TSN-Guard: Monitoring Packet Forwarding Misbehavior of Time-Sensitive Flows in TSN

Chuwen Zhang, Boyang Zhou, Liang Cheng, Yuxi Liu, Tian Pan, Ying Wan, Wenquan Xu, Yi Wang, Shuzhen Tian, Ji Miao, Yang Xu, Bin Liu

Abstract

Time-Sensitive Networking (TSN) is proposed in recent years to satisfy strict transmission performance for time-sensitive flows. To achieve this goal, several packet scheduling and shaping algorithms have been standardized in TSN, but they need to be configured carefully. Although theoretical models are refined to produce a sophisticated configuration, in real implementations, the expected performance cannot always be provided. Packet forwarding violations may happen due to inherent mechanism defects and time synchronization errors. This kind of misbehavior will potentially cause unpredictable consequences when TSN is applied to the industrial control. In this paper, we design TSN-Guard, a real-time TSN performance monitoring system, including its architecture design and data collection mechanism, to timely report packet forwarding misbehavior to the TSN controller for violation mitigation. TSN-Guard offloads the misbehavior discovery on data plane to relieve the burden on the controller and network links. To reduce the interruption to the controller, it uses In-band Network Telemetry (INT) probe packets to collect misbehavior in aggregation. We conduct extensive experiments using the OMNET++ simulator and analyze the performance of probe planning algorithms. The results exhibit that TSN-Guard is light-weight and offers fast-response and full-coverage properties.

I. Introduction

With the development of the Industrial Internet of Things (IIOT) and Cyber-Physical Systems (CPS), Time-Sensitive Networking (TSN) has been proposed to provide guaranteed delay performance for time-sensitive flows [1], [2]. The principle of TSN operations is to transmit both the time-sensitive traffic and time non-sensitive traffic through the Ethernet. Among them, Time-Triggered (TT) traffic, profiled by fixed size and period information, usually carries time-sensitive and critical data, such as real-time industrial control instructions, and should be prioritized to guarantee its delay performance.

Given predictable features of TT traffic, several synchronous scheduling and shaping mechanisms have been proposed and standardized, such as Time-Aware Shaping (TAS) [3] and Cyclic Queuing and Forwarding (CQF) [4]. Their core idea is to set fixed and appropriate scheduled time windows for TT flows along their forwarding paths. Therefore, such window planning affects the Quality of Service (QoS) of TT flows.

Theoretical models [5]–[8] have put forward some seemingly wonderful solutions for the window-planning problem in theory, but the QoS performance of TT traffic is hard to be guaranteed in the engineering practice. The fundamental reasons come from two perspectives. First, these theoretical models do not lead to exact feasible solutions as both TAS and CQF have inherent mechanism defects as illustrated in Section II. Second, some key parameters in the models, such as the time synchronization error, are time-varying random variables whose distribution cannot be predicted precisely in advance. Therefore, theoretical models striking to achieve an optimal balance between the delay performance guarantees and TSN utilization are prone to yield window-planning results sensitive to the parameter values.

Since the actual TSN network cannot always guarantee the performance as planned by the theoretical models, misbehavior may happen when flows are not forwarded at their specific time instants expected by the theoretical window-planning result. Therefore, it is necessary to provide a real-time monitoring system of TSN to report misbehavior to the controller when they happen, with a goal of reregulating system parameters to mitigate or avoid misbehavior in the future. Unfortunately, until now, there has been lacking such an effective monitoring system.

In this paper, we present TSN-Guard, a light-weight, fast-response, and full-coverage system for discovering, recording, and collecting the packet forwarding misbehavior in TSN. By a dedicated misbehavior identifier placed in the data plane, packet-forwarding misbehavior events can be captured in time. When the controller receives a misbehavior notification, it will launch In-band Network Telemetry (INT) [9] probe packets to collect the misbehavior data along the planned paths in a period of time. TSN-Guard adopts optimized probe paths and sending moments to improve the system performance further. The results on the OMNET++ simulator [10] show that TSN-Guard outperforms other schemes in terms of bandwidth cost, response time, and frequency of interrupting the controller, e.g. saving 95% bandwidth and 27% response time than the scheme that notifies the controller every misbehavior event.

The contributions in our research include: 1) we analyze root causes of packet forwarding misbehavior in TSN and design TSN-Guard, including the identifier and INT-based collection mechanism, to monitor misbehavior efficiently; 2) we propose Traveling Salesman Problem (TSP) based algorithms to derive satisfying probe path plans, and a Dynamic Programming (DP) based algorithm to arrange probe sending moments; 3) we realize TSN-Guard on the OMNET++ simulator and conduct extensive experiments to verify its performance.

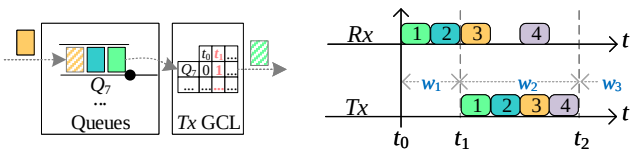


Fig. 1: TAS structure overview and packet forwarding example

In the remainder of this paper, Sec. II reveals the root causes of misbehavior and justifies the necessity of a shadow monitoring system, Sec. III describes the architecture and principles of TSN-Guard, Sec. IV details the algorithms for probe planning, Sec. V evaluates the performance, Sec. VI surveys the related work, and Sec. VII concludes our work.

II. Misbehavior in TSN Traffic Scheduling

A. Synchronous Scheduling Mechanisms

With the network-wide synchronous clock, packet scheduling mechanisms such as TAS and CQF have been developed to ensure the deterministic forwarding for TT flows.

TAS is an important synchronous scheduling mechanism for allocating bandwidth resources to flows with different priorities. TAS allocates a separate packet queue for each priority, where at least one queue is exclusive for TT traffic. Ahead of each queue, TAS sets a gate to determine whether the queue's head packet can be dequeued for processing. The open and closed states of the gates are controlled by the Gate Control Lists (GCLs) so that packets with different priorities are transmitted in different time windows, which have variable lengths and do not overlap. As shown in Fig. 1, Q_7 is for TT traffic and is opened within time t_1 to t_2, so the four packets of TT traffic are transmitted in the window w_2. To meet latency and jitter requirements of TT traffic, network operators need to derive suitable GCLs, which is hard in large-scale networks.

To simplify the design of GCL, CQF uses fixed-size windows and forwards packets in a ping-pong manner: any packet received in the window w_i must be sent in the window w_{i+1} and received by its next-hop switch in w_{i+1}. As a result, the delay for each hop is deterministic, and the end-to-end latency can be calculated by summing up per-hop delay, if the windows are planned properly. As shown in Fig. 2, CQF can be implemented based on TAS by deploying an Rx GCL and setting two queues for TT traffic, as Q_7 and Q_6. To ensure packets being queued in the ping-pong manner, the states of Q_7 and Q_6 in any window must be the opposite.

B. Imperfect Theoretical Models

Both TAS and CQF rely on a feasible window-planning, including GCL and each flow's sending time, to satisfy the requirements of TT traffic. Corresponding theoretical work is based on an optimization model with constraints abstracted from features of TSN networks and requirements of TT flows, such as [5]–[7] for TAS and [8] for CQF. However, due to the inherent mechanism defects of TAS or CQF, and the time synchronization error in wired [11] and wireless TSN networks [12], flows may not be forwarded as the theoretical model expects, which we define as misbehavior, resulting in violations of end-to-end latency and/or jitter potentially.

1) Inherent Mechanism Defect of TAS: Since the queues for TT traffic are FIFO (First-In-First-Out), the packet order can only be guaranteed for packets coming from the same source. If two packets come from different sources, the order can be arbitrary because of the time synchronization error. As shown in Fig. 3, if packet p_3 arrives at the router earlier than packet p_2, p_3 is possible to be enqueued before p_2 and forwarded in the window w_2 instead of its expected window w_4, where w_2 and w_4 are the open windows for TT flows. Thus p_2 is delayed to be transmitted in w_4 and may violate the end-to-end latency requirement.

Existing theoretical models (e.g., [5], [6]) isolate packets with enough space to guarantee that each packet is transmitted in its expected window. However, they cannot guarantee no misbehavior happening and may lead to sub-optimal resource utilization in TSN. First, the models assume that the hosts can send packet exactly as scheduled, which is difficult as the hosts would need per-flow queues rather than per-priority queues used in current TSN. Second, the stringent constraints on isolating packets reduce the TSN design space, sometimes leading to no design solution even though there exists a feasible design. Third, it is difficult to set proper parameter values to realize the best trade-off without instrumenting the network. For example, if the model takes a large value of the time synchronization error into consideration, it is likely to result in no solution or adopting a large window which reduces the bandwidth utilization. On the contrary, using smaller values leads to a higher probability of misbehavior occurrence.

2) Inherent Mechanism Defect of CQF: CQF has two inherent mechanism defects. The first is the misbehavior caused by the time synchronization error. As shown in Fig. 4, the clock of router R2 is faster than that of router R1 due to the time synchronization error, and R1 forwards a packet to R2 in the window w_{i-1}. The packet arrives at R2 in the window w_{i-1} of R1 but in the window w_i of R2. As a result, R2 will transmit this packet in the window w_{i+1} of R2, violating the ping-pong manner and increasing the end-to-end latency.

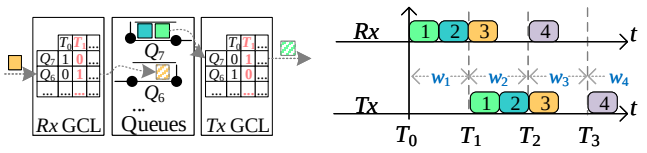
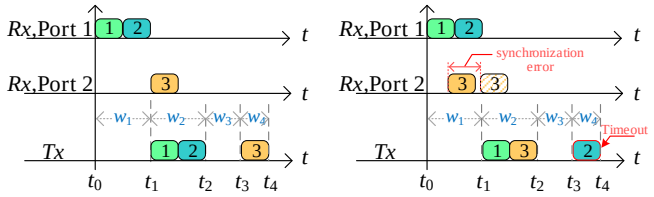


Fig. 2: CQF structure overview and packet forwarding example



(a) Normal Forwarding (b) Misbehaved Forwarding

Fig. 3: TAS misbehavior example.

Another defect is that the ping-pong manner requires that each window must be able to transmit an entire TT queue, so the window size may be large. Without considering the time synchronization error, for a TT flow with n-hop, its latency is no more than n windows and jitter is 1 window, both of which may be too large for its requirements and cause the flow not being scheduled. The experiment in [8] proves this conclusion, e.g., only 1200 flows out of 2000 ones can be scheduled.

The most recent theoretical model for CQF is the Injection Time Planning (ITP) [8]. It derives a feasible window-planning for CQF, i.e., the sending offsets of TT flows and the window size, under constraints like the ping-pong manner. At the model level, ITP does not provide constraints on the flow jitters, i.e., the jitter can reach up to a window size, which may lead to unmet jitter requirements. Even if the jitter requirement is considered in the future model, ITP cannot tackle the above-mentioned mechanism defects properly, as a feasible window-planning must be testified by the real network.

Current theoretical work is only for TT flows, which have fixed periods and sizes. Particularly, each generated window in TAS can only accommodate the exact number of packets scheduled to it. However, in real networks, TT flows may present fluctuations in their periods and sizes, leading to misbehavior against the static and rigid window-planning.

In summary, the theoretical methods mentioned above fail to guarantee no misbehavior unless the sending time of packets can be precisely controlled, the parameter values are fixed, and TT flows are periodical with fixed sizes. The fundamental problem is that the static window-planning does not take the real network conditions into account. Therefore, we need a TSN monitoring system to collect misbehavior information, which may be used by a rescheduling algorithm to adjust the window-planning in run time. In this paper, we focus on the high-performance TSN monitoring system and solve challenges in designing and implementing it.

III. TSN-Guard System Design

A. System Requirements

We raise three requirements for the TSN monitoring system.

1) Full Coverage: As the example in Fig. 3(b), one TT flow's timeout may be caused by the misbehavior of other TT flows. Therefore, to find out and remove the crux of the current window-planning, the system should be aware of the per-hop forwarding status of all flows, i.e., realizing a full coverage.

2) Light Weight: The system should reduce the following three types of overhead. First, network devices need to process and/or record the per-flow per-hop forwarding status, which brings extra computing and storage overhead. Second, collecting such forwarding status consumes bandwidth resources. If collecting forwarding status occupies parts of the TT windows, it may make normal TT flows misbehaved. Third, the controller needs to handle misbehavior reports and collection packets. Therefore, the monitoring cost should be well controlled.

3) Fast Response: The time from when a forwarding misbehavior happens to when it is received by the controller determines whether the system can realize fast fault recovery and reduce the potential economic losses, so the system needs to collect the misbehavior information as fast as possible.

B. System Overview

To meet the above three requirements, we propose TSN-Guard to answer the following questions: what to monitor, how to discover it, and how to collect its information.

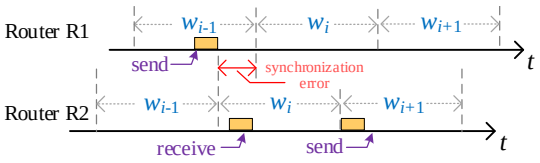


Fig. 4: CQF misbehavior example

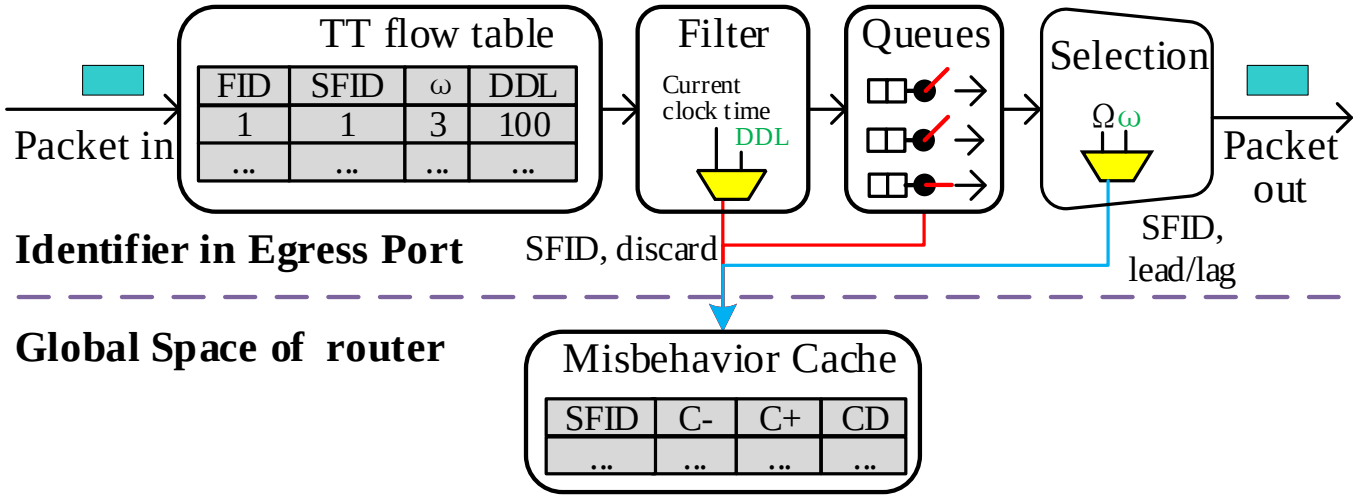


Fig. 5: The structure of misbehavior identifier and cache.

1) Misbehavior Profile: The most accurate forwarding status is the arrival and departure timestamps of each TT packet, but it is too expensive for storage and transmission. TSN-Guard offloads the misbehavior discovery process from the controller to the dataplane, so a router only needs to export misbehaviors to the controller. Further, we compress the misbehavior information by keeping the misbehavior profile only, resulting in low transmission overhead.

Note that the window-planning fundamentally defines a packet-to-window mapping. Let f_i denote a TT flow[1] and $p^l_{i,j}$ denote a packet of f_i waiting for transmission on the l-th port of router j. The real and expected transmission window of $p^l_{i,j}$, are denoted by $\Omega^l_{i,j}$ and $\omega^l_{i,j}$, respectively. There are four cases between them: i) the packet $p^l_{i,j}$ is forwarded normally, i.e., $\Omega^l_{i,j} = \omega^l_{i,j}$; ii) the packet is forwarded ahead of its expected window, i.e., $\Omega^l_{i,j} < \omega^l_{i,j}$; iii) the packet is forwarded after its expected window, i.e., $\Omega^l_{i,j} > \omega^l_{i,j}$; and iv) the packet is dropped, i.e., $\Omega^l_{i,j} = \emptyset$. The latter three cases all lead to a misbehavior event. Therefore, each router can maintain three counters of the lead, lag, and discard packets for each flow with much lower overhead compared with using accurate timestamps. All counters along a flow's path depict its misbehavior profile, and help the controller to find the root cause. For example, as shown in Fig. 4, if the controller figures out that one packet of the timeout flow is forwarded normally by Router A but forwarded in a delayed window by Router B, it may increase the corresponding window by this packet's size or let the timeout flow start sending one window earlier.

2) Misbehavior Discovery: To discover the type of misbehavior, we design a misbehavior identifier as shown in Fig. 5.

First, when a packet $p^l_{i,j}$ arrives at the egress port connected to link l, the identifier first gets its expected window $\omega^l_{i,j}$, deadline and Short Flow ID (SFID) by a TT flow table which is configured by the controller. SFID is a short id to distinguish TT flows only. This flow table can be implemented by the match-action table in P4 [13] easily. Particularly, if the window-planning is not changed at the beginning of a new hyper period, the router can just update ω of all table entries by adding the total number of windows in the hyper period.

The packet with these three parameters then goes to a filter. Typically, the filter is based on the Per-flow Filtering and Policing (PSFP) [14], which can discard packets if a packet's arrival time is later than its deadline. Next, the packet will wait for transmission in its corresponding queue, where it may be discarded by the queue management algorithms, e.g., DropTail or Random Early Detection (RED). When the filter or the queue discards a TT packet, the identifier will get a misbehavior of the SFID and the discard type.

At last, when the TT packet $p^l_{i,j}$ is selected for transmission, the identifier will compare its expected piggyback window $\omega^l_{i,j}$ with the current window Ω which increases by one whenever the pointer of the GCL moves. If they are not equal, the identifer will get a misbehavior of the SFID and a lead or lag type based on the comparison result.

3) Collection Mechanism: To export all misbehavior data discovered by the identifiers to the controller, a collection mechanism is needed. Event-triggered mechanism [15] is optional, which enables routers to notify the controller whenever a misbehavior event occurs. This mechanism achieves the shortest response time and saves bandwidth resources when misbehavior

[1]In this paper, each repeated TT flow in a hyper period (the least common multiple of all TT flows' periods) is regarded as a new flow.

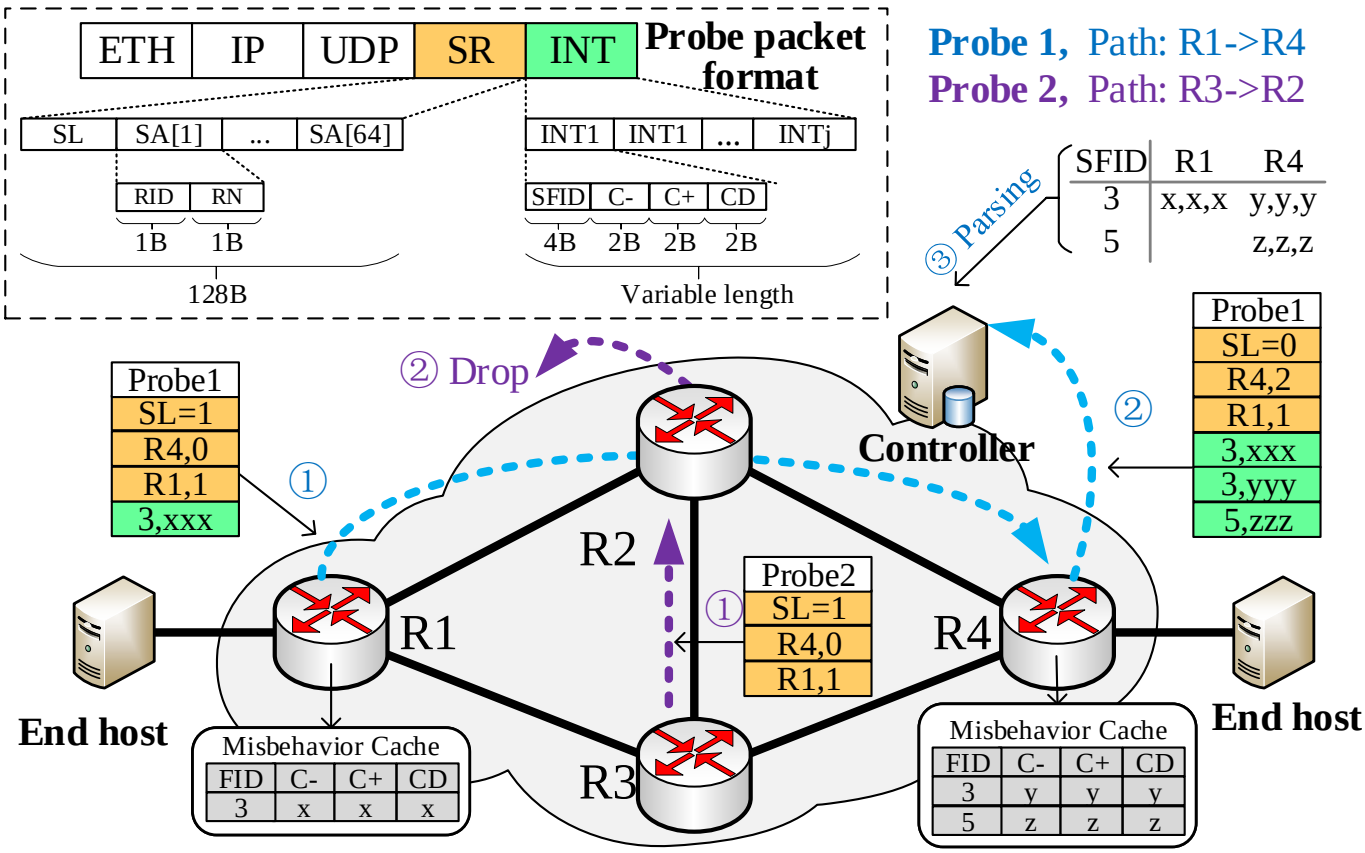


Fig. 6: INT probe packet format and TSN-Guard overview.

events occur rare in the TSN networks. However, a misbehavior event is likely to cause a series of subsequent packets misbehaved, resulting in much bandwidth resource cost and even a burst to the controller.

To avoid misbehavior exporting burst to the controller, each router first stores the misbehavior data in a local cache. As shown in Fig. 5, the cache is a storage in the router's global space, recording misbehavior data from each egress port. Each entry consists of four fields: SFID and three counters C^-, C^+ and C^D, corresponding to the number of lead, lag, and discard packets, respectively. When receiving a SFID and a misbehavior type, the cache increases the corresponding counter by one, or inserts a new entry if the SFID is new. After exporting, the cache will remove all entries to avoid reporting duplicated misbehavior data to the controller.

Second, the controller needs to read all caches by an efficient collection mechanism. A straightforward way is to let the controller ask each router of the cached misbehavior data or each router exports that to the controller periodically, like SNMP [16], NetFlow [17], [18]. The problem is that the controller needs to tackle as many connections as the number of routers, not quite scalable as the network expands. To reduce the number of connections, we resort to INT probe packets like [19] to read caches along their paths, and the fixed paths are realized on the Segment Routing (SR) [20].

Packet format. As shown in Fig. 6, we use UDP packets to carry the misbehavior data. We set a specific destination port `INT_PROBE` in the UDP header to tell the packet parser that it is a INT probe packet. The INT payload consists of multiple records, each of which occupies ten bytes: four-byte SFID and the tree two-byte counters. The SR header consists of Segment Left (SL), which stores the number of remaining segments, and Segment Array (SA), which stores the predefined path of the probe packet. Each element in SA is composed of two fields, one-byte Router ID (RID) to store the segment destination and one-byte Record Number (RN) to store the number of misbehavior flows read from the router. SA stores RIDs in a reverse order of the probe path, so SL can be used as the index to the next RID. To reduce the complexity of the parser, we set the SR header to be fixed at 128 bytes, i.e., supporting up to 64 routers, which is large enough for general networks.

Probing process. We use the architecture in Fig. 6 to explain how probes are generated, forwarded, and analyzed.

In TSN-Guard, one or several routers are configured as the INT probe generators, which can generate 'empty' INT probe packets with a predefined SR header and the `INT_PROBE` UDP destination port. As shown in Fig. 6, Router R1 generates Probe1 with a path {R1, R4}, and R2 generates Probe2 with a path {R3, R2}. For simplicity, probe packets can be generated periodically, but this mode is luxurious when misbehavior events rarely happen. Hence, we propose an event-triggered probe generation mode as follows. When a new misbehavior is identified, the router will notify the controller immediately and then keep a silent state for a hyper period, during which it does not actively send any misbehavior notification. Whenever the controller received a misbehavior notification, it will tell probe generators to start probing for a hyper period. Note that any

INT probe packet will make the routers into the silent state, reducing the number of misbehavior notifications to the controller. Nevertheless, both periodical and event-triggered generation modes need orchestrated policies of the probe path planning and sending moments to realize low-overhead and real-time monitoring, which will be detailed in Section IV.

When going through a router, a probe packet triggers the probe process only if its destination IP equals to the router's IP address. Otherwise, it will be forwarded as an ordinary L3 packet. The probe process consists of three steps: i) appending the cached misbehavior data to the tail of the packet payload and setting the SA[SL].RN by the number of the cached flows; ii) decreasing SL by 1 and replacing the destination IP address in the IP header with the IP address of SA[SL].RID; and iii) forwarding this probe packet. If SL equals to 0 in the step ii, it means the probe packet has reached the end of its path. In this case, the packet will be forwarded to the controller only if its payload is not empty, as Probe1 with the misbehavior data of R1 and R4 in Fig. 6. A probe packet without misbehavior data will never be sent to the controller as Probe2 in Fig. 6, mitigating interruption to the controller. At last, the controller analyzes the received probe packets to get the per-hop per-flow misbehavior data. Because SA and INT payload grow in opposite directions, the controller parses the INT probe packet from the tail of SA and the head of payload to both ends.

IV. INT-probe Polices

The performance of TSN-Guard relies on INT probe path planning policy and probe sending moment policy. In this section, algorithms of these policies are explained in detail.

A. Probe Path Planning

First, we need to clarify the terms *walk*, *trail*, and *path* in graph theory. For a simple graph, a walk can be represented by a vertex sequence in which each pair of adjacent vertices must be connected by an edge, a trail is a specific walk without repeated edges, and a path is a specific trail without repeated vertices. For the instance in Fig. 7, $\{v_2, v_3, v_4, v_2, v_4\}$ is a walk, $\{v_2, v_3, v_4, v_2\}$ is a trail, and $\{v_2, v_3, v_4\}$ is a path. Although the forwarding task of an INT probe is a walk from the perspective of routers, it is a path from the perspective of segments in SR, so we use the term of path planning.

We list four parameters to evaluate the performance of a probe path planning: i) the number of paths or path number, reflecting the interruption to the controller; ii) the total number of hops or total hops, reflecting the total bandwidth occupation of all probes; iii) maximum hop, reflecting the maximum transmission delay of probes; and iv) execution time, the time cost of getting a new probe path planning.

Pan et al. in [19] propose a path planning policy based on the Euler trail for edge coverage. However, the misbehavior profile is stored in routers, transforming the path planning to a vertex coverage problem. Here we propose two algorithms for probe path planning without and with the hop constraint.

1) No hop constraint: In this scenario, the hops of the probe path can be infinite. To design such a path planning algorithm, we propose the following lemma and theorem first.

Lemma 1. For any simple connected graph $G(V, E)$, if $\boldsymbol{w}$ is the shortest walk covering V, it can be partitioned into a sequence of $n-1$ shortest paths, where $n = |V|$.

Proof. First, let $\boldsymbol{z}$ store the unique vertex $\boldsymbol{w}$ in order and $h(z_i)$ records the corresponding index, i.e., $z_i = w_{h(z_i)}$. Obviously, we have $z_1 = w_1$ and $z_n = w_{|\boldsymbol{w}|}$. Then, we use $\boldsymbol{z}$ to partition $\boldsymbol{w}$ into $n-1$ paths, where the i-th path $\boldsymbol{p}_i = \{w_{h(z_i)}, w_{h(z_i)+1}, \ldots, w_{h(z_{i+1})}\}$. Finally, we prove the that any path $\boldsymbol{p}_i$ is the shortest path between z_i and z_{i+1} by contradiction. Assume there exists a path $\boldsymbol{p}_i$ which is not the shortest path between z_i and z_{i+1}, we can use the vertices on the shortest path between z_i and z_{i+1} to replace the corresponding vertices in $\boldsymbol{w}$. Therefore, $\boldsymbol{w}$ is not the shortest walk, which contradicts the premise. □

For any simple connected graph $G(V, E)$, we can generate a fully connected graph $G'(V, E')$, where the V keeps unchanged but the weight of any edge (u, v) is the shortest distance between the u and v in $G(V, E)$. In other words, $G(V, E)$ and $G'(V, E')$ are the graphs from the perspective of physical routers and the probe path planner, respectively.

Theorem 1. Finding the shortest walk covering V of G equals to finding the shortest Hamilton Path in G'.

Proof. Based on Lemma 1, the shortest walk covering V can be partitioned into $n-1$ shortest paths according to $\boldsymbol{z}$. Obviously, $\boldsymbol{z}$ is the shortest Hamilton Path for G'. □

Theorem 1 illustrates that the path planning problem can be reduced to finding the shortest Hamilton Path of G', which is the TSP problem as G' is fully connected. Our probe path planning without the hop constraint is based on the most advanced heuristic algorithm for TSP, Christofides–Serdyukov algorithm [21], [22] which is proved to realize the best approximation ratio within a factor of $\frac{3}{2}$ of the optimal solution length. Generating the graph G' needs $O(|V|(|V| + |E|))$ time complexity by doing Breadth-First Search (BFS) starting at each vertex, and solving TSP can be reduced to $O(|V|^2)$ complexity if we use a greedy policy in its process, resulting in the overall time complexity as $O(|V|(|V| + |E|))$.

2) k-hop constraint: Path planning without the hop constraint has two drawbacks in a large-scale network: the probe packet may grow larger than the largest Ethernet frame along its path, and its transmission delay may be long, increasing the response time or collection time. Therefore, we propose the k-hop constraint that each path cannot exceed k hops, and the probe path planning problem becomes that using the minimum paths to cover all routers under the k-hop constraint.

Algorithm 1: k-hop Constraint Path Planning

Input: Graph $G(V, E)$, maximum hop k.
Output: Path set R.

1 $R \leftarrow \emptyset$, $Q \leftarrow G.V$, $\boldsymbol{P} \leftarrow$ allShortestPath(G)
2 **while** $Q \neq \emptyset$ **do**
3 $s \leftarrow$ findVertexWithMimumDegree(Q, G)
4 $\boldsymbol{p}_{\max} \leftarrow [s]$, $Q \leftarrow Q - \{s\}$
5 **while** $\boldsymbol{p}_{\max}$.hop $< k$ **do**
6 $\boldsymbol{p}' \leftarrow$ findMaxCoverPath($s, \boldsymbol{P}, Q, k - p_{\max}$.hop)
7 **if** $\boldsymbol{p}'$ is [] **then break**
8 $\boldsymbol{p}_{\max}$.append($\boldsymbol{p}'$), $Q \leftarrow Q - \boldsymbol{p}'$, $s \leftarrow p'_{\text{end}}$
9 $R \leftarrow R \cup \{\boldsymbol{p}_{\max}\}$
10 **return** R

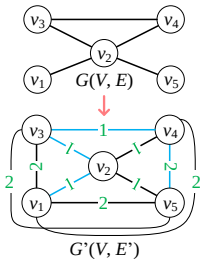


Fig. 7: Example graph $G(V, E)$ and $G'(V, E')$

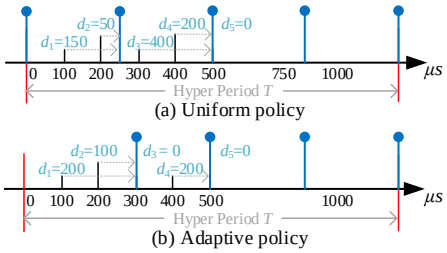


Fig. 8: Uniform probing and adaptive probing on the example flows.

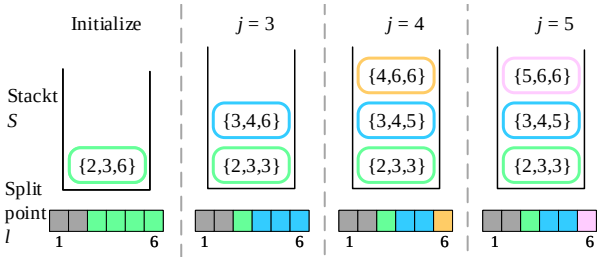


Fig. 9: Process of the DP-based probe sending algorithm for the flows in Fig. 8 and $i = 1, m = 3$.

This problem can be reduced to the k-path partition problem, which is NP-hard in general graphs [23]. We use a heuristic algorithm (Algorithm 1) to seek a sub-optimal solution. First, we start a path $p_{\max}$ at the vertex s with the minimum degree in the uncovered vertex set Q (as line 3). Then, from s, we search the sub-path p', which covers the maximum uncovered vertices among all paths within $k - p_{\max}$.hop. Then we append p' to $p_{\max}$ and update the uncovered set Q. If the hops of $p_{\max}$ are less than k, we continue the above process to find a next sub-path starting at the end of p' (as lines 2 to 8). We keep seeking a new path until Q is empty. It takes $O(|V|(|V| + |E|))$ time complexity to obtain all shortest paths, $O(|V|^2)$ to get the start vertex, and $O(|V|^2)$ to construct probe paths. Thus the overall time complexity is $O(|V|(|V| + |E|))$.

B. Probe Sending Moments

TSN-Guard sends probe packets in one or several hyper periods to collect misbehavior data. Note that probe sending moments influence the response time and bandwidth overhead significantly. Fig. 8 shows an example of seven TT flows with deadlines at 100, 200, 300, 300, 400, 400, 500 μs, and 1ms hyper period. If TSN-Guard sends probe packets every 250μs, half of them are redundant as no TT flow ends in the interval (500, 1000] μs. On the contrary, if it sends probes at 300 and 500 μs, no redundant probe packet exists. The former policy is the uniform probing and the latter one, which considers the distributions of TT flows' deadlines, is the adaptive probing.

To find the optimal adaptive probing, we conduct analysis as follows. In a hyper period T, we use an array $\boldsymbol{r} = \{r_1, r_2, ..., r_N\}$ to record N unique deadlines of all n TT flows in an ascending order. Assume that the system is allowed to send probes M times each T, so a sending policy can be represented by an ascending array $\boldsymbol{\pi} = \{t_1, t_2, ..., t_M\}$. Particularly, since the probes are generated periodically with the hyper period T, we have $t_{i+M} = t_i + T$. The nearest probe after a flow's deadline decides when the flow's entire misbehavior profile can be read from router caches, so we define the flow waiting time, $d_i = (t_{g(r_i)} - r_i)c_i$, where $g(r_i)$ is the index of the nearest following probe and c_i is the number of flows with the same deadline r_i. As shown in Fig. 8, d_i is 150, 50, 400, 200, and 0 μs using the uniform probing, and 200, 100, 0, 200, and 0 μs using the adaptive probing. Only acquiring a flow's entire misbehavior profile, the controller can analyze whether its end-to-end latency is violated, where is the root misbehavior and so on, so we can evaluate a policy by the mean of d_i for all flows, i.e. $D_{\boldsymbol{\pi}} = \frac{1}{n}\sum_{r_i \in [t_1, t_1+T)} d_i$ and the objective becomes finding the optimal policy $\boldsymbol{\pi}^*$ to minimize $D_{\boldsymbol{\pi}}$. To solve this optimization model, we first give a simple proposition to restrict the range of values in $\boldsymbol{\pi}^*$.

Proposition 1. If $\boldsymbol{\pi}^*$ is the optimal policy for $\min D_{\boldsymbol{\pi}}$, any $t_j^* \in \boldsymbol{\pi}^*$ must satisfy $t_j^* \in \boldsymbol{r}$.

Proof. Assume $t_j^* \in \boldsymbol{\pi}^*$, but $t_j^* \notin \boldsymbol{r}$. A policy $\boldsymbol{\pi'}$ is the same as $\boldsymbol{\pi}^*$ except that t_j^* is replaced by its most recent flow deadline $r_{f(t_j^*)}$, where $f(t_j^*) = \max\{i | r_i \leq t_j^*\}$. Because $D_{\boldsymbol{\pi'}} - D_{\boldsymbol{\pi}^*} < 0$, $\boldsymbol{\pi'}$ is better than $\boldsymbol{\pi}^*$, which contradicts the fact that $\boldsymbol{\pi}^*$ is the optimal solution. Hence, the assumption is false and the proposal is proved. □

1) DP-based Algorithm: Proposition 1 tells us that any probe sending moments in the optimal policy must be equal to a flow deadline, reducing the optimization problem to selecting the optimal M elements in $\boldsymbol{r}$. We propose DP-based algorithm with $O(N^3)$ (where $M \geq 3$) time complexity, and further optimize its time complexity to $O(N^2 \log N)$.

Algorithm 2: DP-Based Probe Sending

Input: $\boldsymbol{r}$, $\boldsymbol{c}$, and M
Output: Optimal policy
1 Zeros matrix: $\boldsymbol{\sigma}^{N\times 2N\times(M+1)}, \boldsymbol{L}^{N\times 2N\times(M+1)}$. $\boldsymbol{\pi}^*\leftarrow[]$
2 Get $\boldsymbol{\sigma}_{:,:,2}$ by Eq. (2)
3 **for** m in $3\rightarrow M+1$ **do**
4 **for** i in $1\rightarrow N$ **do**
5 $\boldsymbol{\sigma}_{i,:,m}, \boldsymbol{L}_{i,:,m}\leftarrow$ split$(i, m, \boldsymbol{\sigma}_{i,:,m-1}, \boldsymbol{\sigma}_{:,:,2})$
6 $s\leftarrow \arg\min_{s\in 1\rightarrow N}\boldsymbol{\sigma}_{s,s+N,M+1}$, $e\leftarrow s+N$
7 **for** m in $M+1\rightarrow 3$ **do**
8 $\boldsymbol{\pi}^*\leftarrow[L_{s,e,m}, \boldsymbol{\pi}^*]$, $e\leftarrow L_{s,e,m}$
9 **return** $[s, \boldsymbol{\pi}^*]$
// Key function
10 **Function** split$(i, m, \boldsymbol{x}, \boldsymbol{y})$ **:**
11 $\boldsymbol{x'}=[0]^{2N}$, $\boldsymbol{l}=[0]^{2N}$
12 **for** j in $i+m\rightarrow i+N$ **do**
13 **for** l in $i+m-2\rightarrow j-1$ **do**
14 update x'_j and l_j if l is a better split point.
15 **return** $\boldsymbol{x'}, \boldsymbol{l}$

First, we try to get the recursive formula, the key of the DP-based algorithm. For flows whose index is in the interval $[i, j]$, assume we have the optimal solution for sending probe m times (where $m\geq 2$), and the minimum sum of d_i for these flows is $\sigma_{i,j,m}$. Note that probe sending moments must include the two ends r_i and r_j for any interval $[i, j]$. When $m\geq 3$, we can split the interval $[i, j]$ at l by deploying $m-1$ probes in the first interval $[i, l]$ and two probes in the remaining interval, where $l\in\{i+m-2, i+m-1, \ldots, j-1\}$. Therefore,

$$\sigma_{i,j,m}=\min_{l=i+m-2\rightarrow j-1}\sigma_{i,l,m-1}+\sigma_{l,j,2},\quad m\geq 3. \tag{1}$$

The boundary conditions, i.e., $\sigma_{i,j,2}$, can be obtained below.

$$\sigma_{i-1,j,2}=\begin{cases}(r_j-r_i)c_i+\sigma_{i,j,2}, & i-1<j;\\ 0, & else.\end{cases} \tag{2}$$

At last, as the probe sending moments are periodical with the hyper period T, planning M groups of probe packets in any hyper period $[r_i, r_{i+T})$ actually costs $M+1$ ones for the index interval $[i, i+N]$. Therefore, the optimal policy $\boldsymbol{\pi}^*$ should realize $\min_{s=1\rightarrow N}\sigma_{s,s+N,M+1}$.

According to the recursive formula Eq. (1) and the boundary conditions Eq. (2), we design the DP-based algorithm (Algorithm 2). To record the policy $\boldsymbol{\pi}^*$, at the same time of minimizing $\sigma_{i,j,m}$ (as lines 3 to 5), we save the optimal split point ℓ in $L_{i,j,m}$, and finally construct the policy $\boldsymbol{\pi}^*$ by accessing the matrix $\boldsymbol{L}$ in a reverse order (as lines 7 to 8). As we need four loops of m, i, j, l, the overall time complexity is $O(MN^3)$, approximately equal to $O(N^3)$ when $N\gg M$.

2) Optimized DP-based algorithm: We further reduce the time complexity of the DP-based algorithm by Proposition 2.

Proposition 2. Assume $\ell=L_{i,j,m}$ is the optimal split point for $\sigma_{i,j,m}$. For any $j'>j$ and $i\leq l<\ell$, ℓ is a better split point for $\sigma_{i,j',m}$ than l.

Proof. As ℓ is the optimal split point for $\sigma_{i,j,m}$, it cannot be worse than l, i.e.,

$$\sigma_{i,\ell,m-1}+\sigma_{\ell,j,2}\leq\sigma_{i,l,m-1}+\sigma_{l,j,2}. \tag{3}$$

We can prove that $\boldsymbol{\sigma_{:,:,2}}$ satisfies the quadrilateral inequality, which is omitted due to space limits. Since $l\leq\ell\leq j\leq j'$,

$$\sigma_{l,j,2}+\sigma_{\ell,j',2}\leq\sigma_{l,j',2}+\sigma_{\ell,j,2}. \tag{4}$$

Combining Eq. (3) and (4), we have

$$\sigma_{i,\ell,m-1}+\sigma_{\ell,j',2}\leq\sigma_{i,l,m-1}+\sigma_{l,j',2}, \tag{5}$$

Therefore, ℓ is a better split point for $\sigma_{i,j',m}$ than l. □

Proposition 2 illustrates that once we find the optimal split point ℓ for the current range, the points ahead of ℓ do not need to be considered anymore, i.e., the split points are monotonic. Thus, we use a monotonic stack to reduce the time complexity.

Algorithm 3 shows the *optimizedSplit* function to replace the *split* function in Algorithm 2. Each item in the stack S stores an optimal split point ℓ, and its coverage $[left, right]$. First, we push $\{i+m-2, i+m-1, i+N\}$ to S, and then increase j from $i+m-1$ to $i+N$ to get $\sigma_{i,j,m}$ by one traverse. Each time, we calculate the coverage of j and pop S'top or modify

Algorithm 3: Optimized DP-Based Probe Sending

1 **Function** optimizedSplit($i, m, \boldsymbol{x}, \boldsymbol{y}$) **:**
2 $\boldsymbol{x}' = [0]^{2N}$, $\boldsymbol{l} = [0]^{2N}$
3 $S \leftarrow \emptyset$ // stack element $\{\ell, \texttt{left}, \texttt{right}\}$
4 S.push($\{i+m-2, i+m-1, i+N\}$)
5 **for** j in $i+m-1 \rightarrow i+N$ **do**
6 **while** $S \neq \emptyset$ and j *is better than* S.top().ℓ *to split* $[i, S.\text{top}().\text{left}]$ **do**
7 S.pop()
8 idx = BinarySearch($\boldsymbol{x}, j, S$.top())
9 **if** $idx > i+N$ **then continue**
10 **if** $idx \leq S$.top().right **then** S.top().right = $idx - 1$
11 S.push($\{j, idx, i+N\}$)
12 **while** !S.empty() **do**
13 **for** j in S.top().left $\rightarrow$ S.top().right **do**
14 $l_j \leftarrow S$.top().ℓ
15 $x'_j \leftarrow x_{l_j} + y_{l_j, j}$
16 S.pop()
17 **return** $\boldsymbol{x}', \boldsymbol{l}$

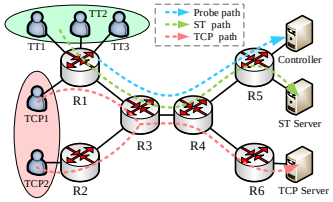
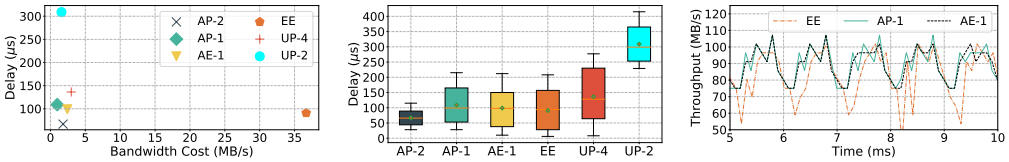


Fig. 10: Example network.



(a) Delay v.s. bandwidth cost (b) Delay (c) TCP throughput

Fig. 11: Performance of different monitoring mechanisms on the example network.

its right end to ensure that the coverage of j is not overlapped with those in S, and then push the item of j into S (as lines 6 to 11). At last, we can traverse the stack to get the optimal policy as lines 12 to 16 in Algorithm 3.

As shown in Fig. 9, $j = 2$ is the initial best split point for the interval [3, 6], and we then find $j = 3$ is better in the interval [4, 6], so we update the coverage of the top item to $\{2, 3, 3\}$ and push a new item of $\{3, 4, 6\}$ into the stack. If j is better than the entire coverage of the top item, e.g., $j = 5$, we will pop the stack and continue the process.

As the stack is monotonic, we can use binary search to get the left end of j with $O(\log(n))$ time complexity. Hence, the optimized DP-based algorithm takes $O(N^2 \log(N))$ time complexity when $N \gg M$.

V. Evaluation

In this section, we first present the performance of TSN-Guard on an example network scenario. Then we evaluate different algorithms for probe path planning and sending moments in large-scale networks by OMNET++ simulations.

A. Results of an Example Network

Our simulation experiments are based on NeSTiNg [24], a TSN simulation library for OMNET++. We modify NeSTiNg to realize the functions of TSN-Guard, including misbehavior discovery, caching, and INT-based collection mechanism.

To evaluate the performance, we study an example network as shown in Fig. 10. The topology is consistent with the examples in RFC 8655 [25], and there are three types of flows. First, we set up three TT flows to the TT server with the hyper period 1 ms. Each flow is composed of 10 successive packets of 200 Bytes. Their start times are 10, 110 and 210 μs, and deadlines are 100, 200, and 300 μs. To ensure their deadline requirements, we set the scheduled windows for them: the gate of the seventh queue is opened only in the time window [20, 40], [120, 140], and [220, 240] μs in the port of R1 to R3, and are postponed by 20 μs in the port of R3 to R4 and so on. To study TSN-Guard's response to packet misbehavior, we double the flow rate of the first TT flow, which may lead to packets being delayed and even discarded. Second, we set up two TCP flows, both towards the TCP server with 100 MB packets. Third, as routers R2 and R6 are not involved in the forwarding process of TT flows, we set one INT-probe packet with a path R1-R3-R4-R5-Controller. To avoid probe packets influencing TT packets, packets carrying misbehavior data are queued in the sixth queue, the second highest priority queue. The link rate is 1Gbps with 0.5 μs delay, and other settings are the same as NeSTiNg.

To prove that TSN-Guard is light-weight and fast-response, we compare the performance of adopting the following collection mechanisms, where M denotes the number of probing or exporting per hyper period: 1) AP-M, default in TSN-Guard, the

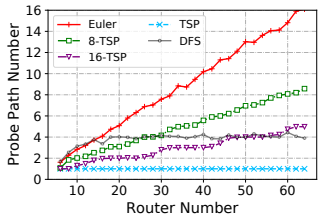


Fig. 12: Number of probe paths of the four algorithms.

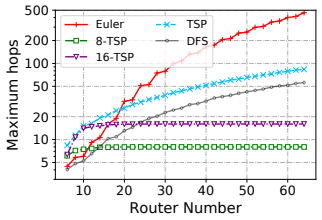


Fig. 13: Maximum path hops of the four algorithms.

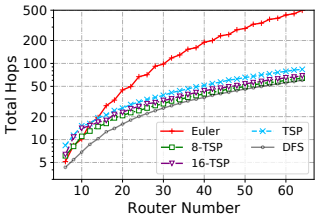


Fig. 14: Total path hops of the four algorithms.

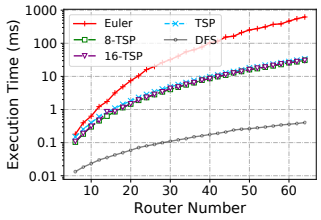


Fig. 15: Execution time of the four algorithms.

INT-based adaptive probing; 2) UP-M, the INT-based uniform probing; 3) AE-M, the direct exporting at adaptive moments; and 4) EE, the event-driven direct exporting.

Fig. 11(a) shows the average misbehavior collection delay versus bandwidth cost of different collection mechanisms. We can see that AP-2 realizes the best comprehensive performance, up to 1.8 MB/s bandwidth and average 66 μs delay, which are 5% and 73% of EE, respectively. The delay of EE is the second because of two reasons: first, the window settings block the transmission of misbehavior for a short while, as it cannot use the TT flows' windows; second, the burst exporting packets slow down the overall transmission speed. Moreover, its bandwidth cost is much larger than others. Uniform probing incurs longer delays, e.g., AE-4 is even worse than AP-1. AE-M achieves similar performance on delay and bandwidth as AP, but its fatal drawback is that it interrupts the controller as many times as the number of routers in every collection.

We compare the delay distribution of the mechanisms as shown in Fig. 11(b). Note that the delay variance of AP-2 is the smallest. While the minimum delay of EE is the smallest, its wide range increases its average delay. Since uniform probing ignores the flow distributions, the collection delays of UE-4 and UE-2 are significantly higher than others.

We also measure the throughput at the TCP server to estimate the influence of collection mechanisms on non-TT flows. Fig. 11(c) shows that TCP throughput suffers much when adopting EE. The performance of AP-1 and AE-1 is similar as their bandwidth costs are close, but AP-1 only interrupts the controller once every collection. In brief, AP is light-weight as it does not significantly impact the TCP flows and the controller even if the misbehavior burst is presented.

B. Results on Probe Path Planning

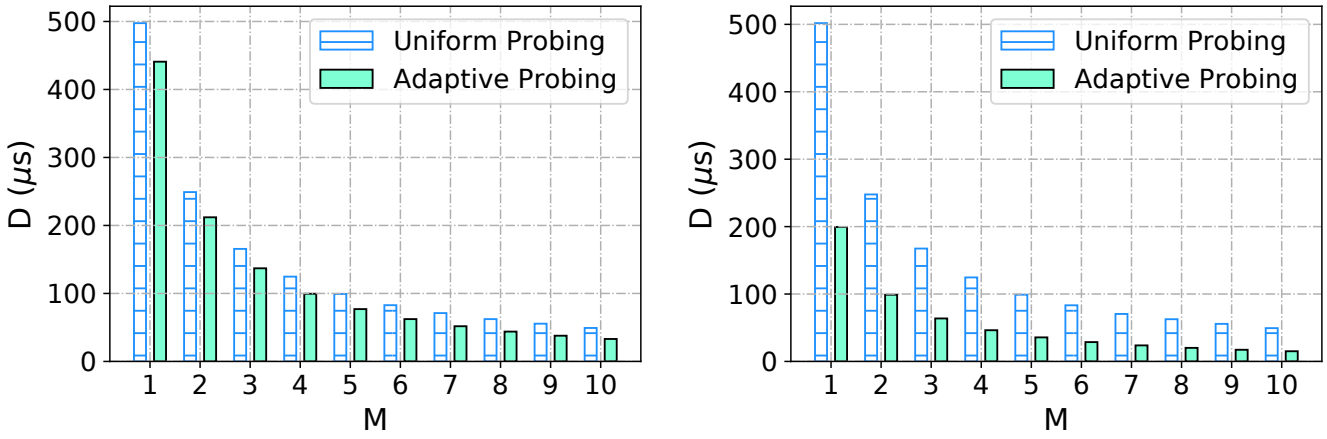
We compare the probe paths from the following algorithms: i) a balanced Euler trail based algorithm [19], which yields the most efficient paths for covering all edges, i.e., ports, ii) TSP based algorithm (§IV-A1), iii) k-TSP based algorithm (§IV-A2), where $k = 8$ or 16, and iv) Depth-First Search (DFS) based algorithm, which traverses all vertices, i.e., routers, in DFS manner. The number of routers is in the range of 6 to 64, and for each of them, we randomly generate 100 connected graphs to get the average performance.

Probe path number. Fig. 12 shows the path number as the number of routers increases. As TSP based path always covers all routers by one path, its line keeps horizontal, which means it brings the minimum interruptions to the controller. The number of k-TSP based paths increases linearly with the number of routers increasing, the slope is approximately equal to the number of routers divided by k. Thus, we can choose a higher k to reduce the path number, i.e., the interruptions to the controller. The number of DFS based paths fluctuates around 4, which is related to the connectivity of the graph. If we increase the edges during generating graphs, the average path number will decrease. Since Euler Trail covers all edges rather than vertices, it needs much more paths than other algorithms.

Maximum hops. Fig. 13 shows the maximum hops as the number of routers increases. As k-TSP path restricts the maximum number of hops, we can find both the lines of 8-TSP and 16-TSP enter a platform period gradually. Therefore, the maximum probe transmission time of k-TSP is stable as the number of routers increases. On the contrary, the other three lines keep increasing, especially for the Euler trail based one. Since TSP based path suffers repeated edges in most graphs, its maximum hops is more than the DFS based path which never visit nodes and edges repeatedly.

Total hops. Fig. 14 shows the total hops as the number of routers increases. DFS based path will not visit any repeated edge, so it keeps the minimum total hops and needs the minimum bandwidth. Nevertheless, the gap between it and the 8-TSP based path is lower than five, so its saved bandwidth is inconspicuous in contrast to its drawbacks of using more paths and larger maximum hops. As it is more likely to find a path without repeated edges when k is fewer, 16-TSP based path needs more hops than 8-TSP, and TSP based path needs more hops than any k-TSP based path. Because Euler trail covers all edges, its total hops are much higher than others.

Execution time. Fig. 15 shows the execution time as the number of routers increases. The four algorithms are written in Python 3.7 and run on a workstation with a 3.20GHZ Interl Core i7 CPU and 16G 2400Mhz DRAM. The DFS based algorithm needs the least time because its time complexity is $O(|V| + |E|)$. TSP and k-TSP based algorithms take approximately equal amount of execution time, about 30 ms under 64 routers, which is fast enough for probe path regeneration when the topology is changed. The Euler trail based algorithm takes the longest time, nearly one second under 64 routers.



(a) Uniform Distribution

(b) Normal Distribution

Fig. 16: Mean of flow waiting time using the two algorithms

C. Results on Probe Generation Moments

We compare the performance of the two algorithms for probe sending moments: adaptive probing and uniform probing. We set the hyper period T to 1 ms and generate 100 flows with deadlines in [0, T). The deadline obeys the uniform distribution or the normal distribution, whose mean value is 500 μs and standard deviation is 100 μs. Then, we can send probe packets M times per T, where $M \in \{1, 2, .., 10\}$. At last, we calculate the mean of flow waiting time D using the two algorithms, and the results are shown in Fig. 16.

Fig. 16(a) shows the results when the deadlines obey the uniform distribution. We can see adaptive probing outperforms the uniform probing all the time, and it can achieve average 33 μs delay, in contrast to 50 μs delay of the uniform probing, when M is ten. For the normal distribution shown in Fig. 16(b), adaptive probing performs much better than uniform probing, and it achieves average 15 μs delay in contrast to 49 μs delay of the uniform probing, when M is ten. Particularly, the performance of adaptive probing when M is four, is even better than that of uniform probing when M is ten. In conclusion, adaptive probing outperforms uniform probing, especially when the flow deadlines are unevenly distributed.

VI. Related Work

TSN Scheduling. Different from Time-Triggered Ethernet (TTE) [26], the schedule in TSN is per-queue rather than per-packet. To solve the packet disorder problem, [5] extends the methods in TTE [27], [28] to isolate the adjacent packets with enough space. To alleviate the packet isolation constraint, [6] uses the first-order theory of array for the packet-to-window map. [7] abstracts the TAS schedule as a Job-Shop Problem by assuming TT packets are forwarded immediately. These models all cost long time to get a feasible schedule, and sometimes they are unsolvable, so [29] proposes a Machine Learning based algorithm to speculate if the model is solvable in advance. ITP [8] is the first efficient schedule mechanism for CQF, though it cannot reduce the inherent defects of CQF.

The above theoretical methods are all static, which cannot adaptively adjust the schedule according to network feedback and changes. Although [30]–[32] reschedule efficiently when the network topology changes, they are not suitable for the forwarding misbehavior. In contrast, TSN-Guard provides network feedback to the controller for rescheduling.

Network monitoring. In the past, raw data of flows and device status in dataplane are collected to the software collectors in a push or pull mode. The collectors then analyze the raw data to figure out whether there are exceptions in network operations, such as SNMP [16], NetFlow [17], [18] and sFlow [33]. Their main drawback is that there is a trade-off between the data collection overhead and the monitoring accuracy. With the development of dataplane programmablity, such as P4 [13], many analytical tasks, such as detecting top-k heavy hitters [34], can be done in network devices locally. Some sketch-based monitoring solutions (e.g., [35]–[38]) have been proposed, which are flexible and low overhead for many monitoring tasks. However, the data collection mechanisms still use the pull mode or time-triggered push mode, both obeying the one-to-one principle between controller and devices.

INT [9] breaks the one-to-one principle, enabling packets to piggyback monitoring data along their paths. Pan et al. [19] use INT to realize the fine-grained and real-time data collection by elaborating the probe packet paths. The data collection mechanism proposed in this paper is inspired by [19]. Recent work [15] proposes an event-triggered push mode, which allows device to push data to the controller when some specific events are triggered. This mode enables the controller to find the network problem in its infancy and do adjustment in time.

All previous monitoring methods are not specialized for TSN and cannot monitor the packet forwarding misbehavior (first mentioned in [39]). Although Bu et al. [40] propose a TSN monitor, TSN-Insight, by extending the Precision Time Protocol

(PTP) packet to carry the monitoring data, it can only monitor the regular behavior. As far as we know, we are the first to try to monitor misbehavior in TSN.

VII. Conclusion

In this paper, we propose TSN-Guard to monitor the packet forwarding misbehavior of time sensitive flows in TSN. We have deeply analyzed the root causes of this kind of violation and completed a detailed system implementation. We have designed a light-weight misbehavior collection scheme using INT probes. To evaluate the performance of TSN-Guard, we model it on the OMENT++ simulator and carry out extensive experiments. The results show that TSN-Guard outperforms other schemes on incurred overhead, including bandwidth consumption and interference to controller, and response time.

References

[1] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. ElBakoury, "Ultra-low latency (ULL) networks: The IEEE TSN and IETF DetNet standards and related 5G ULL research," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 88–145, 2018.

[2] Time-Sensitive Networking Task Group. [Online]. Available: https://www.ieee802.org/1/pages/tsn.html

[3] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic," *IEEE Std*, pp. 1–57, Mar. 2016.

[4] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks - Amendment 29: Cyclic Queuing and Forwarding," *IEEE Std*, pp. 1–30, 2017.

[5] S. S. Craciunas, R. S. Oliver, M. Chmelík, and W. Steiner, "Scheduling Real-Time Communication in IEEE 802.1 Qbv Time Sensitive Networks," in *Proceedings of the ACM International Conference on Real-Time Networks and Systems*, 2016, pp. 183–192.

[6] R. S. Oliver, S. S. Craciunas, and W. Steiner, "IEEE 802.1 Qbv gate control list synthesis using array theory encoding," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 13–24.

[7] F. Dürr and N. G. Nayak, "No-wait packet scheduling for IEEE time-sensitive networks (TSN)," in *Proceedings of the ACM International Conference on Real-Time Networks and Systems*, 2016, pp. 203–212.

[8] J. Yan, W. Quan, X. Jiang, and Z. Sun, "Injection Time Planning: Making CQF Practical in Time-Sensitive Networking," in *Proceedings of the IEEE INFOCOM*, 2020.

[9] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[10] OMNeT++, Discrete Event Simulator. [Online]. Available: https://omnetpp.org

[11] S. Waldhauser, B. Jaeger, and M. Helm, "Time Synchronization in Time-Sensitive Networking," *Network*, vol. 51, 2020.

[12] D. Shrestha, Z. Pang, and D. Dzung, "Precise clock synchronization in high performance wireless communication for time sensitive networking," *IEEE Access*, vol. 6, pp. 8944–8953, 2018.

[13] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[14] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks–Amendment 28: Per-Stream Filtering and Policing," *IEEE Std*, pp. 1–65, Sep. 2017.

[15] J. Kučera, D. A. Popescu, H. Wang, A. Moore, J. Kořenek, and G. Antichi, "Enabling Event-Triggered Data Plane Monitoring," in *Proceedings of the Symposium on SDN Research*, 2020, pp. 14–26.

[16] C. Hare, "Simple Network Management Protocol (SNMP)." 2011.

[17] B. Claise, G. Sadasivan, V. Valluri, and M. Djernaes, "Cisco Systems Netflow Services Export Version 9," 2004.

[18] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "FlowCover: Low-cost Flow Monitoring Scheme in Software Defined Networks," in *2014 IEEE Global Communications Conference*. IEEE, 2014, pp. 1956–1961.

[19] T. Pan, E. Song, Z. Bian, X. Lin, X. Peng, J. Zhang, T. Huang, B. Liu, and Y. Liu, "Int-path: Towards optimal path planning for in-band network-wide telemetry," in *Proceedings of the IEEE INFOCOM*, 2019, pp. 487–495.

[20] C. Filsfils, N. K. Nainar, C. Pignataro, J. C. Cardona, and P. Francois, "The Segment Routing Architecture," in *Proceedings of the IEEE GLOBECOM*. IEEE, 2015, pp. 1–6.

[21] N. Christofides, "Worst-case Analysis of a New Heuristic for the Travelling Salesman Problem," Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Tech. Rep., 1976.

[22] R. van Bevern and V. A. Slugina, "A Historical Note on the 3/2-Approximation Algorithm for the Metric Traveling Salesman Problem," *Historia Mathematica*, 2020.

[23] G. Steiner, "On the k-Path Partition of Graphs," *Theoretical Computer Science*, vol. 290, no. 3, pp. 2147–2155, 2003.

[24] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, and K. Rothermel, "NeSTiNg: Simulating IEEE Time-Sensitive Networking (TSN) in OMNeT++," in *Proceedings of the IEEE Conference International Conference on Networked Systems (NetSys)*, 2019, pp. 1–8.

[25] N. J. Farkas, Finn, P. Thubert, and B. Varga, "Deterministic Networking Architecture, ser. RFC 8655," *Internet Engineering Task Force - (IETF)*, Oct. 2019.

[26] T. Steinbach, H.-T. Lim, F. Korf, T. C. Schmidt, D. Herrscher, and A. Wolisz, "Tomorrow's in-car Interconnect? A Competitive Evaluation of IEEE 802.1 AVB and Time-Triggered Ethernet (AS6802)," in *Proceedings of the IEEE Vehicular Technology Conference (VTC Fall)*, 2012, pp. 1–5.

[27] F. Pozo, G. Rodriguez-Navas, H. Hansson, and W. Steiner, "SMT-Based Synthesis of TTEthernet Schedules: A Performance Study," in *Proceedings of the IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2015, pp. 1–4.

[28] W. Steiner, "An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-Hop Networks," in *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2010, pp. 375–384.

[29] T. L. Mai, N. Navet, and J. Migge, "On the Use of Supervised Machine Learning for Assessing Schedulability: Application to Ethernet TSN," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, 2019, pp. 143–153.

[30] N. Wang, Q. Yu, H. Wan, X. Song, and X. Zhao, "Adaptive Scheduling for Multicluster Time-Triggered Train Communication Networks," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1120–1130, 2019.

[31] Z. Li, H. Wan, Z. Pang, Q. Chen, Y. Deng, X. Zhao, Y. Gao, X. Song, and M. Gu, "An Enhanced Reconfiguration for Deterministic Transmission in Time-Triggered Networks," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1124–1137, 2019.

[32] K. Lee, T. Park, M. Kim, H. S. Chwa, J. Lee, S. Shin, and I. Shin, "MC-SDN: Supporting Mixed-Criticality Scheduling on Switched-Ethernet Using Software-Defined Networking," in *Proceedings of the IEEE Real-Time Systems Symposium (RTSS)*, 2018, pp. 288–299.

[33] P. Phaal, S. Panchen, and N. McKee, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks," 2001.

[34] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-Hitter Detection Entirely in the Data Plane," in *Proceedings of the Symposium on SDN Research*, 2017, pp. 164–176.

[35] Q. Huang, P. P. Lee, and Y. Bao, "Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference," in *Proceedings of the ACM SIGCOMM Conference*, 2018, pp. 576–590.
[36] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring With Univmon," in *Proceedings of the ACM SIGCOMM Conference*, 2016, pp. 101–114.
[37] M. Yu, L. Jose, and R. Miao, "Software Defined Traffic Measurement with OpenSketch," in *Proceedings of the USENIX NSDI*, 2013, pp. 29–42.
[38] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the ACM SIGCOMM Conference*, 2018, pp. 561–575.
[39] V. Addanki and L. Iannone, "Moving a Step Forward in The Quest for Deterministic Networks (DetNet)," in *Proceedings of the IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 458–466.
[40] T. Bu, Y. Yang, X. Yang, W. Quan, and Z. Sun, "TSN-Insight: An Efficient Network Monitor for TSN Networks," in *Proceedings of the Asia-Pacific Workshop on Networking (APNET)*, 2019.