Cedar: A Reconfigurable Data Plane Telemetry System

Chris Misa DRP Report. Advisors: Ramakrishnan Durairajan and Reza Rejaie

ABSTRACT

Modern telemetry systems rely on programmable switches to perform the required operations within the data plane in order to scale with the rate of network traffic. These systems create a stream processing pipeline for all telemetry operations and statically map a subset of operations to switch resources. These systems exhibit two inherent restrictions: First, the fraction of operations in the data plane decreases with the number and complexity of telemetry tasks. Second, changing telemetry tasks requires rebooting the switch to install a new telemetry task, *i.e.*, no agility.

To address these restrictions, this paper presents *Cedar*, a first-of-its-kind re**c**onfigurabl**e** **d**atapl**a**ne telemet**r**y system. *Cedar* is based on a new Broadcom ASIC with the BroadScan module that offers a wide range of reconfigurable telemetry operations. Leveraging this unique capability, we introduce and explore reconfigurable data plane telemetry systems design issues. *Cedar* can periodically re-map different telemetry operations to limited switch resources. This *temporal partitioning* of telemetry tasks leads to scalability and agility of *Cedar*. Our experimental evaluations of *Cedar* demonstrate that both the required switch resources and volume of generated telemetry reports in each period are very small and gracefully scale with the number of tasks.

1 INTRODUCTION

Network telemetry systems provide continuous and real-time measurements about the state of the network [29] which is critical for network operators to detect increasingly complex events ranging from performance degradation to security attacks. Network telemetry systems such as Gigascope [16], Chimera [11], and NetQRE [32] can analyze network traffic for a wide range of telemetry tasks using expressive, high-level languages on general-purpose CPUs. However, it is prohibitively costly for such CPU-based systems to scale with the rate of traffic in modern networks. While data plane telemetry systems (*e.g.*, Marple [24], OpenSketch [30], Sonata [19]) can scale with traffic rates, the scarcity of data plane resources, e.g., memory, limit either the number of supported telemetry tasks (*i.e.*, queries) [24] or performance gains [19] for these systems.

State-of-the-art data plane telemetry systems (see Table 1) generally rely on programmable switches [10, 12]. At the heart of these systems is the "compile-and-deploy" model which maps telemetry operations statically to switch resources (*e.g.*, stages of a Tofino switch [7]). As a result, these programmable data plane telemetry systems exhibit two inherent restrictions. First, these systems must create a stream processing pipeline that incorporates the entire set of operations associated with all the target telemetry tasks identified by operators. Therefore, as the size of the processing pipeline grows with the number and complexity of telemetry tasks, *only* a smaller fraction of all operations can be executed in the data plane. Second, to install or remove (or even change the features of) any telemetry task, operators need to reboot the switch to deploy a new program, *i.e.*, no agility.

Framework	Data plane	Reconfig.	HW Switch
NetQRE [32]		✓	
DREAM [22]		✓	
Marple [24]	✓		
Sonata [19]	✓		✓
Cedar	✓	✓	✓

Table 1: Comparison of *Cedar* and other query-based telemetry systems.

To address these restrictions, in this paper we present a first-of-its-kind re**c**onfigurabl**e** **d**ata pl**a**ne telemet**r**y system, called *Cedar*. *Cedar* is based on a commercial network switch with the new Broadcom BCM57340 chipset that is equipped with the *BroadScan* hardware module. The Broadcom ASIC in the switch is not fully programmable, such as Barefoot Tofino [7], but the BroadScan module offers a wide range of telemetry operations that can be easily reconfigured on-the-fly (at runtime) [8]. Thus, we call this a *reconfigurable switch*. Leveraging this unique capability, we introduce and explore reconfigurable data plane telemetry systems by making the following contributions:

Programmable vs. Reconfigurable Data Planes. We compare and contrast the key capabilities and limitations of programmable and reconfigurable data plane telemetry systems (§ 2). We show that reconfigurable systems can periodically, once per epoch, assess the required operations, reconfigure the data plane accordingly, and then collect the relevant information from the network. By *temporal partitioning* of telemetry tasks, limited switch resources can be periodically allocated to different operations. Therefore, a significantly larger fraction of telemetry operations is performed in the data plane, ensuring seamless scaling with an increase in the number and complexity of telemetry tasks. The runtime

flexibility of reconfigurable data plane telemetry systems easily facilitates any change in the target telemetry tasks or their features on-the-fly.

Design and Implementation of *Cedar*. We examine key design considerations of reconfigurable telemetry systems such as proper encoding of input tasks and reconfiguration intervals (§ 3). We present the implementation[1] of *Cedar* (§ 4) to demonstrate how it effectively harnesses the unique capabilities of BroadScan. For example, to the best of our knowledge, *Cedar* is the first data plane telemetry system that places the switch ASIC directly in the control loop. *Cedar* has a modular software architecture that can serve as an extensible platform for reconfigurable data plane telemetry. New telemetry tasks can be easily expressed as a flowchart and then incorporated into the system.

Experimental Evaluation of *Cedar*. Using a reconfigurable switch (BroadScan-enabled BCM 57340 series System Verification Kit), we conduct trace-driven experiments to incrementally evaluate the performance and accuracy of single and multiple telemetry tasks on *Cedar* (§ 5). Our key findings can be summarized as follows. *(i) Cedar* can execute individual telemetry tasks using less than 200 operations per epoch in the data plane on average while detecting target events within 22-42 seconds. *(ii)* More importantly, the number operations and reported tuples per epoch gracefully scales with the number of concurrent telemetry tasks. These findings clearly illustrate that *Cedar* can execute all the telemetry operations in the data plane as the load of telemetry operations grow. *(iii)* Despite the periodic reporting by *Cedar*, it is able to reduce the telemetry traffic (*i.e.*, reported tuples) by up to four orders of magnitude compared to other data plane telemetry systems.

Our efforts in developing this work are as a third party, not associated with Broadcom Inc. apart from using their hardware.

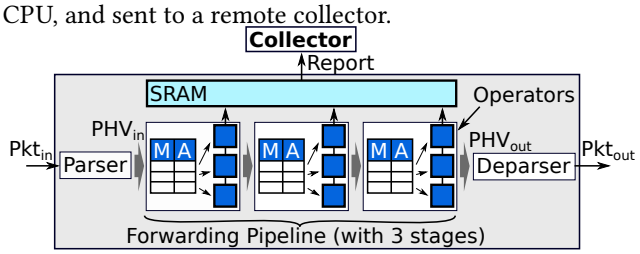
This work does not raise any ethical issues.

2 PROGRAMMABLE VS. RECONFIGURABLE DATA PLANES

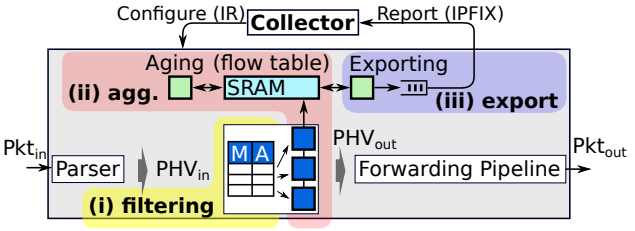
In this section, we present the abstract packet processing model for programmable and reconfigurable data planes and their implications on network telemetry systems.

2.1 Packet Processing Model

Programmable Switch. Data plane telemetry systems that rely on protocol-independent switch architecture (PISA) (*e.g.*, RMT [13], Tofinio[7], Netronome [6]) offer programmable parsing and packet-processing pipelines with a fixed number of operators arranged in consecutive physical stages (see Figure 1a). Each stage features a self-contained match-action (MA) table on packet header vectors (PHV) which contain fields extracted by the parser as well as custom metadata fields. If a header field in a PHV matches a rule in a table, a set of customized (stateful or stateless) operations is performed on the PHV before it is forwarded to the next stage. Each stage can perform a fixed number of independent operations in parallel (*i.e.*, the width of the processing pipeline) where the MA table of that stage triggers different combinations of operations. As a natural consequence of this pipeline design, the result of these operations in a particular stage can only be used by operations in later stages. The operations in each stage also access a small amount (c.a. megabytes) of high-speed SRAM which can be used to maintain state (*e.g.*, in a flow table) across packets. State information in the flow table must be periodically read from this SRAM, by the switch's CPU, and sent to a remote collector.



(a) PISA



(b) BroadScan

Figure 1: Architecture of PISA switch and switch with BroadScan.

As identified by prior work [19, 24], the PISA maps naturally to the stream processing network telemetry model [16]. The goal is to create a processing pipeline that sequentially performs all the required operations to distill raw network traffic into useful measurement results. This linear order of execution can be represented as a directed, acyclic graph (DAG) where each node performs an operation. In this model, a telemetry task is described as a query on an abstract stream of tuples (*i.e.*, PHVs) using filter, map, and reduce operations. Since switch resources are shared between telemetry and basic switch operations (*e.g.*, forwarding, loop detection, encapsulation), only a subset of switch resources/stages can be allocated to telemetry tasks. The query is compiled to determine the proper mapping of required operations to different stages of the switch to create the processing pipeline

[1] Upon publication, we will make *Cedar*'s source code publicly available to enable other researchers in the community to validate our results, extend the capabilities of *Cedar*, and develop new telemetry tasks.

for telemetry tasks within the data plane using the allocated stages. Therefore, *in programmable data plane telemetry systems, both the width and depth of the processing pipeline within the data plane are limited by the switch resources allocated to telemetry operations.*

While PISA switches offer the ability to program all aspects of switch operations, facilitating rapid prototyping of new data plane telemetry tasks, their "compile-and-deploy" model hinders any change in the mapping of operations to different stages as it requires rebooting the switch with a new program which takes 10s of seconds of downtime. Even state-of-the-art techniques used for the "warm reboot" feature found in Tofino switches imposes seconds of downtime [5]. The key implication of this restriction is that *all the required telemetry tasks with most of their parameters must be determined a priori at the deployment time or the switch should be rebooted to incorporate any changes.*

Reconfigurable Switch. The packet processing model for a switch with reconfigurable telemetry capabilities is essentially the same as the PISA model. The key difference is the ability to arbitrarily change the telemetry operations without disturbing other switch functions, thanks to recent innovations in switch hardware (*e.g.*, Broadscan described in § 4.2). As shown in Figure 1b, the BroadScan hardware module can be compared to a PISA switch with a single stage. It has a single match-action table that is placed between the parser and the main ingress pipeline. However, since telemetry operations can be changed (within millisecond) on-the-fly without disrupting other switch operations, Broadscan motivates a dynamic "divide-and-reconfigure" model for using the switch's telemetry resources. The distinct operations of one (or multiple concurrent) telemetry task(s) can be executed in hardware sequentially over the course of several brief epochs, thus deferring decisions about particular operations and resource allocations to task runtime. In a sense, the required processing pipeline for an (arbitrarily long) telemetry task can be incrementally (re)configured in the data plane at runtime. Said differently, *only the width of the processing pipeline is limited in a reconfigurable switch model.*

BroadScan also extends the classic match-action paradigm with several telemetry-specific hardware processes that enable a task to efficiently manage the aggregation state in the flowtable and to control the exporting strategy of telemetry reports emitted *directly* from switch hardware. These capabilities naturally lead to describing a BroadScan configuration in terms of three basic operations: filtering, aggregation, and exporting (described in § 4.2). Filtering works similarly to the filter operation in the stream processing model. Aggregation corresponds roughly to a reduce operation in the stream processing model, but BroadScan offers telemetry-specific operations such as aging-out old table entries and counting table overflow events. Export manages the process of sending aggregation results to the collector entirely in hardware, freeing the switch CPU for the task of periodically reading and exporting the telemetry state. For more details, see Appendix 8.3.

Per-Stage Limitations. In both of the above models, the main limitation in each stage of the processing pipeline is that each PHV can match only a single action in the MA table which determines the applied operation to each packet, *i.e.*, independent parallel operations in each stage can only operate on a mutually-exclusive subset of incoming packets. For example, consider the two tasks of counting sources and counting destinations. Packets can be independently grouped by sources and destinations if we impose a filter to separate incoming PHVs into two, disjoint groups (*e.g.*, UDP and TCP packets). However, we cannot simultaneously count sources and destinations for two overlapping groups (*e.g.*, UDP packets and packet from a particular subnet). A PISA switch could utilize separate stages of the processing pipeline to group both sources and destinations for all PHVs to overcome this limitation. *Cedar* addresses this limitation by leveraging BroadScan to count sources and destinations in consecutive epochs.

2.2 Implications on Telemetry Systems

To highlight the implications of the two models on telemetry systems, we consider the telemetry task of detecting victims of DNS reflection volumetric DDoS attacks [27] (Figure 2). In particular, we are interested in detecting destinations that receive DNS packets from a large number of sources (threshold 2), but only on the condition that this total DNS traffic is sufficiently large (threshold 1). To facilitate a head-to-head comparison, the left side of Figure 2 presents this task as a flowchart, while the right side shows the corresponding operations in a query-like format that might be issued in current programmable data plane telemetry systems (*e.g.*, [19]). This example reveals a few important implications of each approach to data plane telemetry as follows:

Representation of Tasks. Telemetry tasks often require to check whether a certain condition is met before executing the rest of the task, leading to an iterative pattern of execution. For example, loop 1 in Figure 2 is a *triggering condition* (explained below) that should be satisfied before considering other operations. More importantly, the number of iterations for each loop depends on the network conditions and is determined during runtime. Flowcharts offer a proper representation of telemetry tasks because they accurately encode key dependencies between operations as well as iterative execution patterns. Returning to Figure 2, this flowchart encapsulates a key dependency between operations: if the number of DNS packets is below threshold 1, none of the

other operations need to be evaluated. We refer to such dependencies as *trigger conditions*. Since we expect such attacks to be relatively rare under normal circumstances, flowcharts will spend a majority of their evaluation time in the first loop, simply counting their respective packet types or performing other simple operations, leaving the majority of the BroadScan resources available for other tasks. In contrast, a query is only able to express a sequential order of execution across different operations (or a DAG) which is suited for a stream processing pipeline but cannot capture these iterative (cyclic) operations.[2]

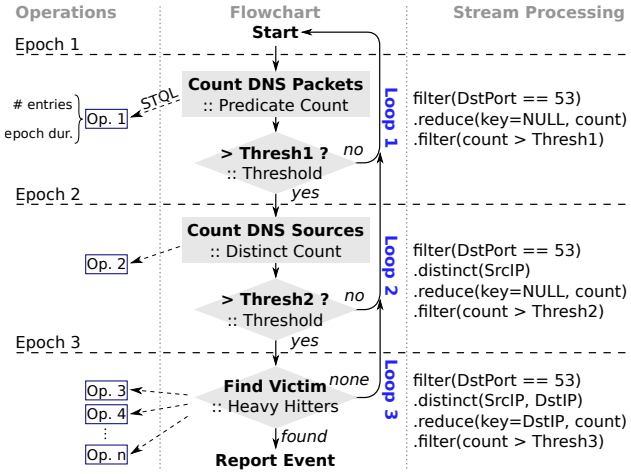


Figure 2: Example flowchart expressing a telemetry task for detecting DNS reflection attacks and the corresponding queries (for each loop) in the stream processing paradigm. key=NULL **indicates reduction of all packets in the epoch to a single group.**

Agility of Telemetry Systems. The offered agility of reconfigurable telemetry systems in allocating data plane resources leads to a significantly higher level of scalability and flexibility for these systems in action. In contrast, the "compile-and-deploy" model of programmable telemetry systems implies that the processing pipeline should incorporate all operations of all required telemetry tasks since the execution path of each task is not known a priori. This leads to inefficient utilization of the already-limited switch resources. To illustrate this issue, consider a linear task that is represented by a DAG with a few branches of operation. In each round of execution, only the operations on a single execution path (a branch of the DAG) are used while resources for all operations must be allocated. This inefficiency reduces the ability of the programmable telemetry systems to perform all operations in the data plane and limits their scalability.

[2]Note that the corresponding queries for each loop in Figure 2 could be combined in parallel in a stream processing pipeline, with their results evaluated and joined conditionally in a system like Sonata [19]. However, this strategy results in very inefficient use of data plane resources.

Similarly, if network operators decide to change features of a telemetry task or install/remove a telemetry task (*e.g.*, to cope with a newly identified attack or performance event), they must compile and deploy the new program which requires rebooting the switch leading to 10s of seconds of downtime. On the other hand, reconfigurable telemetry systems allow the operator to enable and disable desired telemetry tasks, or change resource allocations between tasks, by adjusting the corresponding flowcharts, without impacting other switch functions. Even if there are not sufficient data plane resources to accommodate a new task, the operator can retry at a later time or the runtime can start the task as soon as resources become available.

Telemetry Traffic & State Management. In both the approaches, periodic telemetry reports from the switch enables the collector to maintain the state of individual telemetry tasks and detect the target events. In reconfigurable telemetry systems, the output of operations in each epoch (*i.e.*, single stage) must be reported to the collector. However, in programmable telemetry systems, the output of operations in each stage are fed into the related operations in other stages of the switch, and only one (or a small number of stages) report their output to the collector. Therefore, if all operations fit in the data plane, the rate of telemetry reports from the switch to the collector could be larger in reconfigurable systems. However, as the number of operations per tasks or the number of concurrent tasks increases, it is more likely that their operations do not fit in the data plane of programmable systems. In these cases, the switch should send reports and mirror filtered data packets to the collector that leads to a significantly larger rate of traffic to the collector. This, in turn, increases the complexity of statement management between the switch and the collector [19].

Diversity of Offered Telemetry Options. *Cedar* currently supports a collection of telemetry operations and associated parameters that are most commonly used by network operators. However, we do not support all possible monitoring options because of hardware constraints. For example, BroadScan currently does not measure queue-length on each port or observed delay by each packet.[3] Similarly, the hardware-based implementation of different telemetry-related methods in reconfigurable switches are reasonably optimized but cannot be changed. For example, BroadScan implements its own hash function for managing the flow table that has a first miss utilization of 92% and average utilization of 98%. However, this hash function cannot be updated with a better alternative (*e.g.*, Cuckoo Hash [25]) and its output can only be used for indexing into the flow table. In a programmable telemetry system, on the other hand, one can program and

[3]Broadcom has indicated that the next generation of BroadScan will incorporate these operations.

reboot the switch to select from a variety of hardware hash function implementations and use the resulting hash values in new and creative ways (*e.g.*, in the deployment of sketching techniques [15, 21]).

3 DESIGN CONSIDERATIONS

In this section, we examine a few basic considerations in the design of a reconfigurable data plane telemetry system.

Temporal Partitioning of a Task. To enable the partitioning of individual telemetry tasks on-the-fly, the representation of each task should specify *(i)* the dependency between different required operations, and *(ii)* the required resources, namely the number of flow table entries, for each operation. The inter-operation dependency is used to identify one (or multiple independent) operation(s) in each task that can be performed in the next epoch. The required number of flow table entries for individual operations reveals whether these operations can be executed in one epoch. If the aggregate demand of all planned operations in an epoch exceeds the size of the flowtable, the system must implement a resource scheduling scheme to manage such an overloaded epoch, *e.g.*, by slightly delaying a subset of operations while considering the temporal dependencies between consecutive operations. *Cedar* currently implements a simple FIFO scheduling of operations and we leave investigation of more complex schemes to future work.

Setting Epoch Duration. Duration of an epoch, which we define as the time between two consecutive reconfigurations of the telemetry operations in the data plane, is a key design parameter of a reconfigurable telemetry system which has an opposite effect on the accuracy of individual operations and the completion time of related tasks. On the one hand, very short epochs could introduce noise in measured traffic features. For example, as shown in Figure 3, the number of unique sources exhibits larger variation when measured over very short time scales (a coefficient of variation of over 6% with 0.1 second epochs). Furthermore, epoch duration should be significantly longer than the reconfiguration time for the data plane to avoid error induced by packets missed during reconfiguration.

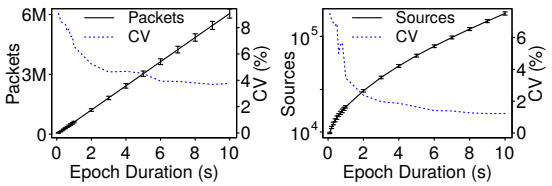


Figure 3: The effect of epoch duration on the number of observed packets and number of distinct sources for CAIDA traffic trace. Error bars show std. deviation.

On the other hand, increasing epoch duration linearly extends the completion time of tasks that require a fixed number of epochs. For example, when executing the MRT algorithm with an expansion ratio of 4, a search of the IPv4 address space takes 16 epochs in the worst-case. If each epoch takes 10 seconds, this leads to a worst-case turnaround time of over 2.5 minutes which could be too long for some tasks. More importantly, the number of required flow entries for an operation may need to increase with the epoch duration to maintain the same level of accuracy since the feature magnitude tends to increase (see again Figure 3).

Therefore, we believe that there is sweet spot for epoch duration that strikes the balance between the above factors. Plots in Figure 3 illustrate how the value of two traffic features along with their coefficients of variation (CV) across different segments of traffic evolves with epoch duration. These examples illustrate that epoch duration of 2-3 second offers the sweet spot for a telemetry task based on these features. A similar type of analysis can be performed for any task using offline processing on traffic traces, or online estimation of the variability at small epoch durations and the increasing magnitude at large epoch durations.

Disjoint Packets Across Epochs. Temporal partitioning of a telemetry task implies that operations in different epochs observe temporally disjoint segments of network traffic. This raises the following key question: *how does the temporally disjoint nature of operations in individual telemetry tasks affect their accuracy when implemented in reconfigurable switch data planes?*

To address this question, we note that many current telemetry systems assume that their measured traffic features remain detectable (*e.g.*, above a certain threshold) for at least 10s of seconds to minutes while the target security or performance-related event of interest occurs. Therefore, these features can be captured at any time while the event is detectable. To illustrate this fact, plots in Figure 4 present the number of unique sources per second in two different DNS reflection attack traces [27]. While these attacks have widely different intensities (in terms of the absolute number of sources), their main feature remains measurable/high for minutes. Existing telemetry systems such as DREAM [22] and Sonata [19] already rely on temporally disjoint measurement, using online accuracy estimation and dynamic query refinement respectively, *without* any accuracy concern.

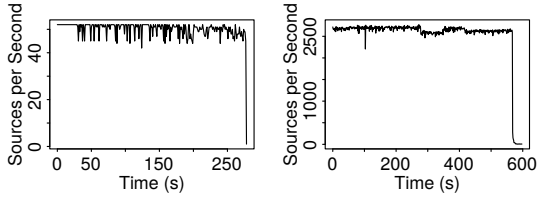


Figure 4: Number of observed sources per second in two traces of DNS reflection attack traffic.

In the design and implementation of *Cedar* (§ 4), we use flowcharts (instead of queries) because they address these considerations by directly accommodating iterative task.

4 IMPLEMENTATION OF *CEDAR*

This section describes the implementation details of *Cedar* including the different components and their interactions and shows how *Cedar* meets all the design considerations (in § 3).

4.1 System Overview

Figure 5 depicts an overview of *Cedar* and how the different components interact with each other. We rely on a switch that is based on the Broadcom BCM57340 chipset and offers reconfigurable monitoring capabilities. The monitoring capabilities of the switch are (re)configured by a collector application software.[4] The collector receives individual telemetry tasks from an operator and manages their execution. A telemetry task is described as a flowchart that expresses the ordering and dependency between all operations needed to complete that task. Each node in the flowchart may execute one or more operations, possibly over multiple epochs, before transitioning to another node. In other words, operators who generate flowcharts to achieve a specific telemetry task only need to deal with nodes as a high-level abstraction (*e.g.*, using a node to find heavy hitters) and do not need to bother with the low-level operations required (*e.g.*, the iterations of the MRT algorithm).

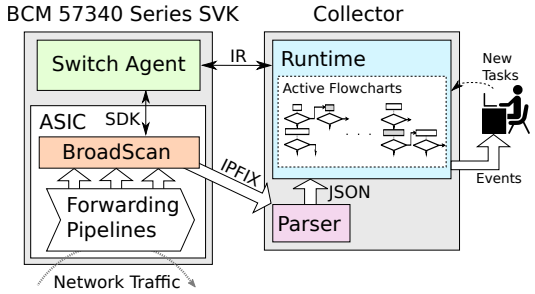


Figure 5: Overview of *Cedar* showing closed circle of communication used to evaluate monitoring tasks.

To perform "temporal partitioning" of a task, the runtime maintains a list of active nodes for each task (starting with the root) and determines, for all running tasks, which operations from these active nodes should be executed in each epoch. It then determines the required monitoring capabilities in the switch and emits the relevant commands to the switch agent. The switch agent, running on the switch's CPU, receives commands from the collector, and (re)configures the monitoring capabilities of the switch through a module in the switch ASIC, called BroadScan (described in § 4.2). Once configured, Broadscan directly sends IPFIX packets (using a format fixed by the operation) that contain the monitored information to the collector based on the reporting conditions (*e.g.*, periodic, once a threshold is reached). At the end of each epoch, the runtime collects the resulting reports from the parser for each operation and feeds these results back into the active nodes to complete any required post-processing. Active nodes then inform the runtime to make a transition to the next node in the flowchart, in which case the runtime adds the next node to the list of active nodes, or that they have more operations to execute, in which case the runtime holds the node in the list of active nodes.

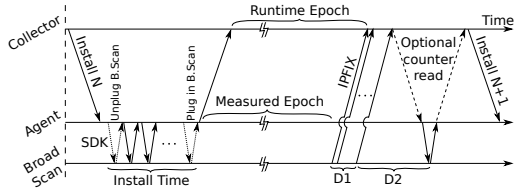


Figure 6: Sequence of communication between collector (runtime), switch agent, and BroadScan hardware.

Hardware reporting and challenges. A key feature of this interaction, as shown in Figure 6, is that, since telemetry reports are generated and transmitted directly from switch hardware, the switch ASIC is placed in the control loop of the telemetry process. While bypassing the switch CPU improves the scalability of the system, it also raises three design challenges that must be addressed.

- The incremental configuration of the BroadScan hardware (labeled Install Time in Figure 6) takes a long time (tens of milliseconds) compared to the rate of the network traffic driving the monitoring hardware. The BroadScan hardware must be "unplugged" before modifying its configuration and "plugged in" after a valid configuration is set for the next epoch to prevent erroneous IPFIX exports under a partial configuration.
- Telemetry operations that perform some aggregation (*e.g.*, counting distinct sources) often only need to export their results at the end of the epoch—due to MTU restrictions this may lead to several IPFIX data packets per operation (see label D1 in figure 6). As these packets are transmitted, new traffic arrives at the switch and potentially updates the results between the exported IPFIX packets.
- Some key counters and registers may not be included in the exported IPFIX data and must be read from the ASIC's memory space by the agent after results have been received by the collector (see label D2 in Figure 6). Again, these counters and registers may have been updated by the continuous flow of monitored traffic during the RTT between the switch hardware and the collector leading to discrepancies between the values read and the value returned in the IPFIX reports.

[4]We use the term "collector" (instead of "controller") to emphasize that this software only manages the monitoring capabilities of the switch. In contrast, a controller manages the forwarding behavior of a switch.

In our prototype, the collector is directly connected to the switch's management port and we observe the short RTT between them minimizes these potential errors. In larger systems where this RTT might be significant, we contend that the switch agent could listen for an interrupt or mirror IPFIX data packet from BroadScan and immediately "unplug" the traffic fed into the monitoring system to mitigate both of these potential discrepancies.

4.2 Broadscan

At the core of *Cedar* is BroadScan [1], a re-configurable hardware module in the switch ASIC, that performs telemetry operations at line rate. In contrast to P4-based systems, BroadScan is completely independent of the forwarding function of the ASIC allowing telemetry operations to be changed during runtime *without* any impact on other critical switch functions. BroadScan is centered around an independent hardware *flow table* and features several dedicated hardware processes to manage filtering, aggregating, and exporting monitoring information. BroadScan receives parsed information about each packet traversing the ASIC's main forwarding pipeline through a dedicated bus and evaluates three stages to generate useful telemetry reports: (i) filtering, (ii) aggregation, and (iii) export. We note that, while the current prototype of *Cedar* relies on particulars of the BroadScan hardware, *Cedar* could be implemented on a different ASIC if that ASIC supported similar features in terms of reconfigurability and hardware table management.

Filtering. matches particular bit patterns in packet headers allowing rules to be applied to specific slices of traffic (similar to the *flowsets* of ProgME [31]). Filters on single fields can be combined using logical conjunction and disjunction in more complex predicates allowing telemetry applications to drill down on relevant packets along several dimensions, *e.g.*, "only monitor traffic on TCP port 80 sent to destinations in the 192.0.2.0/24 or 198.51.100.0/24 subnets." Filtering is implemented by a dedicated TCAM match-action table inserted between the parser and the main ingress match-action tables, allowing the evaluation of complex predicates over the rich variety of information extracted by the parser. For example, BroadScan can filter packets by subnet, vlan, or TCP flag combinations.

Aggregation. is implemented by a flexible hash table and ALU operations, allowing to group packets along different dimensions and at different resolutions, as well as providing several different aggregation functions. For example, packets could be grouped according to source subnets to aggregate over the IP address space, according to packet length to detect large numbers of similar-sized packets, or by combinations of TCP, flags to measure the ratio between SYN and FIN packets. In addition to simple counting and summation, these ALUs can evaluate minimums, maximums, moving averages, and range-based functions such as histograms. The aggregation stage provides the following three key features to effectively manage the flow table and track relevant flows:

- *Hardware-based Learning*: new table entries are automatically "learned" by hardware—that is when a packet hashes to a table entry marked as invalid, the hardware automatically initializes counters for this entry and begins aggregation operations;
- *Aging out Rules*: table entries can be automatically aged out—an independent hardware process periodically decrements an "age" counter for each table entry and removes entries after they have received no packets for a certain period of time;
- *Detecting/Measuring Table Overflow*: the aggregation table has a fixed, hardware-dependent number of entries, but BroadScan maintains dedicated counters for packets which are missed due to a full table—allowing applications to detect table overflow and estimate the degree to which the table entries sample the total traffic.

Export. of the results of the aggregation stage can be performed in a periodic or event-driven fashion. These options enable *Cedar* to minimize the volume of transmitted telemetry data to the collector while accommodating the required export discipline for individual tasks. Periodic export is implemented by a clock which marks a set of aggregation table entries as ready for export at a fixed interval, the duration of which is controlled through a dedicated register. Event-driven export can be triggered when new entries are learned, existing entries are aged out, or aggregation counters pass certain thresholds. Either mechanism leads to the insertion of records, containing the counter values generated by the aggregation table, into a dedicated export FIFO. Finally, a hardware process coalesces these records into IPFIX data packets, appends the appropriate headers, and inserts the packets into the switch's data plane where they are routed to the collector. To the best of our knowledge, BroadScan is the first hardware telemetry system that generates and emits telemetry data packets directly in hardware. Currently, BroadScan only supports UDP transport of telemetry data from the data plane. Other transport protocols can be implemented by transferring the exported records first to the switch's CPU.

Limitations: A key limitation of BroadScan in comparison with P4-based programmable switches is that it currently features a single TCAM for deciding which data plane packets to consider. Telemetry operations installed in BroadScan select what packets they should apply to in the BroadScan TCAM table, however, since each incoming packet can only match a single TCAM entry, each packet can only be counted towards a single operation. This implies that concurrently executed operations in *Cedar* must apply to disjoint sets of

packets as determined by their filters. Future versions of BroadScan may feature multiple TCAM tables for filtering packets into multiple groups or at multiple vantage points in the switch's topology (*e.g.*, after the ingress pipeline or before the deparser).

4.3 Switch Agent

To control the BroadScan hardware module, we develop a software switch agent that runs on the switch's CPU, acting as an intermediary between the control functions of the collector and the hardware configuration. Our switch agent acts as a server, fulfilling requests for particular hardware configurations from the collector and returning hardware status. These requests are delivered as an intermediate representation that abstracts the hardware configuration space into a compact memory structure. In addition to updating the hardware configuration, the agent keeps track of the currently running operations and can perform sanity checking for compatibility and resource usage.

The agent uses the Broadcom SDK to install the requested configuration changes, however, our current use of BroadScan somewhat exceeds the functionality provided by the Broadcom SDK so parts of our implementation use direct table and register access. Moreover, there are several opportunities for improving the efficiency of hardware configuration through coalescing the configuration update operations required by *Cedar* and bypassing the layers of error checking and verification added by Broadcom SDK calls. We quantify the expected improvement of these methods in Appendix 8.2 and intend to work on this further as future work.

4.4 Remote Collector

To execute adaptive telemetry tasks, we develop a remote collector which receives the telemetry results emitted from BroadScan and responds by changing the monitoring configuration via the switch agent. As shown on the right side of figure 5, our collector consists of (i) an IPFIX parser that received raw IPFIX data stream from the switch and parses it into JSON, and (ii) a runtime that evaluates task flowcharts and coordinates with the switch agent to reconfigure BroadScan at the beginning of each epoch.

We adopt the open-source `ipfixcol2` [3] to parse the raw IPFIX data stream returned from the switch and to forwarded the parsed data as JSON to the runtime. The open modular architecture of `ipfixcol2` allows users of *Cedar* to easily insert a database for long-term storage of raw telemetry results or adopt another transport protocol such as TCP. We developed a custom plugin in `ipfixcol2` to strip padding data fields inserted by BroadScan and provided custom information element definitions for fields not found in the IANA standard [4].

At the beginning of each epoch, the runtime polls each scheduled flowchart node to build an install list of the operations to execute on the switch in that epoch. Nodes express these per-epoch operations to the runtime in a human-readable, high-level Switch Telemetry Query Language (STQL) which abstracts the details of hardware configuration in a stream processing-like paradigm [19]. The runtime compiles expressions submitted in STQL into install commands in an intermediate representation (IR) and sends a batch of install commands for all the operations selected for the particular epoch to the switch. After sending these install commands, the runtime polls the parser to wait for the receipt of telemetry results. Once available, the runtime gathers the results from the parser and distributes them back to the scheduled nodes which are responsible for interpreting the results and making transitions. The runtime evaluates these transitions, generating more active nodes, runs a scheduling algorithm (as described below) to generate the list of scheduled nodes and the epoch duration, and repeats the process.

4.5 Generic Flowchart Nodes

To illustrate the utility of telemetry flowcharts, we describe the implementation of three generic nodes, predicate count, distinct count, and heavy hitters. These nodes can be parameterized to complete a wide variety of common telemetry tasks such as DDoS or port scanning detection. For the implementation details of these nodes, see Appendix 8.1

Predicate Count. The generic predicate node simply counts the number of packets or byte that satisfy a given filter condition. For example, a predicate count node could be given the filter "(SrcIPv4 in 10.0.1.0/24 or SrcIPv4 in 10.0.3.0/24) and IPProtocol == 6" to count the number of IPv4 packets from two particular subnets. Our implementation uses a single operation and a single flow table entry to track the packet or byte counts.

Distinct Count. The distinct count node counts the number of distinct elements, as defined by a grouping expression, observed in in a filtered subset of traffic. For example, to count the number of distinct IPv4 sources sending to a particular subnet, one could give "SrcIPv4" as the grouping expression and a filter expression to only catch packets destined for the particular subnet. Our implementation uses a form of distinct sampling [18], leveraging the BroadScan table overflow counter.

Heavy Hitters. The heavy hitters node extracts heavy elements from traffic, as defined by a grouping expression and a definition of heaviness based on predicate or distinct counts. For example, the classic DDoS detection task can be implemented by grouping over destination addresses and defining heaviness in terms of the distinct count of sources. Our implementations uses the Multi-Resolution Tiling (MRT) algorithm proposed in ProgME [31].

5 EVALUATION

We evaluate the performance and accuracy of *Cedar* in three steps. We start by providing details of our setup including traffic traces and telemetry tasks in § 5.1. In § 5.2, we explore how resource utilization progresses over time, showing that tasks expressed as flowcharts only use resources when needed. Next, in § 5.3, we measure the resource usage for several example tasks showing that in *Cedar*, tasks execute a maximum of 100 to 300 operations per epoch. In § 5.4, we show how leveraging data plane resources allows *Cedar* to reduce load on the collector by up to four orders of magnitude. We show how the maximum and mean value of total required resources and reported tuples scale with the number of telemetry tasks in § 5.5. Finally, § 5.6 verifies that the implemented iterative tasks in *Cedar* achieve the expected accuracy of their corresponding algorithms (~10% error for distinct count and ~80% recall for heavy hitters).

5.1 Setup

Traces. All our evaluations use the CAIDA unsampled and anonymized Internet trace from 2019 [2]. In particular, we use the first five minutes of direction A of the NYC monitor containing ~138 million packets at an average rate of 612Kpps. Each second of this traffic has packets from approximately 32K sources to 42K destinations.

Example Tasks. To evaluate *Cedar*, we implemented four common telemetry tasks, namely DDoS, heavy hitters, new TCP connections, and port scan, using the generic flowchart nodes described in § 4.5. These tasks all use the generic heavy hitters node to detect different traffic patterns by changing how heaviness is defined. We leave implementation of other generic nodes such as frequency moment estimation as future work. Two of these tasks (in particular DDoS and Port Scan) additionally use the distinct count implementation to provide the measure of heaviness associated with each element in the search space. For each task, we iteratively refine the MRT threshold on the particular measure of heaviness until 10 to 15 heavy elements are detected in the traffic trace. The default epoch duration in our flowcharts is one second.

Setting. Our switch is a BroadScan-enabled BCM 57340 series System Verification Kit (SVK). The version of BroadScan in our SVK has a single 2048 entry match action table, 20 programmable ALU operators, and a flow table capacity of 32K entries with 36 bytes per entry for a total of ~1.3MB for aggregation state. Our switch agent runs directly on the SVK's CPU, an ARM Cortex A57 MPCore at 2Ghz with 4GB memory. Our collector and manager software runs on a server with an Intel Xeon Gold CPU at 2.3Ghz and 383GB memory. To drive test traffic through the switch at line rate, we replay the trace using `tcpreplay` [9] through a 40Gb Mellanox MT27700-family network card connected directly to the SVK's data plane. A separate 10Gb Intel X550T network card on the same server connects to the SVK's management interface to manage the *Cedar* control plane.

5.2 Flowchart Execution

We demonstrate how *Cedar* is able to conditionally allocate resources to a telemetry task *as the task's flowchart is executed* by recreating a realistic attack detection scenario and observing the system's behavior during attack detection. For attack traffic, we use a publicly available DNS reflection DDoS attack trace [27] that was captured from a real "Booter" service. This traffic features 281 seconds of reflected DNS traffic at a mean rate of ~45Kpps coming from ~940 sources each second. We mix this attack traffic into the CAIDA trace at a known offset and replay the resulting augmented trace through our SVK while running the DNS reflection flowchart (shown in Figure 2). We set the thresholds in the flowchart to values slightly larger than the maximums observed in the CAIDA trace.

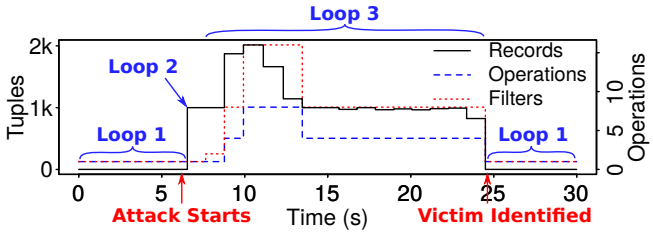
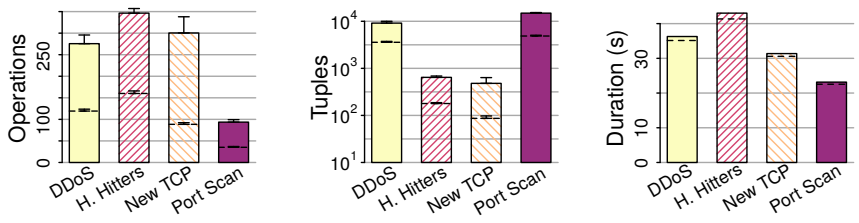


Figure 7: Resource usage of DNS reflection flowchart over time showing the number of records exported to the collector, operations and filters installed in BroadScan for each epoch during detection of a DNS reflection attack. (Loops are shown in Figure 2.)

Figure 7 shows how switch resource usage and report traffic progress as the flowchart detects the victim of the attack. Before the attack begins, the flowchart remains in loop 1 for six epochs, counting DNS packets in each epoch by using a single operation which also returns a single report in each epoch. After 6 seconds, the attack begins and the abnormally large number of DNS packets observed triggers the flowchart to move into loop 2, executing the distinct count node to measure the number of sources sending DNS traffic. The distinct count node executes an aggregation operation in the data plane to estimate the total number of DNS sources resulting in a spike in the number of records reported from BroadScan. After one epoch in loop 2, the distinct count node indicates an abnormally large number of sources sending DNS traffic indicating that a DDoS attack is underway. The flowchart responds by transitioning to the heavy hitters node to find the victim in loop 3. The heavy hitters node executes the MRT algorithm, iteratively zooming in on destination subnets that receive from large numbers of sources, until the victim /32 address is found after 17 epochs in loop 3. During execution of this algorithm, the heavy hitters node uses up to



(a) Operations **(b) Exported reports** **(c) Detection duration**

Figure 8: Per epoch resource usage of example tasks. Bars show max while dotted lines show average over the course of event detection. Error bars show standard deviation over 10 trials on different trace segements.

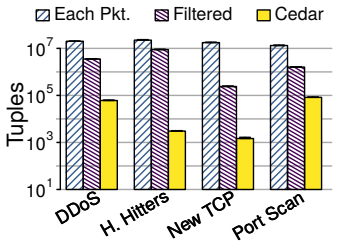


Figure 9: Total load on collector over single detection cycle.

8 parallel aggregation operations (and up to 16 filters in the TCAM table) to monitor disjoint subnets resulting in larger volumes of returned records. After around six iterations of the MRT algorithm, the only subnet still under consideration is the subnet containing the victim, causing the number of parallel operations per epoch to reduce to 4, though the report volume remains relatively high (around 1K tuples per epoch) until the flowchart returns to loop 1.

Summary: This example demonstrates that *Cedar* dynamically controls the allocation of switch resources to tasks based on observed traffic conditions and task semantics. To the best of our knowledge, prior data plane telemetry efforts rely on static mapping of operations to switch resources at deployment time that significantly limits the utilization of switch resources. In *Cedar*, detection tasks only *utilize switch resources for the required operations in each epoch.* For example, the DNS reflection flowchart requires a single operation for a triggering condition and only moves to other parts of the flowcharts if more in-depth detection operations are justified.

5.3 Task Resource Usage

To understand how the resource usage (discussed in § 5.2) varies across different tasks, we run four example tasks and collect the maximum and average operations and exported tuples per epoch from the running system. By using these metrics, we illustrate the average-case resource usages of the tasks as well as the extent of any spikes (*e.g.*, as seen in Figure 7). Rather than constructing attack traffic for each task, we adopt the approach of prior work [19, 21] and tune the thresholds of these tasks to find corresponding patterns already present in the CAIDA trace. A single exception to this is the new TCP task—since we did not observe any hosts fitting this traffic pattern in the five minute trace considered here, we injected a small number of TCP SYN packets (less than 0.5% of the total volume of SYN packets in the original trace) for the new TCP task to detect.

Figures 8a and 8b summarize the resource usages and report volumes for the selected tasks showing that Cedar *can execute different telemetry tasks using less than 200 operations in the data plane per epoch on average.* Tasks that use distinct counts as the heaviness measure (*e.g.*, DDoS, Port Scan) require fewer operations, but generate more report traffic than tasks which use packet (new TCP) or byte (heavy hitters) counts. This is due to the fact that our distinct count implementation must export several records to estimate the number of distinct elements in each monitored subnet whereas the packet and byte count implementations use a single record for each subnet. Figure 8c presents the max and mean detection latency (in seconds or epochs) for individual tasks. This result reveals that our target tasks can be detected in 22 to 42 seconds.

Summary: The main observation in Figure 8 is that the gap between maximum usages and average usages is large for all tasks considered (at least 2×). This confirms the finding of § 5.2—that tasks typically only use their maximum amount of resources for a small fraction of the time—for a wider range of different tasks. As discussed in § 3, reconfigurable telemetry systems like *Cedar* are uniquely capable of leveraging this observation to improve resource utilization by changing allocations over time in response to the usage patterns shown here. We plan to address the algorithmic challenges of formulating scheduling strategies to automatically leverage this characteristic in our future work.

5.4 Reporting Schemes

To understand how the volume of reported information in *Cedar* compares to alternative methods which forward fine-granularity reports to software for reconfigurable processing [28, 32], we measure the total number of tuples returned to the collector in a single detection round under three different reporting methods.

- *Each Packet:* A tuple is returned to the collector for each network packet as in systems like NetQRE [32].

- *Filtered:* The switch ASIC is only used to filter which packets should generate report tuples to forward to the collector as in systems like Everflow [34].
- Cedar*:* The switch ASIC is used for both filtering, reduction, and report generation at the end of each epoch as described in § 4.

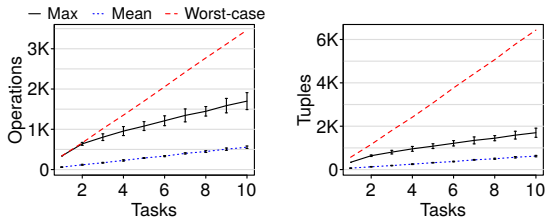
Figure 9 shows that Cedar *is able to reduce the number of tuples sent to the collector by up to four orders of magnitude*, allowing for savings comparable to those achieved by fixed "compile-and-deploy" telemetry systems [19] while offering the greater flexibility afforded by a fully reconfigurable model. We note that there is a roughly two order of magnitude difference in the collector load between tasks like DDoS, which find heavy elements based on distinct counts, and tasks like new TCP, which find heavy elements based on predicated counts. This is due to the fact that our distinct count method must send multiple records to estimate the number of distinct elements for a particular slice of traffic, whereas the predicated count method sends a single record per traffic slice.

Summary: A key motivation for transferring telemetry processing operations into the data plane is the reduction in the volume of reports which must be transported to and processed by a remote collector. Figure 9 shows that, even with a single telemetry stage in the data plane, by using a reconfigurable approach this reduction can still be significant compared to methods which require exporting a tuple for each packet.

5.5 Resource Usage with Multiple Tasks

We evaluate the resource requirements for running multiple concurrent telemetry tasks in *Cedar* using the following methodology. We consider the temporal evolution of required resources per epoch for (similar to Figure 7) for ten different instances of the heavy hitters detection. We repeat the resource usage of each instance by adding silent gaps that are generated by a Poisson process with a mean rate of 2 second. Using this technique, we generate independent 1K-epoch-long emulated resource usage timeseries for each task. We then sum up the resource usage in each epoch across timeseries of x different tasks and identify the mean and max values across all epochs of the resulting timeseries. These values offer representative mean and max resource usage across for these x tasks as they co-occur with different temporal offset.

Figure 10a shows the resulting mean and max resource requirements per epoch for x concurrent tasks over 10 independent. We also show the worst-case resource requirement for x task as the sum of the maximum resource usage across all tasks. Figure 10b shows the same metrics for the number of report tuples returned to the collector. For both operations and reported tuples, we observe the worst-case usages increase linearly with the number of tasks. However, we also observe that in our emulated scenario, the maximum and average usages increase sublinearly, confirming that temporal partition of operations between tasks potentially improves scalability in the number of tasks.



(a) Operations **(b) Exported reports**

Figure 10: Resource usage when running multiple tasks showing worst case and the observed maximum and mean values per epoch. Lines show means, error bars show std. deviations over 10 trials.

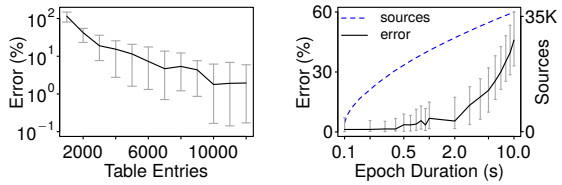
Summary: Due to the large variation in the temporal pattern of task resource usage, combining multiple tasks in *Cedar* can effectively share the resources of a fixed single stage among a number of independent telemetry tasks. In particular, the number of operations required in the data plane and the number of reported tuples are sublinear in the number of tasks running concurrently, indicating the advantages of temporal partitioning for scalability in the number of concurrent tasks.

5.6 Generic Node Performance & Accuracy

Having shown that *Cedar* is able to evaluate dynamic, iterative telemetry tasks and characterizing the resource usage of tasks in *Cedar*, we now turn to evaluating the accuracy of our generic flowchart node implementations. As discussed in § 2, our goal is not to compete with or to highlight the missing aspects of state-of-the-art solutions to these measurement problems (e.g., bitmaps [17] or count-min sketches [15]). Instead, we seek to highlight the efficacy of reconfigurable switch hardware used in *Cedar* by offering initial serviceable tools to best leverage its advantages. This section only discussed the distinct count node evaluation—for an evaluation of the other nodes used in *Cedar*, see Appendix 8.2.

Distinct Count: We evaluate the accuracy of our distinct count estimation node along three dimensions: the number of hardware table entries allocated to the node, the epoch duration, and the duration of the gap between epochs. In this experiment, we use a 30% uniform sample of flows from the CAIDA trace to reduce the number of sources to approximately 10K per second. This allows evaluation of scenarios where the number of table entries is less than the total number of elements (sources) as well as scenarios where the number of table entries is greater than the total number

of elements. We report median error from 120 trials with Error = $\max\left(\frac{n}{\hat{n}}, \frac{\hat{n}}{n}\right) - 1$ where n is the ground truth (calculated offline) and $\hat{n}$ is the value returned by *Cedar*.



(a) Fixed 1 sec epochs. **(b) Fixed 10K table size.**

Figure 11: Performance of Distinct Count Node showing median and error bars at 5% and 95% quantiles.

Figure 11a shows that the error of the distinct count node rapidly decreases with the size of flow table until it reaches ~2% with 10K entries after which it levels off. Figure 11b presents the effect of epoch duration on the accuracy of the distinct count node using flow table with 5K entries. Note that the ground truth number of flows is adjusted with epoch duration to calculate the error. The lowest epoch duration is 0.1 second as imposed by the BroadScan export process (see Figure 1b). This result reflects the trade off described in Figure 3: short epochs observe significant variation while long epochs catch too many elements for the fixed memory to form an accuracy estimate.

Due to space constraints, we show the performance and accuracy results for Heavy Hitters in Appendix 8.2.

Summary: The results of this section confirm that the generic flowchart nodes developed in *Cedar*, are capable of leveraging the reconfigurable telemetry system to produce accurate results (~10% error for distinct count and near perfect precision with ~80% recall for heavy hitters). More over, there is a trade off for both node types in that allocating more table entries improves performance while varying the epoch duration exposes a sweet spot in result variability and search time. The accuracy of our heavy hitters algorithm is comparable to prior software evaluations of the underlying MRT algorithm [22, 31], though in *Cedar* we leverage a reconfigurable data plane to reduce the overheads of exported results.

6 RELATED WORK

We present a brief overview of related efforts in network telemetry in addition to those discussed in § 1 and § 3.

Adaptive CPU-based Telemetry. Our work is closest in spirit to adaptive monitoring efforts such as ProgME [31]. These efforts iteratively update filtering and reduction operations to "zoom in" on particular features of interest. More recently, DREAM [22] and SCREAM [23] propose integrating adaptive monitoring in switch-based systems. However, these efforts rely on CPU-based functional simulations or virtual switches for evaluation and it is unclear if they could be deployed on actual switch hardware due to the overheads of installing hardware programs.

Generic Data Plane Telemetry. An alternative approach to the "compile-and-deploy" model, is to develop a single, unified telemetry summary using sketches that can be adapted to multiple (possibly unforeseen) tasks. Sketches combine constant-time updates and counting to estimate useful traffic features, such as heavy hitters [15], entropy [33], per-flow delay [26], or micro-bursts [14]. To increase deployability of sketch-based solutions, OpenSketch [30] provides a framework for composing and automatically tuning a number of sketch primitives to produce useful traffic features. Universal sketching [21] builds a single sketch which can be used to estimate multiple, possibly unforeseen, useful summary statistics. However, captured features (*e.g.*, source address, destination address, source-destination pairs) by a sketch are fixed at deployment time and there are limits to the types of queries that can by satisfied by the resulting universal sketch. `*flow` [28] adopts a method using grouped packet vectors to leverage the generic aggregation capabilities of a PISA switch while performing the specific aggregation required by a particular query in software. However, the reduction granularity is fixed at deployment time and the resulting stream of per-packet information poses post-processing challenges.

7 CONCLUSION & FUTURE WORK

The compile-and-deploy model of modern data plane systems leads to inefficient utilization of switch resources, hinders their scalability in the face of an increase in the number and complexity of telemetry tasks, and lacks agility. To address these limitations, we present the design, implementation, and evaluation of *Cedar*: a first-of-its-kind reconfigurable data plane telemetry system. At the core of *Cedar* is a recent innovation in switch hardware called Broadscan which enables the agile (re)configuration of the data plane at *runtime*. This capability provides a unique opportunity to partition telemetry tasks temporally and facilitates efficient switch resource utilization, better scalability with the number of telemetry tasks, and reduced load on the remote collector.

In future work, we plan to extend this effort in the following directions. First, we will incorporate a resource scheduling scheme in *Cedar* to avoid overloaded epochs where the required resources for concurrent operations exceed switch memory. Second, we are working on deploying *Cedar* over campus and enterprise networks to examine *Cedar*'s performance and agility in action. Third, we intend to develop a distributed telemetry system where multiple *Cedar* nodes cooperatively determine proper operations at individual nodes to detect a network-wide event reliably.

REFERENCES

[1] [n. d.]. BCM56870 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series. ([n. d.]).

[2] [n. d.]. The CAIDA UCSD Anonymized Internet Traces Dataset - 2019. https://www.caida.org/data/monitors/passive-equinix-nyc.xml. ([n. d.]).

[3] [n. d.]. CESNET/ipfixcol2: High-performance NetFlow v5/v9 and IPFIX collector. https://github.com/CESNET/ipfixcol2. ([n. d.]).

[4] [n. d.]. IP Flow Information Export (IPFIX) Entities. https://www.iana.org/assignments/ipfix/ipfix.xhtml. ([n. d.]).

[5] [n. d.]. Network Infrastructure High Availability with SONiC Warm Reboot. https://www.barefootnetworks.com/blog/bringing-ocp-vision-reality-programmable-dataplanes-progress-report/. ([n. d.]).

[6] [n. d.]. P4 with the Netronome Server Networking Platform. https://www.netronome.com/blog/p4-with-the-netronome-server-networking-platform/. ([n. d.]).

[7] [n. d.]. Product Brief Tofino Page. https://barefootnetworks.com/products/brief-tofino/. ([n. d.]).

[8] [n. d.]. Software Development Kit Logical Table. https://www.broadcom.com/products/ethernet-connectivity/software/sdklt. ([n. d.]).

[9] [n. d.]. Tcpreplay - pcap editing and replaying utilities. https://tcpreplay.appneta.com/. ([n. d.]).

[10] 2018. $P4_{16}$ Portable Switch Architecture (PSA): Version 1.1. https://p4.org/p4-spec/docs/PSA-v1.1.0.html. (2018).

[11] Kevin Borders, Jonathan Springer, and Matthew Burnside. 2012. Chimera: A declarative language for streaming network traffic analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 365–379.

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[14] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the microburst culprits with snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*. ACM, 22–28.

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

[16] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 647–651.

[17] Cristian Estan, George Varghese, and Mike Fisk. 2003. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. 153–166.

[18] Phillip B Gibbons. 2001. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, Vol. 1. 541–550.

[19] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 357–371.

[20] Faisal Khan, Nicholas Hosein, Soheil Ghiasi, Chen-Nee Chuah, and Puneet Sharma. 2013. Streaming solutions for fine-grained network traffic measurements and analysis. *IEEE/ACM transactions on networking* 22, 2 (2013), 377–390.

[21] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 101–114.

[22] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2014. DREAM: dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 419–430.

[23] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2015. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. ACM, 14.

[24] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 85–98.

[25] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo hashing. In *European Symposium on Algorithms*. Springer, 121–133.

[26] Josep Sanjuàs-Cuxart, Pere Barlet-Ros, Nick Duffield, and Ramana Rao Kompella. 2011. Sketching the delay: tracking temporally uncorrelated flow-level latencies. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 483–498.

[27] J.J. Santanna, R. van Rijswijk-Deij, R. Hofstede, A. Sperotto, M. Wierbosch, L. Zambenedetti Granville, and A. Pras. 2015. Booters - An analysis of DDoS-as-a-service attacks. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 243–251. https://doi.org/10.1109/INM.2015.7140298

[28] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow. In *2018 USENIX Annual Technical Conference (USENIXATC 18)*. 823–835.

[29] Q. Wu, Strassner J., Farrel A., and Zhang L. 2016. Network Telemetry and Big Data Analysis. https://tools.ietf.org/html/draft-wu-t2trg-network-telemetry-00. (2016).

[30] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 29–42.

[31] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. 2011. ProgME: towards programmable network measurement. *IEEE/ACM Transactions on Networking (TON)* 19, 1 (2011), 115–128.

[32] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative network monitoring with NetQRE. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 99–112.

[33] Haiquan Chuck Zhao, Ashwin Lall, Mitsunori Ogihara, Oliver Spatscheck, Jia Wang, and Jun Xu. 2007. A data streaming algorithm for estimating entropies of OD flows. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 279–290.

[34] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. 2015. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 479–491.

8 APPENDIX

8.1 Implementation of Generic Nodes

Predicate Count: Measuring the volume or rate of particular subsets of traffic is a foundational operation in most event detection and management tasks. In *Cedar*, the predicated count node simply uses BroadScan to count the number of packets and bytes which satisfy a given filter condition, allowing volume and rate estimations when combined with knowledge of the epoch duration. This node is given a filter as a predicate over the supported header fields, potentially including masking, conjunctions, and disjunctions, which is passed directly to the runtime system as an STQL expression. For example, a count node could be given the filter "(SrcIPv4 in 10.0.1.0/24 or SrcIPv4 in 10.0.3.0/24) and IPProtocol == 6" to count the number of IPv4 packets from two particular subnets. At the end of the epoch, the resulting packet and byte counts are stored in the global state allowing subsequent nodes in the flowchart to react to the observed results.

Distinct Count: Many telemetry operations require counting or estimating the total number of distinct elements observed in a given subset of traffic. In *Cedar*, we implement a generic distinct count node which estimates the number of distinct elements as defined by grouping packets along one or more fields. This node is given a filter, in the same form as the predicate node, and an expression which defines how to group packets, expressed as a list of field names with possible masks. For example, to count the number of distinct IPv4 sources in a particular traffic subset, one could give "SrcIPv4" as the grouping expression. Again, the resulting distinct count is written into the global state for use by subsequent nodes in the flowchart.

Our distinct count node uses the BroadScan hardware to estimate the number of distinct elements using less table entries than elements, adopting a technique similar to distinct sampling [18]. Each distinct count node is allocated a certain number of table entries when it is scheduled and each table entry tracks a particular distinct element. If the total number of distinct elements in the current traffic is less than the number of table entries allocated, the node reports the exact number of distinct elements. If the total number of distinct elements exceeds the number of table entries allocated, the node uses the value of the exceeded counter returned from BroadScan to estimate the total number of distinct elements based on the distribution of the returned table entries. The intuition for this estimation is that the set of elements sampled in the aggregation table and the set of elements that exceeded the aggregation table should have similar distributions in terms of the number of packets per element. Specifically, if the total number of elements sampled in the aggregation table is q_s, the density of packets per element is δ_s, and the number of packets that exceeded the aggregation table is p_e, our count distinct node estimates the total number of elements $\hat{n}$ as

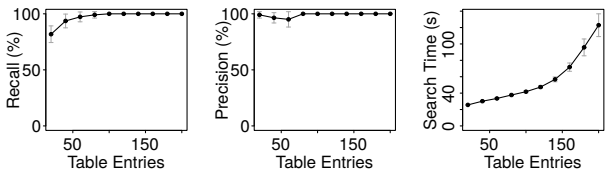
$$\hat{n} = q_s + p_e/\delta.$$

Due to the skewed nature of packet distributions in network traffic, we find that the median provides the most useful estimate of the density δ. Our evaluation section provides empirical evaluation of the performance of this estimator though we leave rigorous mathematical analysis to future work.

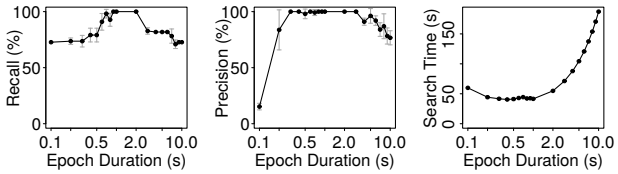
Heavy Hitters: Another common task for telemetry systems is finding heavy hitters: elements which represent more than a given fraction of the total number of elements of that type in the given traffic. In *Cedar*, we implement a generic heavy hitters node that finds heavy elements based on arbitrary counting strategies for a particular subset of traffic flowing through the switch. This generic node can be applied towards traditional heavy hitter tasks, such as finding top talkers, or security event detection tasks, such as DDoS victim detection by changing the count to interpret as "heavy." Arbitrary counting implementations are dynamically loaded through a generic interface which accepts a subset of traffic specified as an STQL filter expressions and returns an (estimated) count to the heavy hitters node.

To implement heavy hitter detection, we use the Multi-Resolution Tiling (MRT) algorithm originally proposed by Yuan et al. [31] and improved through several enhancements by Khan et al. [20]. MRT is an adaptive algorithm which uses sequential hypothesis tests to iteratively zoom in on potential heavy hitters. The simple yet powerful fact that if the combined count for a group of elements does not exceed the threshold, then the counts for none of the individual elements in the group drives the zooming process—in each iteration if the aggregated count for a group exceeds the threshold, MRT zooms in by partitioning the group and repeating the process on the resulting subgroups in the next epoch, otherwise the group is no longer considered. We also implement the flow momentum enhancement proposed by Khan et al. [20], which essentially delays the decision to drop a group from consideration based on the mean counts used in arriving at that group. In practise, this enhancement is required to deal with the often bursty quality of network traffic.

The inputs to this algorithm are the heavy hitter search space (*e.g.*, source IP addresses) and a threshold on the counted value (*e.g.*, 5% of the link bandwidth in bits per second). After several iterative stages, the algorithm returns a (potentially empty) list of elements from the search space which meet or exceed the given threshold (*e.g.*, sources contributing to more than 5% of the total link bandwidth). We implement the division into subgroups by adding a fork method to our framework. Mirroring the classic `fork` call in Unix, this method



(a) Effect of table size (per operation) with fixed epoch of 1 second.



(b) Effect of epoch duration with table size based on threshold.

Figure H.1: Performance of heavy hitters node showing mean and standard deviation of recall, precision, and search time.

creates a number of copies of the current node which can be independently parameterized and evaluated in the next epoch. Child nodes which find heavy elements append an identifier of the element (*e.g.*, the source ip address) into a list in the global state and make a transition indicating successful location of a heavy element, while child nodes that drop a particular group from consideration, as per the MRT algorithm, make a transition indicating no heavy element has been found. This allows designers of flowcharts to explicitly deal with located heavy elements by, *e.g.*, transitioning to a reporting node when heavy elements are located.

8.2 Extended Evaluation

Task Details. Table H.1 shows how the example tasks we implemented use the generic heavy hitters node with different search spaces and definitions of heaviness.

Performance and Accuracy of Generic Nodes: Heavy Hitters We investigate the effect of flow table size and epoch duration on the accuracy of our heavy hitter detection technique (*i.e.*, MRT algorithm [31]) in detecting the top-10 destination addresses that receive connections from the largest number of source addresses (the core of the DDoS victim detection task). To evaluate this task, we augment the CAIDA trace described in Section 5.1 with ten selected heavy flows from the (busier) alternate direction. In this way, the added heavy flows follow realistic traffic patterns, similar to the lighter flows from direction A of the trace, while accounting for less than 3% of the total packets traversing the switch. For these experiments we report (i) recall as the fraction of ground-truth top-10 destinations found by *Cedar*, (ii) precision as the fraction of destinations reported by *Cedar* that are actually in the top-10, and (iii) detection latency as the time between starting the flowchart node and detecting all heavy hitters. We set the heaviness threshold at 250 sources per destination based on analysis of the non-heavy flows making up the background traffic and vary the number of table entries between 20 and 250.

The plots in Figure H.1a present the mean recall, precision, and detection latency across 10 runs of the heavy hitters node with 1 second epochs as a function of the number of table entries allocated to each primitive counting operation. While the distinct count operations early in the search process are likely see far more sources (due to the fact that they look at large subnets rather than individual destinations) and hence report in-accurate results, we note these operations are only needed to guide the search and the particular distinct count accuracy does not matter until the counts approach the threshold of 250 sources. These results show that our heavy hitters node effectively detects the 10-top heavy flows with ~80% recall and ~100% precision. We note that higher precision is an expected result of using the MRT algorithm which does not report a heavy element unless that element passes the specified threshold. Lower recall is also expected due to the fact that, during the search, the MRT algorithm dismisses entire subnets based on a single operation, potentially missing heavy elements. Finally, we note that latency increases with table size because less primitive counting operations can fit in each epoch. Unlike for nodes which execute a single operation (*e.g.*, distinct count), when nodes must execute multiple operations, changing the number of table entries exposes a tradeoff between recall and detection latency.

The plots in Figure H.1b present the mean recall, precision, and detection latency across 10 runs for the heavy hitter node as a function of epoch duration. As in figure 11b, our choice of epoch durations are limited by hardware constraints (in the minimum) and the increased accumulation of elements (in the maximum). To deal with the fact that different epoch durations are expected to capture different numbers of distinct elements, we scale the threshold based on the total number of distinct sources observed under each epoch duration leading to a range of thresholds between 64 and 622 elements. We also scale the number of table entries per operation as 1/2 of the threshold leading to a range of 32 through 311 entries.

Since the heavy hitters node executes in multiple epochs, increasing the epoch duration has a clear effect on the time taken to discover the heavy elements. We also note that accuracy (recall and precision) becomes less predictable with short epochs. This is likely due to the increased noisiness of the distinct count results under shorted epochs which

Task	Search Space	Heaviness	Max Ops	Mean Ops	Max Reports	Mean Reports
DDoS [30]	dests.	distinct sources	275.6	119.3	9148.7	3578.5
Heavy Hitters	sources	packets	346.4	160.1	643.4	178.9
New TCP [32]	dests.	SYN packets	300.4	88.5	479.7	86.6
Port Scan [19]	sources	distinct ports	93.6	35.2	14899.7	4862.7

Table H.1: Example tasks implemented in *Cedar* showing maximum and mean operations and reports per epoch. Results averaged over 10 trials.

causes the MRT algorithm to wrongly zoom in on or discard a subnet.

Effect of Number of Operations on Reconfiguration Time. To illustrate the ability of *Cedar* to quickly reconfigure telemetry operations, we measure the reconfiguration time as the time taken between submitting a configuration to the switch and getting confirmation of that configuration's installation (from the vantage point of the manager). We also measure five components of this total reconfiguration time as follows:

- *compile,* the time taken to compile the high-level STQL code emitted by flowchart nodes to the low-level Intermediate Representation (IR) of the hardware configuration;
- *transmit install,* the time taken to transmit the IR install configuration instruction from the manager to the switch;
- *install,* the time taken on the switch to configure hardware registers and memories for the requested configuration;
- *transmit remove,* the time taken to transmit the IR remove configuration instruction from the the manage to the switch; and
- *remove,* the time taken on the switch to reset the configured hardware registers and memories.

Figure H.2 presents the total re-configuration time (and its breakdown across different components) as a function of the number of concurrent, randomly-generated, single-filter operations. We observe that the installation time linearly increases with the number of operations. This result is consistent with prior work in terms of hardware configuration overheads [19] and the linear effect of the number of operations [22]. However, unlike prior work [22], the update time in *Cedar* is fast enough for us to evaluate task performance in an actual hardware prototype. We note that this result depends in part on our switch agent's reliance on the Broadcom SDK and that lower-level programming interfaces are likely to be fast, though trading off portability between different switch models (see section 4).

We also examined how the complexity of a single query, in terms of the number of filter table entries it requires, affects the installation time. Our result shows that the effect of the number of filters for individual operations has a negligible impact (less than 6 ms/operation with a similar standard deviation) on the total reconfiguration time.

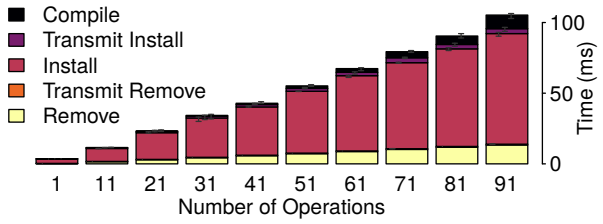


Figure H.2: Reconfiguration time as a function of the number of operations. Error bars show 25% and 75% quantiles.

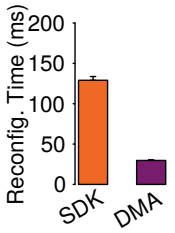


Figure H.3: Reconfiguration time for 128 operations, iteratively through the SDK (left) and in DMA-accelerated batches (right).

Improving Reconfiguration Time. To illustrate the ability of *Cedar* to quickly reconfigure telemetry operations, we measure the reconfiguration time as the time taken between submitting a configuration to the switch and getting confirmation of that configuration's installation (from the vantage point of the manager). We observe that installing a single operation takes roughly one millisecond and the installation time grows linearly with the number of operations, a result consistent with prior efforts [19, 22]. However, unlike prior work [19], the update time in *Cedar* includes changing the fundamental operations executed instead of just the filter and reduce granularities. *Cedar* is able to overcome the prohibitively long hardware update times encountered in prior

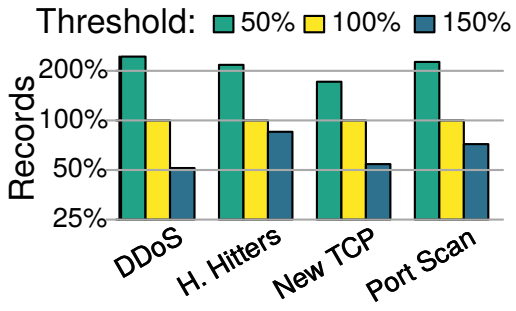


Figure H.4: Effect of adjusting detection threshold on collector load.

adaptive telemetry systems [22], enabling temporal partitioning of possibly unrelated operations in consecutive epochs as described in section 3.

To better understand the update time bottlenecks, we instrument *Cedar* to also report the time spent transporting configuration data between switch agent and collector and the time spent by the switch agent configuring hardware. As shown in Figure H.3, we observe that the configuration time is dominated by the switch agent configuring hardware. This result is a product of the programming interface exposed by the Broadcom SDK which forces us to write configuration operation by operation and performs a wide range of verification checks. However, the BroadScan configuration memories may also be updated in one shot through a DMA-accelerated interface. To understand the potential savings of this acceleration, we also timed writing the entire BroadScan configuration through DMA batchs. As shown on the right in Figure H.3, and confirmed by other experiments at Broadcom, this acceleration allows writing the entire configuration in ~30 ms, a nearly 4× improvement over the 131 ms time reported in prior work [19].

We also examined how the complexity of a single query, in terms of the number of filter table entries it requires, affects the installation time. Our result shows that the effect of the number of filters for individual operations has a negligible impact (less than 6 ms/operation) on the total reconfiguration time.

Sensitivity of Resource Usage to Threshold. Due to the iterative nature of our generic heavy hitters node, varying the threshold may have an effect on the volume of reported traffic—higher thresholds tend to drop regions of the search space earlier than lower threshold leading to less records exported. To understand this effect, we vary each of the thresholds fixed above by ±50% and observe the effect on the number of records returned. As shown in figure H.4, the number of records returned varies by a factor of 2 for most applications.

8.3 ASIC in the Control Loop

Telemetry systems that use switch ASICs must deal with the issue of collecting results generated in hardware typically by exporting this data to a centralized server. While prior efforts typically require the switch's CPU to read counter values and send the results as reports to the central collector, systems built on BroadScan can leverage the hardware's ability to directly inject telemetry report packets into the data plane, bypassing the switch CPU. To illustrate the potential savings of this hardware export method, we profiled the maximum rate at which our SVK's CPU could read telemetry reports from the BroadScan hardware and forward it to our collector, finding that the CPU could only achieve around 8K reports per second. The BroadScan hardware, on the other hand, easily scaled to over 20K reports per second. Admittedly, our SVK ASIC is controlled by a low-power ARM Cortex A57 MPCore and a more powerful onboard processor would likely increase this rate. Nonetheless, we argue that the ability of the hardware to directly export telemetry reports can significantly improve the efficiency of record export while reducing the load on the switch's CPU.