel Tofino switches [1] and smartNICs [2]). This design enables line-rate aggregation of flow or packet statistics without exporting raw traffic to a remote collector. For example, FlowRadar encodes flow identifiers and counters inside an IBLT during each epoch and later decodes them to diagnose failures such as blackholes or transient loops. The widespread adoption of IBLTs across these systems highlights their practical importance and motivates the need to ensure their robustness under adversarial inputs.

TABLE I: Existing systems using IBLTs

Function	Systems	Role of IBLT
Network monitoring	FlowRadar [3], LossRadar [4]	Capture and encode network flow or packet data for monitoring and aid in the diagnosis and root cause analysis of network failures.
Set reconciliation	Graphene [23], Diff [24], SmSR [22]	Encodes the symmetric difference between two sets stored in two identical IBLTs. The resulting difference can be listed (decoded).
Error correction	Biff [21]	Generates error correcting codes through the listing operation of IBLTs

III. Motivation: Attacks on IBLT

IBLTs are a key component of telemetry systems such as FlowRadar [3] and LossRadar [4], enabling fine-grained visibility into flow statistics directly in the data plane. Unlike traditional sampling based approaches, these systems maintain compact yet detailed state that supports the computation of

metrics such as entropy [44], number of distinct flows [45], and heavy hitters [46]. These metrics in turn drive network-wide functions including traffic engineering [47], DDoS detection [48], and anomaly detection [49]. The correctness of telemetry data is therefore critical to operational decisions.

However, deploying IBLTs in the data plane increases the attack surface, as they operate directly on raw network inputs (packets and flows). An adversary controlling a compromised host can inject carefully crafted flows to manipulate the IBLT state through pollution. Such attacks are highly efficient: we show that injecting as little as $1-2\%$ malicious flows can render more than 50% of the stored statistics undecodable (more in §X).

The consequences are severe: corrupted flow statistics can mislead load balancing and rerouting mechanisms, trigger incorrect fault diagnoses, or falsely indicate blackholes and transient loops in systems like FlowRadar and LossRadar. Beyond immediate disruption, polluting in-network telemetry systems create blind spots in network visibility that can facilitate secondary attacks, like DDoS or data exfiltration. Repeated attacks may ultimately erode trust in data plane based telemetry, forcing operators toward less efficient monitoring alternatives, increasing operational costs and potentially degrading network performance. In the following sections, we formalize the threat model and demonstrate how adversaries can execute pollution attacks against IBLT based data plane telemetry systems.

A. Threat model

We consider a network logically partitioned into a control plane and a data plane. Following commonly adopted assumptions in programmable network security [50], we assume: (i) the control plane is uncompromised, (ii) control-data plane channels are secure, but (iii) the data plane including end hosts and interconnecting links may be compromised. An adversary may compromise a host [51]–[53] and perform Man-in-The-Middle (MitM) actions like traffic sniffing using the compromised hosts. The adversary interacts with the IBLT based monitoring primitive solely through network traffic.

Attack objective. The adversary aims to *pollute* the IBLT by injecting crafted flows such that the listing (decode) process stalls, rendering telemetry statistics (*e.g.,* flow identifiers and counters) partially or fully undecodable.

Attacker capabilities. The adversary can (1) observe traffic inbound and outbound from the compromised host as a MitM and (2) inject crafted flows to pollute the IBLT. The adversary cannot directly read, modify, or flip bits inside the device hosting the IBLT. All influence over the IBLT state occurs through packet insertions into the network.

Attacker knowledge. Consistent with Kerckhoffs's principle [54], we assume the adversary knows the IBLT implementation and public parameters but not any cryptographic secrets (*e.g.,* secret seeds for the hash function). Specifically, the adversary is aware of: (1) size of IBLT (2) number of hash functions, (3) the type of hash, and (4) a *partial state* of the IBLT. We

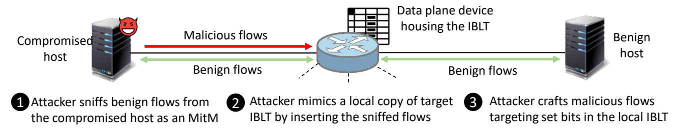


Fig. 3: **Crafting malicious flows that target occupied cells in the IBLT**

m1 m1

KeySum	f1	f2	f1⊕f3	-	f2⊕f3	-
Count	1	1	2	0	2	0

KeySum	f1⊕m1	f2⊕m1	f1⊕f3	-	f2⊕f3	-
Count	2	2	2	0	2	0

Fig. 4: **Impact of malicious flows on pure cells of an IBLT: malicious flow $m1$ collides into $f1$ and $f2$ reduces the pure cell count to 0. No flows can be decoded from the IBLT (listing/decoding is stalled).**

define *partial state* as the attacker's inferred knowledge of which cells are currently occupied in the IBLT (as shown in Fig. 3). The adversary does not know exact cell contents but can estimate occupancy over time by observing traffic.

Attacker strategy. The attacker's goal is to reduce the number of *pure cells* in the IBLT. To achieve this, the adversary crafts flows that hash into cells already occupied by existing flows (Figure 4). By increasing collisions across all hash indices of a flow, the attacker decreases the likelihood that any cell remains pure [26]. Since decoding relies on pure cells to iteratively peel elements, their reduction triggers a cascading stall, preventing complete extraction of stored flows.

B. Practicality of the adversary

Network monitoring primitives operate in short intervals called *epochs* (*e.g.,* $280\,$ms). At the end of each epoch, the IBLT state is exported to the control plane for decoding and analysis, after which it is cleared. We classify an epoch as *affected* (*i.e.,* the attack is successful) if more than 50% of its flows remain undecodable.

Fig. 5 compares two adversaries: (i) a *deployment-aware* attacker with precise knowledge of the IBLT parameters, who crafts flows to collide with occupied cells on all hash indices, and (ii) a *deployment-unaware* attacker, who lacks such knowledge and generates malicious flows at random that may or may not collide with occupied cells. We evaluate these attacks on FlowRadar over 215 epochs using CAIDA traces [43].

Impact on epoch decodability. The deployment-aware attacker naturally affects more epochs by targeting occupied cells. However, both attackers inflict significant damage. Even $2-3\%$ malicious flows render a non-trivial fraction of epochs undecodable.

Practical constraints on targeted attacks. IBLT-based systems such as FlowRadar and LossRadar are open-sourced, and their default parameters are publicly documented. However, in practice, deployments often modify parameters (*e.g.,* table size, number of hash functions) or introduce hash randomization (*e.g.,* secret seeds) [55], limiting an attacker's ability to precisely target specific cells. Moreover, even when such

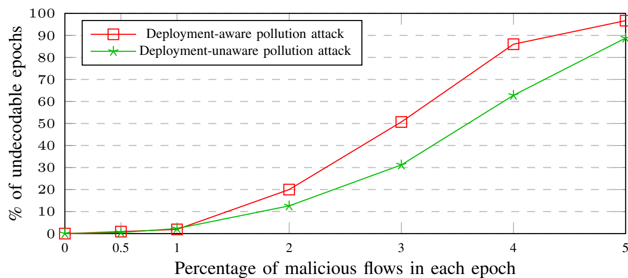


Fig. 5: **Deployment-aware vs Deployment-unaware pollution attack on FlowRadar**

parameters are known, executing a targeted attack within a short epoch (order of ms) is challenging. The adversary must maintain a shadow copy of the IBLT, infer and track cell occupancy by observing benign flows, and generate 5-tuples that hash to specific cells, all within a tight time budget and repeated every epoch. At line rates, even producing a small fraction of constrained malicious flows is computationally demanding. Despite these limitations, the attack surface remains. An attacker need not precisely target cells. He can still increase collisions and reduce the number of pure cells by crafting random flows that may or may not collide on all indices.

Conclusion. *Deployment-aware* attack is often impractical, but not necessary. A *deployment-unaware* attacker can still degrade performance by crafting malicious flows without solving per-flow hash constraints or maintaining a shadow IBLT. Although weaker than precise targeting, Fig. 5 shows that even modest untargeted traffic can disrupt a substantial fraction of epochs motivating the need for lightweight detection and recovery mechanisms. Accordingly, we focus on the *deployment-unaware* adversary in the remainder of the paper.

IV. Requirements for a detection and recovery-based mitigation system

Problem Statement. We address the following question: *'How can we detect and recover statistics lost due to pollution attacks on IBLT-based monitoring primitives?'*. Although pollution attacks stall the listing (decode) process, the flows themselves remain encoded in the IBLT's state. Detecting such stalls and recovering the undecodable flows is therefore beneficial. Moreover, detection can serve as a trigger for additional (maybe costly) mitigation actions, while recovery restores telemetry statistics needed for network decisions.

Design Principle. A practical defense strategy is to first detect anomalies using lightweight mechanisms and then recover quickly, enabling optional reactive measures if necessary. This mirrors common DDoS defense architectures, where inexpensive monitoring detects attacks before costly countermeasures are deployed [56]. Similarly, for IBLT pollution, our goal is to detect decode stalls and recover statistics using lightweight data-plane support, while leaving heavier mitigation strategies to subsequent stages. Based on this objective, we derive the following requirements for a detection-and-recovery system for IBLT-based monitoring primitives:

[R1] Recovery. The system must restore undecodable flows from a polluted IBLT state.

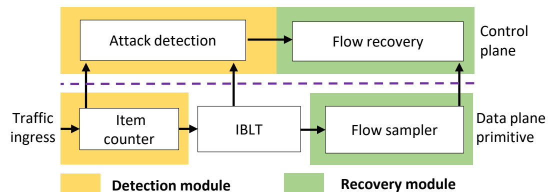


Fig. 6: **ListGuard framework**

[R2] Detection. The system must detect decode stalls due to pollution.

[R3] Generality. Apply broadly to in-network systems that rely on IBLT-based decoding.

[R4] Lightweight. Maintain minimal memory and per-packet overhead and remain implementable on high-speed programmable data planes.

V. ListGuard Design overview

ListGuard is designed to satisfy requirements **[R1–R4]**. Figure 6 illustrates the overall workflow, consisting of a detection module and a recovery module that operate in tandem.

Detection Module. The detection module determines whether an epoch is affected by pollution and triggers recovery when necessary (**[R2]**). To do so, we introduce a lightweight *item counter* in the data plane that tracks the number of distinct flows inserted into the IBLT. After decoding, the control plane compares the number of extracted flows with the counter value to estimate the proportion of undecodable flows. A significant increase in this proportion signals a decode stall.

Recovery Module. Once a decode stall is detected, the recovery module extracts undecodable flows from the polluted IBLT state (**[R1]**). The core idea is to sample *completely colliding flows* (flows that collide on all hash outputs) based on a selection probability p in the flow sampler as shown in Fig. 6. These flows are most likely to reduce pure cells and stall decoding. By subtracting selected sampled flows from the IBLT, new pure cells can emerge, allowing the peeling process to resume. Since the sampled set is a small percentage of the total flows seen, additional memory requirement is minimal satisfying (**[R4]**).

Generality. ListGuard operates alongside the IBLT without modifying its internal structure. This design allows seamless integration with existing IBLT based systems such as FlowRadar [3] and LossRadar [4], satisfying **[R3]**.

VI. Recovery of undecodable flows

As shown in Fig. 6, the data plane flow sampler collects selected flow IDs during each epoch and exports them to the control plane. When the detection module flags an affected epoch, these sampled flows are used to recover undecodable entries from the stalled IBLT.

Resuming a stalled decode. A pollution attack stalls decoding by eliminating *pure cells*. Although decoding halts, many flows may still be recoverable. Our key idea is to reintroduce pure cells by subtracting selected sampled flows from the stalled IBLT state. If this subtraction exposes new pure cells, the peeling process can resume.

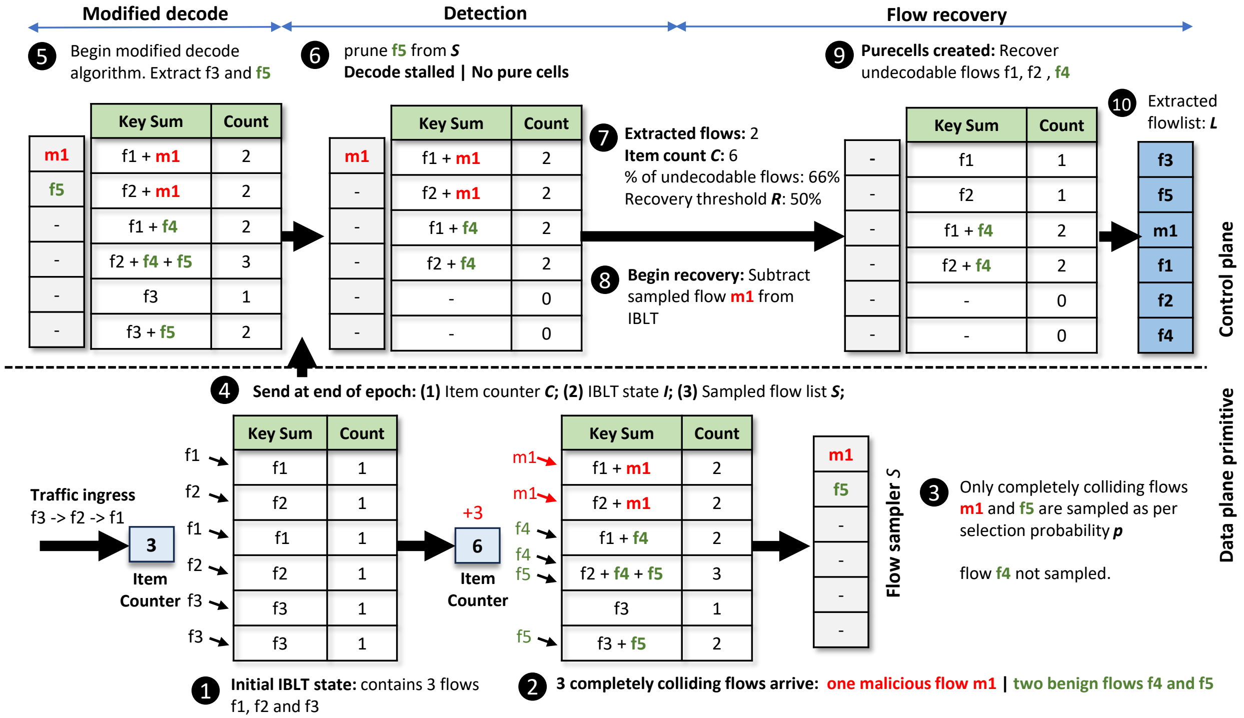


Fig. 7: ***ListGuard* detection and mitigation system:** IBLT with 2 hash functions — ① initial state of the IBLT contains three flows $f1$, $f2$ and $f3$ — ② three new completely colliding flows arrive and increment the item counter: $f4$ and $f5$ are benign, $m1$ is maliciously crafted. All of them are completely colliding flows — ③ based on the chosen selection probability p, the probabilistic sampling technique samples only $m1$ and $f5$ — ④ at the end of the epoch, send the item counter value C, the IBLT state I and the sampled flow list S to the control plane — ⑤ the modified decode algorithm extracts $f3$ and $f5$ — ⑥ prune / remove the extracted flow $f5$ from sampled flow list S. The decoding process is stalled due to the absence of pure cells — ⑦ check if the percentage of undecodable flows is greater than recovery threshold R — ⑧ since the percentage of undecodable flows is greater than R, begin the recovery process by picking $m1$ from the sampled flow list S and subtracting from IBLT — ⑨ use the newly created pure cells to recover flows $f1$, $f2$ and $f3$ — ⑩ all the extracted flows are stored in extracted flow list L

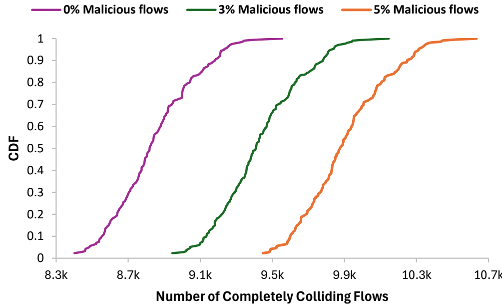


Fig. 8: **CDF of number of CCFs across epochs as the percentage of malicious flows increases**

Sampling strategy. Sampling is critical to recovery. Sampling all flows is impractical due to memory constraints, so we selectively sample *completely colliding flows* (CCFs). These are flows whose hash outputs all map to non-empty cells. CCFs are most responsible for reducing pure cells and are therefore most useful for restarting decoding. Empirically, we observe that the number of CCFs increases under attack as shown in Fig. 8, reinforcing their relevance. Accordingly, *ListGuard* samples only CCFs in the data plane.

Sampling all CCFs would still incur high overhead (typically 33–45% of flows per epoch). We therefore apply uniform probabilistic sampling with selection probability p. A small p risks insufficient recovery and a large p increases memory usage. Empirical evaluation shows that $p = 0.12$ achieves full recovery under our workload (§X). Because probabilistic sampling yields variable sample sizes per epoch, we impose a fixed memory bound in the data plane. Occasional overflows are tolerated in practice, as recovery remains effective with partial samples.

Pruning sampled flow list during decode. Sampled flows are subtracted from the IBLT to uncover new pure cells. However, subtracting flows that have already been decoded would corrupt the state (due to XOR semantics). We therefore modify the decode procedure to prune sampled flows as they are extracted (Algorithm 1). That is, every extracted flow from the IBLT is also removed from the sampled flow list S (⑥ in Fig. 7). After decoding stalls, we compute the fraction of undecodable flows using the item counter. If this exceeds a threshold R (50% in our setting), recovery is initiated.

Recovery procedure. Given a stalled IBLT I and the pruned sampled set S', the recovery algorithm iteratively subtracts flows from S' and invokes the modified decode procedure whenever new pure cells appear (Algorithm 2). This process continues until S' is exhausted. In practice, if p is sufficient, the majority of flows are recovered. The modified decode runs in linear time, while the recovery loop is quadratic in the worst case. Since both execute in the control plane, the computational cost is acceptable.

While our primary focus is on mitigating adversarial pollution, *ListGuard* can also improve the general robustness of

Algorithm 1: ModifiedDecode to prune sampled flows

```
Input: (1) IBLT state I (2) Sampled flow list S (3) Item counter value C
Output: Decoded flowIDs list L
while pure cells remain in the IBLT do
    extract flowID f from IBLT state I;
    if f present in S then
        delete f from S;
    end
    add extracted flowID f to list L;
end
calculate percentage of undecodable flows using L and item counter value C;
if undecodable flow percentage is more than recovery threshold R then
    call Recovery;
end
```

Algorithm 2: Recovery

```
Input: (1) stalled IBLT state I (2) Pruned sampled flow list S'
Output: Decoded flowIDs list L
while S' is not empty do
    choose flowID f from S';
    subtract f from I;
    while pure cells remain do
        call ModifiedDecode with I and S as inputs;
        append the extracted flowIDs in L;
    end
end
```

IBLTs. Even in benign scenarios, traffic bursts or temporary capacity overloads may lead to increased hash collisions, which can reduce the number of pure cells and potentially halt decoding. *ListGuard* can inherently address these non-adversarial decode stalls, supporting more reliable telemetry extraction under varying network conditions.

VII. Detection with ListGuard

The objective of the detection module is to determine whether an epoch is affected and to trigger recovery accordingly. During each epoch, the item counter C records the number of flows inserted into the IBLT. After decoding, the control plane compares the number of extracted flows with C to compute the fraction of undecodable flows (Algorithm 1). A significant increase in this fraction indicates a decode stall and activates the recovery module.

Sampling malicious flows. In addition to enabling recovery, the sampling mechanism provides a reduced set of suspicious flows for further inspection. As described in Section VI, ListGuard probabilistically samples completely colliding flows (CCFs), which are most likely to contribute to decode stalls. Our findings show that even with just 1% of the flows being malicious, we are able to capture malicious flows in over 99% of the epochs with a selection probability of just 0.03 (refer to Fig. 10c for further details). *ListGuard*'s completely colliding flow-based sampling strategy ensures that there is a very high chance of sampling malicious flows in each epoch.

We cannot be certain if the sampled flows contain malicious flows. However, the sampled flows form a compact candidate set that can be analyzed offline or online using existing classification techniques [57]. By reducing the search space from all flows to a small subset, *ListGuard* lowers the computational burden of downstream analysis. For instance, in step ③ of Fig. 7, two out of six flows have been sampled. One of these two sampled flows is malicious. Thus, identifying one malicious flow among two is certainly less complex than among six. Since sampling is already required for recovery, this capability incurs no additional data plane overhead.

Robustness against adaptive adversaries. An attacker aware of *ListGuard*'s sampling strategy might attempt to evade it by crafting partially colliding flows that map to at least one occupied cell. While this can still push the IBLT toward undecodability, it also causes more incoming benign CCFs to be sampled. Despite this, *ListGuard*'s recovery remains effective, as any valid sampled CCF can help restore statistics. We acknowledge that such partially colliding attack flows will avoid appearing in the sampled flow list, bypassing the detection mechanism. Exploring such adaptive adversaries is left to future work.

VIII. Mitigation strategies

We summarize complementary mitigation approaches and assess their suitability for programmable data planes.

Mitigation Strategy 1: Enhanced IBLT constructions provide stronger listing guarantees using matrix-based designs and recursion [23], [31], [32]. While improving decode robustness, they incur significant memory and per-packet overhead, limiting feasibility on resource constrained programmable data planes [1], [2].

Mitigation Strategy 2: Cryptographic authentication of traffic [33], [34] can help limit adversarial insertions, but it may be impractical at high packet rates. While some platforms provide hardware acceleration [2], others lack the instruction support (*e.g.,* loops) required for cryptographic operations at line rate [1]. Furthermore, even when hardware support is available, enforcing per-packet cryptographic authentication for telemetry tracking represents a worst-case workload, potentially introducing nontrivial throughput and latency degradation violating requirement **[R4]**.

Mitigation Strategy 3: Hash randomization using secret salts or pseudorandom functions [55] denies precise targeting by an attacker. However, attackers can still induce collisions and eventual decode stall with modestly increased effort (Fig. 5 and Fig. 10h).

Mitigation Strategy 4: Overprovisioning the IBLT by allocating additional memory increases the effort required to induce decode stalls. However, this approach raises data-plane memory costs and does not guarantee protection against determined attackers. In resource-constrained environments, excess memory allocation is expensive and must be carefully managed. Moreover, even if overprovisioning fails, *ListGuard* can recover the compromised statistics.

IX. Implementation of ListGuard

We implement *ListGuard*'s detection and recovery modules on a Netronome Agilio® CX 1×40GbE SmartNIC [2], along with a software prototype for evaluation. The IBLT, item counter, and flow sampler are implemented as data-plane primitives in Micro-C on the SmartNIC. The SmartNIC is

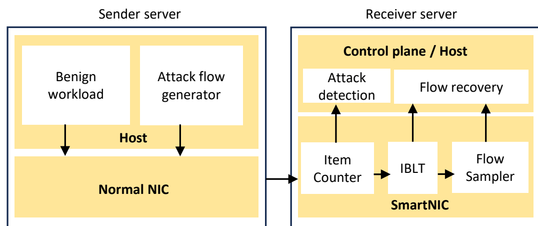


Fig. 9: **Hardware testbed with 2 servers and a SmartNIC**

hosted on a server equipped with 2× Intel Xeon Silver 4316 CPUs (2.30 GHz), 80 cores, and 256 GB RAM. As shown in Figure 9, the SmartNIC receives traffic from a separate sender server via a standard NIC. The sender transmits benign traffic interlaced with malicious flows. Detection and recovery logic execute on the host CPU, analogous to the control plane, while the core control plane logic of *ListGuard* is implemented in Python 3.

X. Evaluation

We evaluate *ListGuard* along five dimensions: (1) recovery capability, (2) malicious flow sampling, (3) memory overhead, (4) latency and throughput impact, and (5) comparison with alternative mitigation strategies.

Experiment methodology. We measure (1) the fraction of affected epochs successfully recovered (**[R1]**) and (2) the fraction of epochs in which malicious flows are sampled (**[R2]**). We then quantify memory overhead (**[R4]**). Next, we note that pollution attacks on the underlying IBLT cause similar information loss in both FlowRadar [3] and LossRadar [4]. Eventhough LossRadar performs a symmetric difference between two IBLTs before decoding, their characteristics under adversarial influence are near identical as shown in Fig. 10a. Therefore, results obtained for FlowRadar extend to LossRadar as well, demonstrating generality (**[R3]**).

Datasets. We use the CAIDA 2018 *dirA March equinix nyc* trace [43]. The trace is partitioned into 280ms epochs (215 total), averaging 23K flows per epoch (8.8K–24K range). IBLT sizes are provisioned based on peak packet/flow counts following the standard guidelines (1.3∗number of expected elements) in [20].

A. Recovering undecodable flows

The selection probability p controls the number of completely colliding flows (CCFs) sampled per epoch and directly impacts recovery performance. Figure 10b shows the relationship between p and the fraction of affected epochs successfully recovered in FlowRadar [3]. As expected, higher p improves recovery, while increasing malicious traffic reduces it. $p = 0.10$ recovers almost all affected epochs for up to 5% malicious flows; $p = 0.12$ consistently achieves full recovery. These settings sample only 0.34%–4% of total flows per epoch (memory overheads discussed in Section X-C).

B. Malicious Flow Sampling

ListGuard detects affected epochs deterministically using the undecodable flow threshold R and triggers recovery accordingly. Beyond recovery, sampled CCFs provide a reduced set of suspicious flows for analysis. Figure 10c shows the fraction of epochs in which malicious flows are sampled as a function of p. A selection probability of 0.06 captures malicious flows in nearly all epochs, except in the 0.5% malicious flow % case. To further examine this low-intensity attack setting, we group consecutive epochs (2–5 epochs per group). As shown in Figure 10d, grouping increases the likelihood of capturing malicious flows, indicating that flows missed in one epoch are likely sampled in subsequent epochs.

C. Memory overheads of ListGuard

The selection probability p determines the number of sampled flows and thus the memory footprint of *ListGuard*.

Mitigation Strategy 4: Table II compares *ListGuard* with naive overprovisioning. We report the additional memory required to prevent any epoch from being affected. Across attack intensities (0.5%–5%), *ListGuard* requires 1.93×–2.83× less memory than naive over provisioning. In practice, *ListGuard* uses roughly half the additional memory while also enabling recovery from polluted states.

TABLE II: Memory overhead (comparison with mitigation strategy 4): Data cells indicate the percentage of extra memory required for complete recovery of all the epochs

	ListGuard with standard IBLT			ListGuard with FlowRadar		
Malicious Flow (%)	**Naive**	**ListGuard**	**savings**	**Naive**	**ListGuard**	**savings**
0.5%	0.99%	0.52%	1.94X	0.99%	0.44%	2.26X
1%	1.99%	1.04%	1.93X	1.99%	0.84%	2.38X
2%	2.99%	1.29%	2.32X	2.99%	1.06%	2.83X
3%	3.99%	1.84%	2.18X	4%	1.53%	2.61X
4%	4.99%	2.18%	2.28X	4.99%	2.29%	2.19X
5%	5.99%	2.81%	2.13X	5.99%	2.56%	2.34X

Mitigation Strategy 1: We also compare against LFFZ [31], which is an IBLT construction that guarantees decoding via custom matrix-based mappings. However, for $\approx$ 24K flows, LFFZ requires 69MB of memory, approximately 230× more than *ListGuard* which requires just 0.3MB as shown in Fig. 10g rendering such approaches infeasible for data planes.

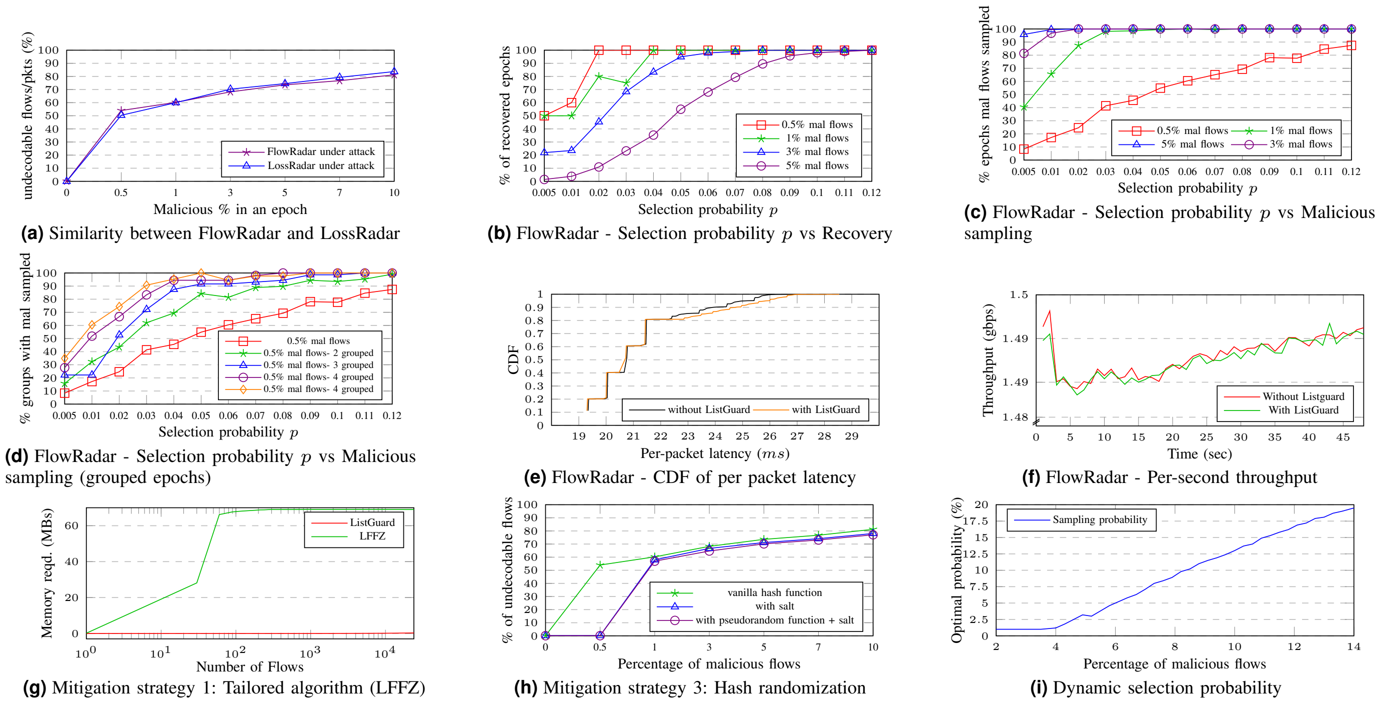
Mitigation Strategy 3: Hash randomization removes targeting precision but does not eliminate pollution. Even with salts, modest malicious traffic ($\geq$ 1%) still degrades decoding (Fig. 10h), necessitating recovery.

D. Per-packet latency

We measure per-packet latency with and without *ListGuard* on the SmartNIC. Tail latency increases by 1.64%, while average latency rises by 0.52% (Fig. 10e). The overhead stems primarily from the flow sampler, which activates only for completely colliding flows (34–44% of flows). This selective activation explains the minimal latency impact.

E. Throughput

To measure the throughput of *ListGuard*, we use TCPReplay [58] to replay CAIDA traces from the sender server in Fig. 9 to the SmartNIC in the receiver server. *ListGuard* reduces throughput by only 0.026% (Fig. 10f), indicating negligible performance impact.



(a) Similarity between FlowRadar and LossRadar

(b) FlowRadar - Selection probability p vs Recovery

(c) FlowRadar - Selection probability p vs Malicious sampling

(d) FlowRadar - Selection probability p vs Malicious sampling (grouped epochs)

(e) FlowRadar - CDF of per packet latency

(f) FlowRadar - Per-second throughput

(g) Mitigation strategy 1: Tailored algorithm (LFFZ)

(h) Mitigation strategy 3: Hash randomization

(i) Dynamic selection probability

Fig. 10: ***ListGuard* evaluation**

F. Dynamic sampling probability

So far, our design assumes a fixed sampling probability p across epochs. While effective, varying traffic and attack intensities change the collision pressure on the IBLT, suggesting that p can be adapted to input conditions. As a preliminary extension, we evaluate a dynamic strategy that adjusts p every 10ms using traffic and IBLT health indicators, including *PureCellRatio* (decodability) and *TotalFlows* (flow intensity). These features are fed into a lightweight Random Forest regressor to estimate the sampling probability needed to maintain decodability.

We observe an approximately linear relationship between attack intensity and the optimal p (where all affected epochs are decodable) in Fig. 10i. In simulations with varying attack rates, the adaptive mechanism predicts an optimal p with 89.12% accuracy, improving resilience while limiting overhead. This complements our static design. However, a deeper study of adaptive policies and hardware feasibility is left for future work.

XI. Related Work

Securing dataplane systems. [17], [59]–[61] emphasize the importance of safeguarding data plane systems against adversarial threats. Our contributions support the fundamental notion presented in these studies, which underscore the urgent requirement to defend such systems.

Detection and defense framework for sketches. [39], [62] present frameworks aimed at securing sketches. TrustSketch [39] leverages secure enclaves to detect malicious modifications when the hosting device itself is compromised. Our work complements these efforts by considering a different threat model where external adversaries interact with the sketch solely through network traffic, without compromising the device that hosts the sketch.

XII. Conclusion and Future Work

In this work, we design and develop *ListGuard* a system for detecting and recovering from attacks targeting in-network IBLT based monitoring primitives. By conducting experiments with realistic traces, we demonstrate the efficacy of *ListGuard* in recovering undecodable statistics from a compromised IBLT. In future work, we plan to develop lightweight techniques to more precisely identify malicious flows within the sampled set under diverse attack workloads.

Acknowledgement

This work is supported by the National Security Council Secretariat (NSCS), India, and the Prime Minister's Research Fellowship (PMRF).

References

[1] A. Agrawal and C. Kim, "Intel tofino2–a 12.9 tbps p4-programmable ethernet switch," in *IEEE Hot Chips, 2020*.

[2] "Netronome agilio," https://netronome.com/agilio-smartnics/.

[3] Y. Li, R. Miao, C. Kim, and M. Yu, "{FlowRadar}: A better {NetFlow} for data centers," in *USENIX NSDI, 2016*.

[4] ——, "Lossradar: Fast detection of lost packets in data center networks," in *ACM CoNEXT, 2016*.

[5] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, "Fine-grained queue measurement in the data plane," in *ACM CoNEXT, 2019*.

[6] W. Wang, P. Tammana, A. Chen, and T. E. Ng, "Grasp the root causes in the data plane: Diagnosing latency problems with spidermon," in *ACM SOSR, 2020*.

[7] S. Sengupta, H. Kim, and J. Rexford, "Continuous in-network round-trip time monitoring," in *ACM SIGCOMM, 2022*.

[8] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *ACM SOSR, 2016*.

[9] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *ACM SIGCOMM, 2017*.

[10] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *ACM SOSP, 2017*.

[11] M. Apostolaki, A. Singla, and L. Vanbever, "Performance-driven internet path selection," in *ACM SOSR, 2021*.

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

[13] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Contra: A programmable system for performance-aware routing," in *USENIX NSDI, 2020*.

[14] J. Xing, Q. Kang, and A. Chen, "{NetWarden}: Mitigating network covert channels while preserving performance," in *USENIX Security, 2020*.

[15] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches," in *USENIX Security, 2021)*.

[16] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo, "Programmable {In-Network} security for context-aware {BYOD} policies," in *USENIX Security, 2020*.

[17] A. Sanghi, K. P. Kadiyala, P. Tammana, and S. Joshi, "Anomaly detection in data plane systems using packet execution paths," in *ACM SIGCOMM workshop SPIN, 2021*.

[18] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, 1970.

[19] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, 2005.

[20] M. T. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," in *IEEE Allerton Conference on Communication, Control, and Computing, 2011*.

[21] M. Mitzenmacher and G. Varghese, "Biff (bloom filter) codes: Fast error correction for large data sets," in *IEEE ISIT, 2012*.

[22] M. Mitzenmacher and R. Pagh, "Simple multi-party set reconciliation," *Distributed Computing*, vol. 31, no. 6, 2018.

[23] A. P. Ozisik, G. Andresen, B. N. Levine, D. Tapp, G. Bissias, and S. Katkuri, "Graphene: Efficient interactive set reconciliation applied to blockchain propagation," in *ACM SIGCOMM, 2019*.

[24] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? efficient set reconciliation without prior context," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, 2011.

[25] R. J. Tobin and D. Malone, "Hash pile ups: Using collisions to identify unknown hash functions," in *IEEE CRiSIS , 2012*.

[26] T. Gerbet, A. Kumar, and C. Lauradoux, "The power of evil choices in bloom filters," in *IEEE/IFIP DSN, 2015*.

[27] P. Reviriego and O. Rottenstreich, "Pollution attacks on counting bloom filters for black box adversaries," in *IEEE CNSM, 2020*.

[28] N. Duffield, C. Lund, and M. Thorup, "Estimating flow distributions from sampled flow statistics," in *ACM SIGCOMM, 2003*.

[29] C. Estan, K. Keys, D. Moore, and G. Varghese, "Building a better netflow," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, 2004.

[30] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang, "Is sampled data sufficient for anomaly detection?" in *ACM IMC, 2006*.

[31] A. Mizrahi, D. Bar-Lev, E. Yaakobi, and O. Rottenstreich, "Invertible bloom lookup tables with listing guarantees," *POMACS*, vol. 7, no. 3, 2023.

[32] N. Fleischhacker, K. G. Larsen, M. Obremski, and M. Simkin, "Invertible bloom lookup tables with less memory and randomness," *arXiv preprint arXiv:2306.07583*, 2023.

[33] S. Yoo and X. Chen, "Secure keyed hashing on programmable switches," in *ACM SIGCOMM Workshop SPIN, 2021*.

[34] I. Oliveira, E. Neto, R. Immich, R. Fontes, A. Neto, F. Rodriguez, and C. E. Rothenberg, "Dh-aes-p4: on-premise encryption and in-band key-exchange in p4 fully programmable data planes," in *IEEE NFV-SDN, 2021*.

[35] NIST and Q. Dang, "Secure hash standard," 2015-08-04 00:08:00 2015. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=919060

[36] N. Sklavos, "Book review: Stallings, w. cryptography and network security: Principles and practice," *Information Security Journal: A Global Perspective*, vol. 23, no. 1-2, 2014.

[37] I. Chenchev, A. Aleksieva-Petrova, and M. Petrov, "Authentication mechanisms and classification: a literature survey," in *Computing Conference (IC) 2021, vol. 3*. Springer.

[38] P. Reviriego, O. Rottenstreich, S. Liu, and F. Lombardi, "Analyzing and assessing pollution attacks on bloom filters: Some filters are more vulnerable than others," in *IEEE CNSM, 2021*.

[39] Z. Cheng, M. Apostolaki, Z. Liu, and V. Sekar, "Trustsketch: Trustworthy sketch-based telemetry on cloud hosts," in *NDSS, 2024*.

[40] P. Reviriego, M. González, N. Dayan, G. Huecas, S. Liu, and F. Lombardi, "On the security of quotient filters: Attacks and potential countermeasures," *IEEE Transactions on Computers*, vol. 73, no. 9, 2024.

[41] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *European Symposium on algorithms*, 2006.

[42] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, "Don't thrash: how to cache your hash on flash," *Proc. VLDB Endow.*, vol. 5, no. 11, Jul. 2012.

[43] CAIDA Datasets. [Online]. Available: https://www.caida.org/catalog/datasets/passive_dataset/

[44] Y. Hu, D.-M. Chiu, and J. C. Lui, "Entropy based adaptive flow aggregation," *IEEE/ACM Transactions on Networking (TON)*, vol. 17, no. 3, 2009.

[45] E. Assaf, R. B. Basat, G. Einziger, and R. Friedman, "Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free," in *IEEE INFOCOM, 2018*.

[46] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *ACM SOSR, 2017*.

[47] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *ACM CoNEXT, 2011*.

[48] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *ACM SIGCOMM, 2016*.

[49] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani, "Fast monitoring of traffic subpopulations," in *ACM IMC, 2008*.

[50] H. Liu, X. Chen, Y. Shen, Q. Huang, Z. Zhou, D. Zhang, and C. Wu, "Vulnerabilities and attacks of inter-device coordination in programmable networks," in *IEEE/ACM IWQoS, 2023*.

[51] B. Lee, "Vmware esxi security patch," https://www.virtualizationhowto.com/2018/11/vmware-esxi-successful-vm-escape-at-geekpwn2018-security-patch/.

[52] "Salt stack vulnerability," https://www.datacenterknowledge.com/security-and-risk-management/hackers-exploiting-saltstack-vulnerability-hit-data-centers.

[53] L. Constantin, "Hypervisor privilege escalation," https://thenewstack.io/privilege-escalation-information-leak-flaws-patched-xen-hypervisor/.

[54] F. A. Petitcolas, "Kerckhoffs' principle," in *Encyclopedia of Cryptography, Security and Privacy*, 2025.

[55] E. Boyle, S. Goldwasser, and I. Ivan, "Functional signatures and pseudorandom functions," in *International workshop on public key cryptography*, 2014.

[56] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, 2004.

[57] M. Ahmed, A. N. Mahmood, and J. Hu, "A survey of network anomaly detection techniques," *Journal of network and computer applications*, vol. 60, 2016.

[58] S.-S. Hong and S. F. Wu, "On interactive internet traffic replay," in *RAID*, 2005.

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

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

[61] C. Black and S. Scott-Hayward, "Adversarial exploitation of p4 data planes," in *IFIP/IEEE IM, 2021*.

[62] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "{SafeBricks}: shielding network functions in the cloud," in *USENIX NSDI, 2018*.