IRIS: An Interactive Programmable Fine-Grained Network Telemetry System

by

Mana Atarod

A thesis accepted and approved in partial fulfillment of the

requirements for the degree of

Master of Science

in Computer Science

Dissertation Committee:

Prof. Reza Rejaie, Co-Chair

Prof. Ram Durairajan, Co-Chair

Dr. Christopher Misa, Core Member

University of Oregon

Winter 2025

THESIS ABSTRACT

Mana Atarod

Master of Science in Computer Science

Title: IRIS: An Interactive Programmable Fine-Grained Network Telemetry System

Network telemetry systems provide insights into network traffic that are essential for maintaining performance and security. While significant progress has been made in the area of telemetry system design and implementation, existing solutions focus on specific design aspects rather than providing a comprehensive end-to-end approach and require extensive expertise in data plane technology and programming. In response to these issues, this work presents IRIS: an interactive, end-to-end, and programmable fine-grained telemetry system built on the BroadScan module found in ASICs from Broadcom, Inc., which are deployed in a wide range of present-day networks. IRIS provides a flexible, high-level interface that enables users to define queries using a simple Python interface. It then translates the user-defined queries into hardware-level configurations for BroadScan, and efficiently retrieves the generated results from it. This work provides an overview of IRIS's architecture and evaluates its ability to utilize BroadScan's flow table at maximum capacity, produce time-based and loss-based statistics, and implement example use cases based on real-world network telemetry tasks.

ACKNOWLEDGEMENTS

This section is dedicated to all the people who have supported and guided me throughout my journey as a graduate student.

I would like to sincerely thank my advisors and committee members, Prof. Reza Rejaie and Prof. Ram Durairajan, for believing in me and giving me the opportunity to start my graduate career by joining the Oregon Network Research Group (ONRG). Their unwavering support and guidance were essential in bringing this journey to completion. I am deeply grateful to Dr. Chris Misa, my mentor and committee member, whose constant help, support, patience, and wisdom guided me every step of the way throughout my entire journey as a graduate student. Not only did his mentorship help me grow in countless ways, but he also laid the foundation for IRIS and implemented the initial version of it, the expansion of which made this thesis possible. I would also like to thank Prof. Suyash Gupta, faculty member of the ONRG lab, for his continued support during the process of preparing this thesis.

I am thankful to Shahram Davari and the Broadcom Inc. team for their support throughout the completion of this thesis.

I would also like to extend my thanks to the graduate members of the ONRG lab—Emad Taghiye, Nima Nikkhah, and River Bartz—who I had the pleasure of collaborating with and learning from. I am also grateful to the previous generations of ONRG students whose work laid the foundation for this thesis, and the future generation who will be carrying on the work.

Lastly, I want to express my deepest gratitude to my family and friends, who were always there for me and provided unwavering support throughout this entire journey.

This work is supported by the National Science Foundation through CNS 2212590 and OAC 2126281. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of NSF.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 11

II. BACKGROUND 15

III. IRIS 19

3.1. IRIS Architecture 19

3.1.1. BroadScan 20

3.1.2. Agent 22

3.1.3. Remote Server 22

3.2. Python Interface 24

3.2.1. Query Definition 25

3.2.2. Visualization 27

3.3. Controller 28

3.3.1. Running Queries 28

3.3.2. Updating Queries 30

3.3.3. Tuning Queries 30

3.3.4. Stopping Queries and Clearing Switch Configurations 35

3.4. Collector 36

3.5. Adding New Capabilities 38

3.6. Available Switch Layout 39

3.7. Required System Configurations 41

IV. EVALUATION 43

4.1. BroadScan Flow Table Capacity 43

4.2. Collector Performance Analysis 44

Chapter Page

4.2.1. IPFIX Record Collection Time 46
4.2.2. IPFIX Record Loss Percentage 47
4.2.3. IPFIX Record Parsing Time 48
4.2.4. IPFIX Record Post-Processing Time 50
4.3. Example Use Cases 51
4.3.1. Tuned Byte Count per Source and Destination IPs 51
4.3.2. Top Destination Host by Prefix Zooming 54
4.3.3. Anomaly Detection 55
4.3.4. TTL Histogram 57
V. CONCLUSION 60
5.1. Future Work 60
REFERENCES CITED 61

LIST OF FIGURES

Figure Page

1. High-Level Overview of ASICs Supporting P4, Such as Tofino Switches . . . 16
2. High-Level Overview of the BroadScan Module . . . 17
3. IRIS Architecture . . . 19
4. Remote Server Class Diagram . . . 24
5. Translation of a Query to its JSON Representation . . . 29
6. Quick Tuning FSM . . . 32
7. Continuous Tuning FSM . . . 34
8. IRIS Collector . . . 38
9. Border Deployment of IRIS . . . 40
10. Lab Deployment of IRIS . . . 41
11. Flow Byte Count and Packet Count Analysis . . . 45
12. Lab Switch IPFIX Record Collection Time Based on Record Count . . . 47
13. Border Switch IPFIX Record Collection Time Based on Record Count . . . 47
14. Border Switch IPFIX Record Loss Percentage Per Epoch . . . 48
15. Border Switch IPFIX Record Loss Percentage Based on Record Count . . . 48
16. Lab Switch IPFIX Record Parsing Time Based on Record Count . . . 49
17. Border Switch IPFIX Record Parsing Time Based on Record Count . . . 49
18. Lab Switch IPFIX Record Post-processing Time Based on Record Count . . . 50

Figure Page

19. Border Switch IPFIX Record Post-processing Time Based on Record Count 50

20. Query Tuning Progress 52

21. Time Series of Record Count for the Executed Tuned Query 52

22. Query Tuning Progress on the Border Switch 53

23. Time Series of Record Count for the Executed Tuned Query on the Border Switch 53

24. Lab Switch Anomaly Detection 57

25. Border Switch Anomaly Detection 58

26. Border Switch TTL Histogram 59

LIST OF TABLES

Table Page

1. Top Destination Prefix Based on Byte Count per Epoch with the Zooming Strategy 55

CHAPTER I

INTRODUCTION

Network operators often need to monitor the state of networks to analyze performance, identify potential issues, detect attacks, or understand user behavior. This process involves constant and near real-time measurement and analysis of network components and events [6]. To this end, they deploy network telemetry systems, which are technologies designed to provide insights into networks and enable efficient, automated network management [17]. A network telemetry system must be able to handle high volumes of data, provide real-time and accurate visibility into the network, and scale efficiently with the growing demands of networks.

An early and widely adopted tool for network telemetry is the Simple Network Management Protocol (SNMP) [7]. This protocol involves managers sending requests to agents (deployed on network devices), with the agents either responding with the requested data or performing an action. SNMP is useful when users need to retrieve specific information, such as CPU usage, interface counters, or memory usage, from a device. However, it only exposes a limited set of counter measurements. In addition to that, since SNMP uses a polling methodology, it struggles to scale well in larger networks.

Another approach for network telemetry is the use of packet-level monitoring tools, such as Wireshark [1, 16, 15]. Extracting specific information from the data captured by these tools often requires extensive post-processing, which can introduce significant overhead and be resource-intensive. For instance, in a typical university campus network, tens of thousands of users may be using the network and generating millions of packets. Using Wireshark requires parsing every

individual packet and extracting information from it, which is not feasible in such a network. This makes them inefficient for real-time traffic monitoring, as the post-processing cannot keep up with the high volume of data generated by the network. Additionally, these tools can consume considerable storage space, as all captured data must be saved for later analysis.

Flow-level monitoring tools, such as NetFlow [5], provide an alternative by exporting network information at a coarser level. This reduces overall storage usage but can still be inefficient for larger networks. Techniques like packet sampling can help with this to an extent, but at the cost of reduced accuracy, meaning they can lead to the system potentially missing critical information. Additionally, these tools capture only a specific subset of flow parameters, including source and destination IP addresses, port numbers, protocol type, type of service, TCP flags, packet, and byte counts, start and end timestamps, interface numbers, and routing information. Any diagnostic requiring information beyond the predefined features captured during data collection, such as TCP window size, would be unable to use these records effectively. Moreover, these tools do not capture details about how the behavior of individual flows changes over time.

A relatively recent approach to network telemetry is programmable data plane telemetry. This category of systems allows operators to use programmable data planes to customize how traffic data is collected directly at the network's core, often in switches, enabling fine-grained monitoring with minimal overhead and allowing for scalability in larger networks. Moreover, since the processing is handled by dedicated hardware pipelines in application-specific integrated circuits (ASICs) rather than relying on the CPU, programmable dataplanes can operate at line-rate,

delivering real-time information to operators. They can also be programmed *on the fly*, allowing operators to dynamically adjust them to meet their needs.

A network telemetry system based on programmable data planes must meet several key factors to properly harness their capabilities. First, it should be flexible enough to allow operators to customize data collection based on their specific needs and network conditions. Second, it must report results quickly enough to meet user needs. For example, if network telemetry is used for detecting and mitigating security attacks, reports must be generated with sufficient granularity to enable users to identify and respond to threats in real-time—rather than after an attack has already occurred. Third, it should be able to scale with the number of traffic size and number of users in the network. For instance, a campus network could have tens of thousands of users, and a telemetry system deployed on it should be able to generate reports in a timely manner. Lastly, it should be resource-efficient, as programmable dataplanes often have limited resources that must be utilized efficiently and effectively.

To this end, this work presents IRIS, an interactive programmable fine-grained network telemetry system, developed by the Oregon Network Research Group (ONRG) at the University of Oregon. IRIS is built on BroadScan, the flow-analytics engine present on the widely deployed Broadcom StrataXGS ASICs [3]. IRIS focuses on providing an easy-to-use and flexible interface for network operators to configure BroadScan with custom queries and collect real-time results.

The rest of this paper is organized as follows: Section 2 discusses the background of programmable network telemetry and the related works in the field, and describes the contributions of IRIS based on their limitations. Section 3 provides a detailed description of the IRIS architecture and its key components.

In Section 4, IRIS is first evaluated to confirm its ability to operate at maximum capacity. Then, its performance is evaluated based on time-based and loss-based statistics on the receiving, parsing, and post-processing of records exported from BroadScan. This section also presents several use cases to demonstrate IRIS in action. Finally, Section 5 concludes the work and outlines future directions.

CHAPTER II

BACKGROUND

Traditionally, the data plane is responsible for processing network packets by executing a series of operations, which include parsing a packet, determining the processing steps to be applied to it, and forwarding the packet based on the outcomes of these operations. When programmable data planes were introduced, dynamic and programmatic changing of the basic packet processing functionality became possible [11]. Programmable dataplanes allow for customizable, real-time, fine-grained data collection directly at the network's core using dedicated ASICs. This technology resulted in the emergence of a new generation of solutions for programmable telemetry. By configuring hardware with specific queries, these solutions can efficiently report only the relevant information, making them faster and more resource-efficient with performance at line-rate.

There are two common programmable data plane switch ASICs used for network telemetry:

1. **Switches with ASICs supporting P4 (e.g., Tofino switches):**
   These ASICs provide a fully programmable match-action pipeline, allowing operators to define the entire sequence of switch operations. Sketch algorithms are commonly used with this technology. Accessible documentation on the inner workings of these switches, combined with their flexible programmability, contributes to their popularity among researchers. However, deploying telemetry solutions on these devices is only feasible in certain types of networks, such as hyperscalers and research and education (R&E) networks with access to research talent. This is due to the need to

define all switch operations from scratch on these devices. Figure 1 shows a high-level view of this type of technology.

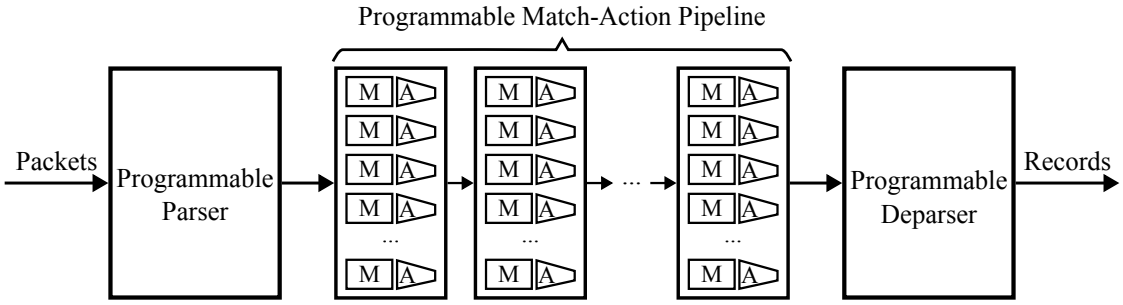

Figure 1. High-Level Overview of ASICs Supporting P4, Such as Tofino Switches

2. **Broadcom-based switches:** These types of switches are already well integrated into existing network stacks and support many common network features out of the box. Switches that have Broadcom's StrataXGS ASIC contain a certain module called the BroadScan flow analytics engine, which adds monitoring capabilities without the need to rebuild the network stack. Unlike the last category of switches, these switches have a pre-defined pipeline that can be configured with Broadcom's software development kit (SDK). However, documentation on how to achieve that is not easily accessible, making it less popular in the research community. Due to the nature of the non-disclosure agreement (NDA) signed with Broadcom, details of the pipeline cannot be shared. However, a high-level overview of the BroadScan module can be observed in figure 2.

Building network telemetry solutions on programmable dataplanes is not without challenges. Using these technologies often requires extensive knowledge about the data plane technology and details about their hardware configurations. Moreover, the limited resources available on these ASICs, including memory and

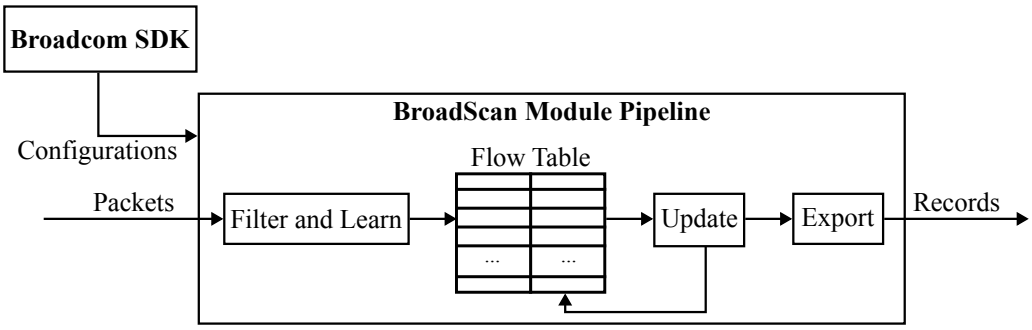

Figure 2. High-Level Overview of the BroadScan Module

table lengths, require optimal resource usage to allow the ASICs to work to their full potential.

Numerous works have been introduced to facilitate the use of programmable data planes for network telemetry to address these challenges. For example, Marple [14], Sonata [6], and ESnet High Touch Services [10] provide declarative interfaces that allow users to express queries for various networking tasks on programmable switches using high-level abstractions. Another group of solutions focuses on building highly scalable systems by balancing improvements in coverage, accuracy, and resource efficiency [19, 9, 8, 18]. Additionally, some works introduce novel algorithms that enable new capabilities and features in network telemetry solutions. For example, Chen et al. focus on enabling the simultaneous execution of multiple queries [4]. Misa et al. present DynATOS [13], a dynamic scheduling algorithm for telemetry queries built on BroadScan, and later its extended version DynATOS+[12], with the added feature of adding user-specified accuracy goals to queries. One limitation of these works is that they typically focus on only one aspect of programmable network telemetry (declarative interfaces, balancing of coverage, accuracy and resource efficiency, adding new capabilities, etc.). Moreover, they often rely on dedicated P4-supporting technology, and only present prototypes

with no real-world deployment. In addition, to the best of our knowledge, no ready-to-use platform for writing dynamic traffic monitoring applications has been presented.

Building a ready-to-use dynamic data plane telemetry platform comes with a few requirements. First, it needs to provide an easy-to-use and unified interface for defining queries, receiving results, and plugging in external tools for extended applications. It should also handle proper query translation to hardware configuration. Balancing resources, accuracy, and coverage is another requirement, based on the limitations of programmable dataplanes. Additionally, operations must be efficient enough to keep pace with received records from these telemetry solutions and deliver near real-time results to the operator after parsing and post-processing the records.

In this work, IRIS is developed on BroadScan, a programmable data plane switch ASIC, to provide such a platform. It is deployed both in a controlled lab environment and on a real campus network to showcase its applicability as a real-world programmable data plane telemetry solution.

CHAPTER III

IRIS

IRIS is ONRG's switch hardware-based network traffic monitoring system. It is built on Broadcom's System Verification Kits (SVKs) which contain ASICs with the BroadScan module. This chapter describes the overall architecture of IRIS, the available layout of the switches in the ONRG lab, and the system requirements for the proper deployment of IRIS.

3.1 IRIS Architecture

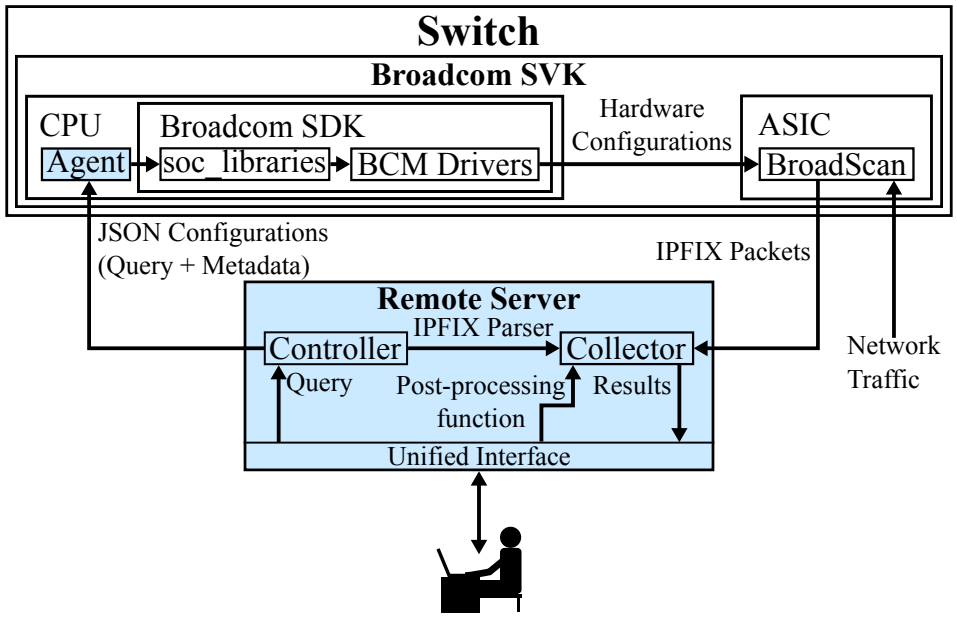

Figure 3. IRIS Architecture

Figure 3 demonstrates the remote-server architecture of IRIS, with the sections specifically implemented by IRIS highlighted. A user operates IRIS through the remote server's interface to send custom queries via the controller to the agent on the switch. The agent receives these queries and employs the BCM drivers through the soc_libraries available in the Broadcom SDK [2] to configure the BroadScan module in the ASIC to track flows, collect flow data, and export

them, all based on the input queries. The user can then collect the exported data through the collector, and if needed, do post-processing on the results. The following sections describe each of IRIS's components in more detail.

3.1.1 BroadScan. The BroadScan flow-analytics engine is a hardware module included in various Broadcom ASICs that can be used to track flows, collect flow data, and report it to a local or remote flow collector. Due to the NDA with Broadcom, the specific hardware details of Broadscan can not be included in this document. However, a high-level and abstract overview is provided to explain IRIS's capabilities.

BroadScan uses a pipeline with the following high-level components to collect and report data from network traffic:

1. **Filter:** Filter packets based on certain header values. This consists of two parts itself:
   (a) **Pre-selection:** Select which packets enter BroadScan
   (b) **Selection:** Select a subset of flows to be considered for a configuration (If multiple queries are to be executed, their selected subflows should be mutually exclusive)
2. **Learn:** Learn flows based on custom flow definition, e.g., the five tuples (source IP, destination IP, source port, destination port, and protocol)
3. **Update:** Update the a custom subset of stored flow parameters when a new packet from a learned flow is received
4. **Export:** Export the flow parameters periodically or based on a specific threshold

A user can use the Broadcom SVK to configure the BroadScan module to define how each of the components in the pipeline must behave. This is feasible through the BCM Drivers, which provide a map between human-readable tables and register names, and machine-readable memory addresses. Broadcom provides these drivers as an open-source SDK [2] which communicates with the ASIC's kernel drivers.

To effectively use BroadScan directly, users must have an in-depth understanding of its pipeline, including the memories and registers involved in its configuration for specific tasks. The SDK provides a set of flowtracker APIs that simplify its usage to some extent. However, these flowtrackers do not expose the full potential of BroadScan, and still require some knowledge of the underlying hardware. While flowtrackers may be sufficient for many industry applications, research often requires greater flexibility for customization.

An alternative to using flowtrackers is programming in C at the System-on-Chip (SoC) level, which grants direct access to memories and registers in order to access all the capabilities of BroadScan. This can be achieved through the APIs available in the SDK's header files and libraries.

IRIS was developed using this latter approach to maximize flexibility and customization. Additionally, it enables users to operate BroadScan through a Python interface without needing to understand the intricacies of BroadScan hardware. IRIS leverages SoC-level APIs, including variations of soc_mem_read(), soc_mem_write(), soc_mem_field_set(), soc_format_field_set(), soc_mem_clear(), soc_reg_field_set(), soc_reg_field_get(), and specialized per-memory read/write APIs provided by the SDK through the BCM drivers. These APIs allow direct interaction with the memories and facilitate the translation of high-level queries

into hardware configurations. Essentially, IRIS can serve as a replacement for the flowtracker APIs, providing a higher-level, query-based interface for defining BroadScan configurations while enabling more flexible access to hardware configurations.

3.1.2 Agent. The agent deployed on the SVK is responsible for receiving the JSON representation of the queries sent by the user from the remote server and translating them into hardware configurations for BroadScan to execute.

To receive queries sent from the remote server, a basic JSON server is implemented using socket programming. Currently, this implementation is designed to handle queries from only a single client at this time.

Once the agent receives a JSON object through the socket containing the queries and metadata sent by the user, it utilizes the Broadcom SDK to convert them into commands that modify the values of the memories and registers in BroadScan, configuring the BroadScan pipeline to generate results based on the queries.

This agent is entirely programmed in C and uses the SDK provided by Broadcom to implement the hardware configurations of BroadScan via the BCM drivers. Although the majority of the contributions of IRIS are in this section, due to the nature of the NDA with Broadcom, details of this implementation that expose BroadScan's hardware are confidential and can not be included in this document.

3.1.3 Remote Server. The remote server in IRIS provides an interface for users to define queries, converts them into their JSON representation, and sends them to the agent on the switch. It also collects exported IPFIX packets

containing query results from the switch and returns the processed results to the user, which is accessible through the same interface.

With ease of use, speed, and flexibility as primary goals, the architecture of the IRIS remote server was designed to align with these objectives, and a combination of Python and Cython was used in its implementation. The interactive interface is fully developed in Python, offering better flexibility and usability. This setup allows operators to either configure and reconfigure BroadScan interactively with new queries as needed, or create algorithms to automatically reconfigure BroadScan based on received results. Most other parts of the remote server are also implemented in Python, with Cython used in performance-critical sections.

Figure 4 demonstrates the class diagram of the remote server, specifying the components implemented in Cython and Python.

The user specifies the high-level BroadScan configurations by creating instances of the Query class. The user must also create an instance of the switch class which will be used to facilitate communication between the remote server and the switch.

The remote server also provides a Logger class that provides a logging feature. The logger can be used to keep track of IRIS's performance statistics and record possible problems in the system.

In addition to its Python interface, the remote server performs operations that, in its logical view, can be divided into two phases: as a controller and as a collector. It acts as a controller when defining queries and transmitting them to the switch, and as a collector when receiving and processing the exported results. The controller and collector are described in more detail in sections 3.3 and 3.4,

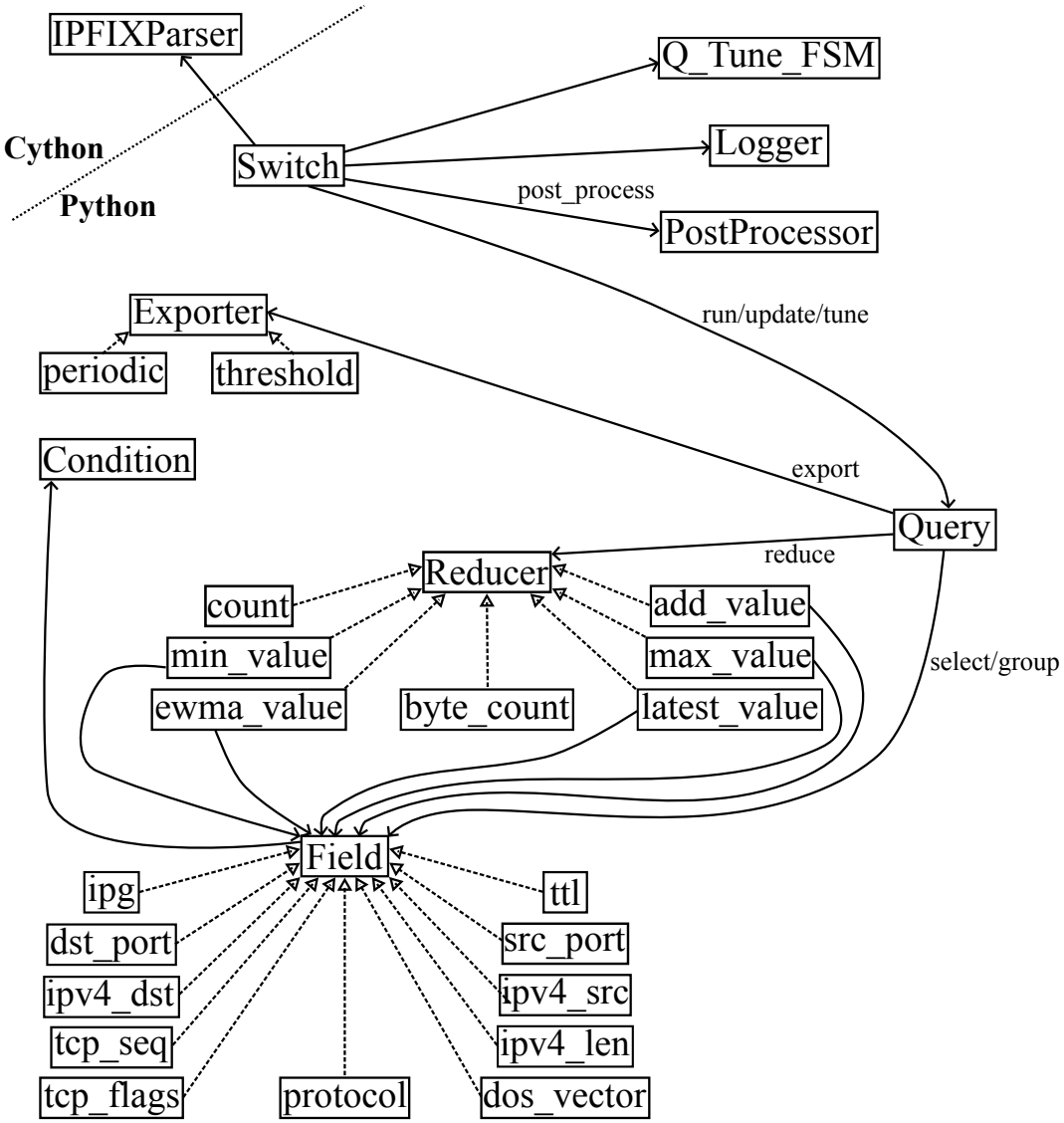

Figure 4. Remote Server Class Diagram

respectively, along with their relationship to the Python interface. These sections also explain other major classes defined in the remote server.

3.2 Python Interface

IRIS provides a unified, easy-to-use, and flexible interactive interface for network operators to configure and reconfigure BroadScan with custom queries on the fly, either interactively or by defining algorithms to update the queries based

on received results. Operator can also define custom post-processing functions for the IPFIX records and receive real-time results, all through the same interface. This interface is implemented in its entirety in python, allowing easy extention and integration with other Python libraries and modules.

3.2.1 Query Definition.

To define a query through the Python Interface, a user must make an instance of the Query class. Several abstract classes are defined to simplify query definition, as depicted in 4:

- **Field:** This abstract class is defined to provide a uniform interface for the fields accessible through IRIS. These Fields can be packet header fields, such as the five-tuple flow definition values (source IP, destination IP, source port, destination port, and protocol), TTL value, or TCP flags. They can also be other types of values that can be calculated by BroadScan's hardware, such as the DoS vector or inter-packet gap (IPG). They can be optionally passed with custom masks and values depending on where they are used.

- **Reducer:** This abstract class provides a common structure for the computations that can be applied to a flow. Each concrete reducer subclass defines the specific operation and optional fields and conditions it takes. Examples of reducers are packet count, byte count, and min/max values. To define conditions in reducers, the Condition class is used to define the range of the target field.

- **Exporter:** This abstract Class can be used to customize the export operation of the query. It has two concrete subclasses:

1. **Periodic:** This type of exporter can be used to periodically export flows with a customizable epoch duration.
2. **Threshold:** This type of exporter can be used with a reducer to export flows when the reducer result reaches a specific value. For example, for byte count, it can export flows only when the packet count for a flow exceeds 1000 bytes.

Using the abstract classes outlined above, the user can then use the following methods in the Query class to customize the query:

- **select():** Takes a list of Fields as its argument with set values and optional masks to filter packets and group a subset of flows. This list logically customizes to the Filter operation of the BroadScan pipeline. It should be noted that the selected Fields for grouping a subset of flows in the selectors list must be unique per query if multiple queries are to be executed in parallel.
- **group_by():** Takes a list of Fields as its argument with optional masks to define flows, e.g., source IP and destination IP pairs, or the five tuples source IP, destination IP, source port, destination port, and protocol. This list maps to the Learn and Track operations of the BroadScan pipeline.
- **process():** Takes a list of Reducers along with the optional Fields and Conditions as its argument, e.g., keep track of the maximum value of TTL per flow. This list configures the Store Flow parameters and Update operations of the BroadScan pipeline.

– **export():** Takes a list of Exporters as its argument, which can either be Periodic or Threshold based. This list defines the export condition of flows, mapping to the Export operation of the BroadScan pipeline.

A few simple examples of how a user can define a query are demonstrated below:

– Define flows as packets with the same source IP and destination IP, and report the number of bytes per flow if they exceed 500:

```
q = Query(q_name) \
    .group_by([ipv4_src(), ipv4_dst()]) \
    .process(byte_count()) \
    .export(threshold('reducer_0', 1000))
```

– Count the total number of packets sent to each special destination port (¡ 1024) and report them every second:

```
q = Query(q_name) \
    .select([protocol(6), dst_port(val=0x0000, mask=0xFC00)]) \
    .group_by([dst_port(mask=0xFFFF)]) \
    .process(count()) \
    .export(periodic(1))
```

– Report maximum IPG per five tuple flow every 5 seconds:

```
q = Query(q_name) \
.group_by([
    ipv4_src(mask=0xFFFFFFFF),
    ipv4_dst(mask=0xFFFFFFFF),
    src_port(mask=0xFFFF),
    dst_port(mask=0xFFFF),
    protocol(mask=0xFF),
    tcp_flags(mask=0xFF)
]) \
.process([max_value(ipg())]) \
.export(periodic(5))
```

3.2.2 Visualization.

Users can use visualization tools with IRIS to better present the post-processed query results. This can be achieved in two ways:

– VIZ: This is IRIS's simple visualization tool developed using the Dash and Plotly libraries in Python that has a few common types of plots pre-implemented. To use this tool, the user needs to create a multiprocessing manager and a new process. VIZ's start() function should be the new process's target, and the query name, a new manager event, and a manager list must also be passed as arguments. VIZ will then be hosted on localhost on the port that it prints for the user, ready to be used.

– Direct implementation: The user can also define custom plots and figures by directly using libraries such as matplotlib or Plotly.

3.3 Controller

The controller's main responsibility is to take the queries the user defines through the interactive Python interface, create a JSON representation of the queries with the necessary metadata, and send them to the switch. It also offers features such as updating a query running on the switch, fine-tuning a the epoch length of a query, or stopping a query and clearing the switch's configurations. In the rest of this section, a JSON representation of a query is simply referred to as query for better readability.

An example of the process in which the controller takes a query defined through the high-level Python interface and translates it into its JSON representation is shown in Figure 5. The query used in this example simply calculates maximum inter-packet gap (IPG) for each 4-tuple TCP flow (source IP address, destination IP address, source port, and destination port) per 1-second epochs.

3.3.1 Running Queries.

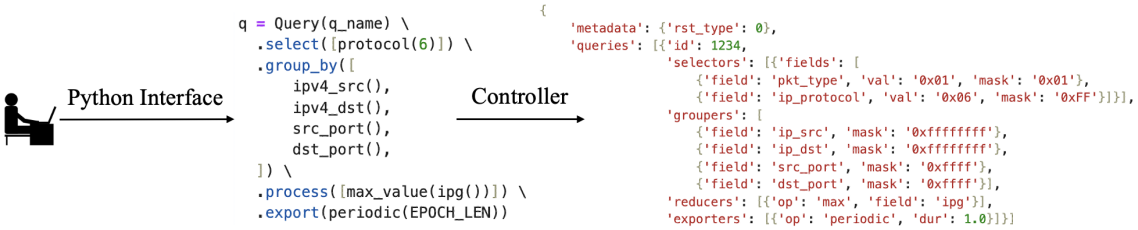

Figure 5. Translation of a Query to its JSON Representation

The run() method of the Switch class instance can be used to send a query to the agent on the switch. This method takes a query or a list of queries as a parameter, along with an optional integer to specify how many epochs the query should be executed for. It then converts the queries into dictionaries and creates a list containing all the dictionary representations of the queries. In addition to that, it creates a new dictionary with metadata key and value pairs, including the reset type. There are two reset types available with IRIS:

- **Full reset:** When this type of resetting is selected, the agent on the switch clears all control memories of the BroadScan module as well as the flow table and flow parameters.

- **Partial reset:** In this reset mode, the agent only updates the specific control memories of BroadScan based on the query it receives but still clears the flow table and flow parameters. This mode should only be used if the user is familiar with BroadScan's hardware details and understands the impact of sending a partial update of a query.

For the run method, the reset mode is always selected as full reset.

Afterward, this method creates a JSON object containing the list of queries and metadata.

The next step involves preparing the collector by creating a new IPFIX parser based on the queries to define the expected results' format, setting up a UDP socket for the collector to receive IPFIX packets from the switch, and then starting a new process to read from the socket and collect the results, optionally for a set number of epochs. The structure of the collector is explained in more detail in section 3.4.

Once the collector is ready, the controller can finally send the new JSON object with the queries and metadata to the agent. This happens through the send_queries() method of the Switch object, which creates a new TCP socket and sends the JSON object to the agent on the switch. If it gets an OK response from the agent, it closes the socket. Otherwise, it warns the user if the agent responded with an error or timed out.

3.3.2 Updating Queries.

The update() method of the Switch class is very similar to the run() method in structure. However, in addition to the query or list of queries, it can take an argument from the user to select the reset type. Additionally, if a previous socket is already available for receiving IPFIX results due to a previous call of the run() or update() method, it can reuse that socket when preparing the collector.

3.3.3 Tuning Queries.

The controller also offers a tune() method in the Switch class in collaboration with the collector. This method uses a finite state machine (FSM) to find the optimal epoch duration for a query based on a target maximum number of records. It takes the following arguments:

- **q:** The query to tune
- **epoch_len length:** The initial epoch duration

- **exp_epoch_cnt:** Number of epochs executed during a single experiment (An "experiment" refers to the execution of a query within a state of the FSM. In other words, exp_epoch_cnt defines how many epochs are to be completed while the query runs in each FSM state.)
- **max_target_rec_cnt:** Maximum desired record count
- **fsm_type (optional):** There are two modes available here:
    - **Quick (0):** This mode is the default and aims to find a stable epoch duration for which at least 90% of the results per experiment do not exceed the max_target_rec_cnt for three consecutive experiments. This mode is useful when a user wants to determine a generally acceptable epoch duration based on the current network conditions, and then run the query using that epoch duration. To this end, the tune() function uses the FSM depicted in figure 6. The state name abbreviations used in this figure are:
        - Init: Initial State
        - US_H: Unsafe Halve
        - S_I: Safe Increment
        - US_D: Unsafe Decrement
        - IRR: Irreducible
        - S: Safe
        - T: Tuned

      This mode starts off in a multiplicative decrease mode to quickly approach the general vicinity of the target epoch duration and then

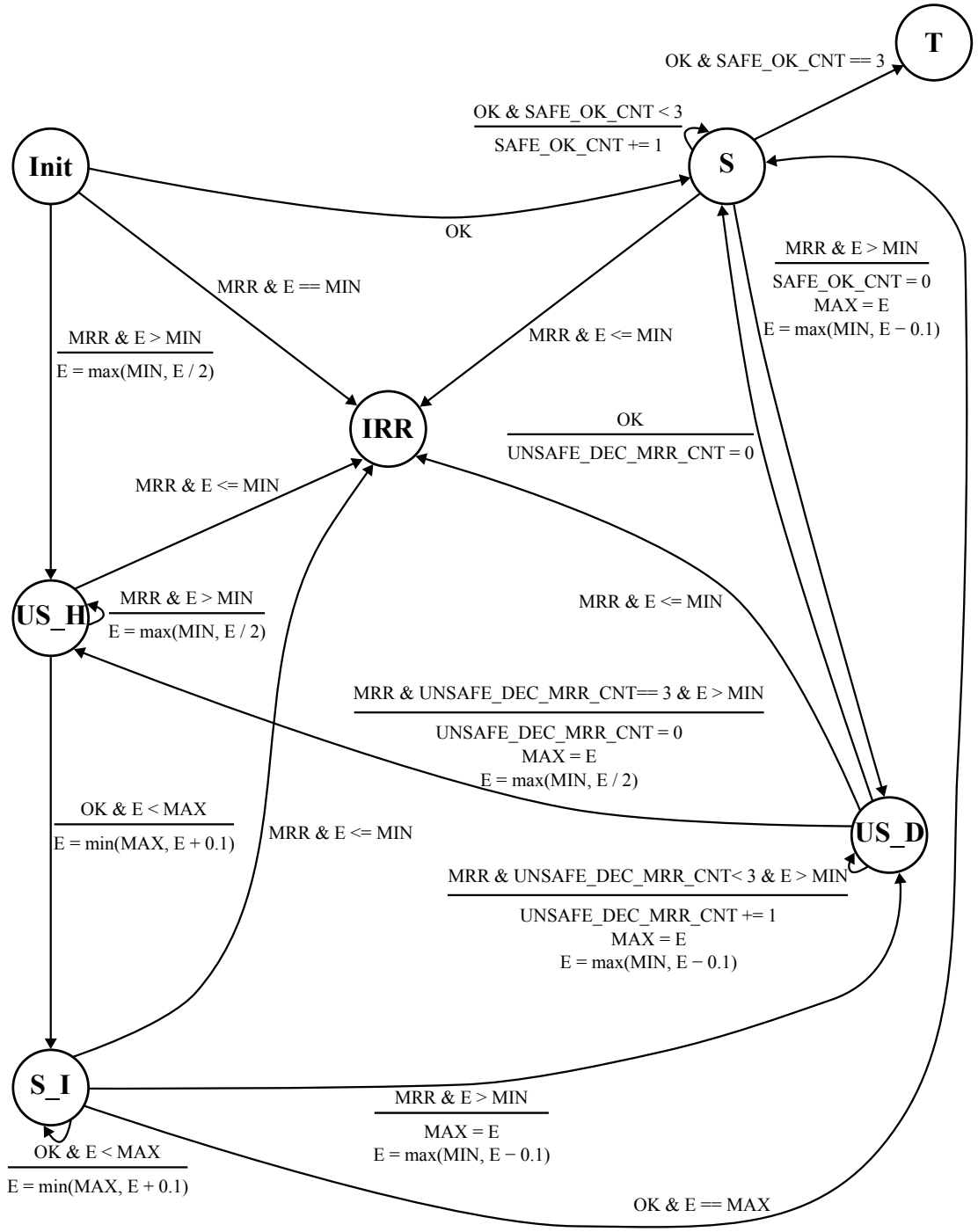

Figure 6. Quick Tuning FSM

State name abbreviations:

transitions to an incremental decrease/additive increase mode to fine-tune it. If more than 10% of the results in an experiment exceed the max_target_rec_cnt at any state of the FSM, the epoch length at that time is set as the upper threshold, and the epoch duration will not be increased beyond that value.

* **Continuous (1):** This mode continuously runs the query and aims to find the maximum epoch duration that at least 90% of the results per experiment do not exceed the max_target_rec_cnt. This mode uses the FSM portrayed in figure 7. The state name abbreviations used in this figure are:
    - Init: Initial State
    - US_H: Unsafe Halve
    - S_I: Safe Increment
    - US_D: Unsafe Decrement
    - S_D: Safe Double
    - IRR: Irreducible
    - S: Safe

  This mode starts off in a multiplicative increase/decrease mode to get to the general vicinity of the target epoch duration faster, and then, like the quick mode, transitions to an incremental decrease/additive increase mode to fine-tune it. However, unlike the quick mode, it does not set an upper threshold on the epoch duration as network conditions may change and a longer epoch duration may result in an acceptable number of records.

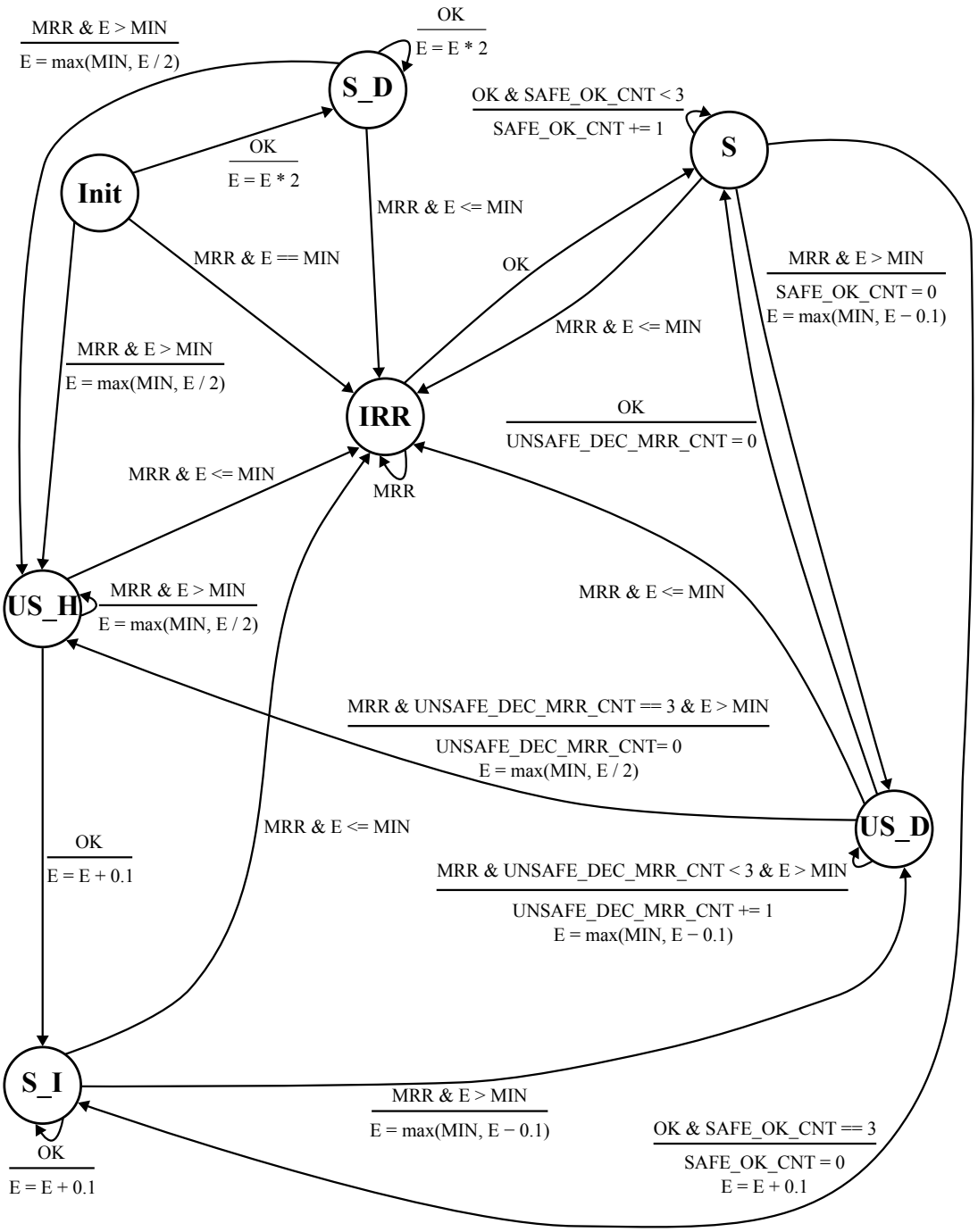

Figure 7. Continuous Tuning FSM

– **tune_dur (optional):** How long to run the tuning process. There is no tuning duration set by default.

– **min_epoch_dur (optional):** This is the minimum accepted epoch duration, which is 0.3 seconds by default.

The tune() function creates an instance of the Q_Tune_FSM class based on epoch_len and fsm_type, and runs the query afterward. While tuning is not complete (tune_dur is not completed, or the FSM has yet to either converge to a final epoch duration or declare tuning impossible within the bounded range in quick mode), the collector collects the results of an experiment and counts the percentages of epochs in the experiment in which the number of records exceeded the max_target_rec_cnt. If this percentage is higher than 10%, it sends an "MRR" signal to the FSM. Otherwise, it sends an "OK" signal. Upon receiving the signal, the FSM moves to the next state and updates its internal epoch value accordingly. After that, the tune() function retrieves the updated epoch value from the FSM and uses the update() function of the Switch class to adjust the epoch length of the periodic exporter for the query and sends the updated value to the switch. Once tuning is complete (if not set to run indefinitely in continuous mode), it returns the final epoch value or declares that tuning failed if it could not determine an epoch duration where the max_target_rec_cnt is not exceeded.

3.3.4 Stopping Queries and Clearing Switch Configurations.

Users can call the stop() method of the Switch instance to reset its state and to command the Agent on the switch to stop all currently running queries and clear all control memories as well as the flow table and flow parameters.

This method also aggregates all the statistical metadata collected during the query runs and reports them, including details such as the cumulative number of IPFIX packets, the number of lost IPFIX records, and timing statistics.

3.4 Collector

The collector's responsibility is receiving all the exported IPFIX records from the switch, parsing them, performing post-processing on them based on the user's custom post-processing function, and reporting the results to the user through the Python interface as a Pandas data frame. With the possibly thousands of IPFIX packets exported in each burst, each packet containing up to 64 IPFIX records with the current BroadScan configurations, there are several potential sources of performance concerns and bottlenecks in the collector regarding IPFIX records:

- **Packet/record loss:** How many records are lost due to IPFIX packet loss?
- **Receiving time:** How long it takes to receive a burst of IPFIX packets from BroadScan?
- **Parsing time:** How long it takes to extract the IPFIX records from the raw byte stream of IPFIX packets and format them?
- **Post-processing time:** How long it takes to execute the custom post-processing function provided by the user?

Motivated by these questions and too ensure the collector operates efficiently, several key factors had to be considered in its design:

- **Slow performance of pure Python:** Although convenient and easy to use, Python is relatively slower in performance compared to more low-level languages such as C. Therefore, to operate more efficiently where speed is critical while maintaining compatibility with the Python components, Cython is a useful tool. In the IRIS collector, the parsing of IPFIX records, observed to be the main bottleneck, is implemented in Cython for better efficiency.

– **Limited UDP socket buffer:** If packets are not received quickly enough from the socket buffer, it will fill up, resulting in packet loss. Python UDP sockets can read from the buffer fast enough to prevent this. However, if the system waits to parse each burst before reading the next batch, it will not be able to keep up with the speed of the incoming IPFIX packets from the switch, resulting in lost records.

  To ensure all packets can be received, a separate process called the IPFIX Receiving Process (IRP) is initialized after the socket is prepared for the connection to the switch. A multiprocessing queue is also instantiated to allow the IRP and the main process to exchange data. This occurs right before sending a query to the switch in the main process, and the IRP begins reading from the buffer continuously. Every time that the IRP gets a burst of IPFIX packets, it adds it to the back of the queue along with the relevant metadata (e.g., epoch number if the query is with an epoch-based export). While the IRP collects the IPFIX packets, the main process is then able to read from the multiprocessing queue, parse the results based on the queries that were sent to the switch, and then pass them as Pandas data frames to the post-processor unit for any custom post-processing function. Figure 8 depicts the structure of the collector and how its different components communicate.

– **Custom Data Post-Processing**: The user is able to define a custom post-processing function for a query, allowing for variable overhead depending on the query. To facilitate this customization, a PostProcessor class is defined. The class constructor takes the switch instance, a custom post-processing function, and an initial state (which is empty by default) as inputs. This

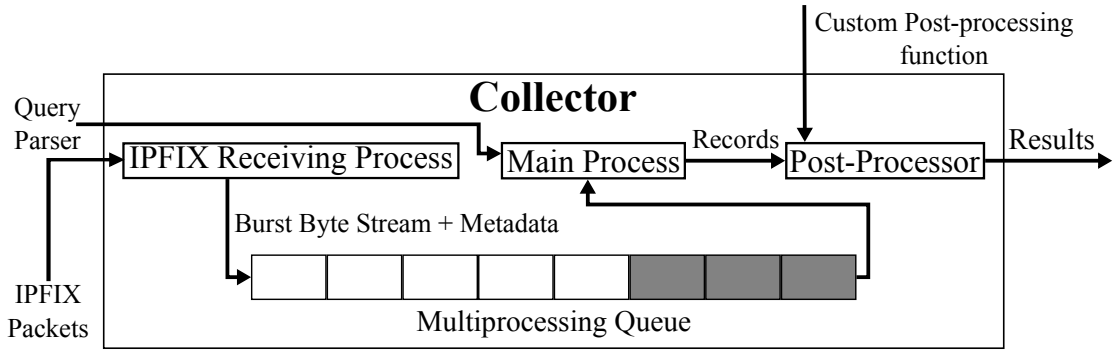

Figure 8. IRIS Collector

class has a get_next() method that takes the next sequence of IPFIX records from the multiprocessing queue of the switch and applies the custom post-processing function to it, which must return the updated state.

It should be noted that this customization also results in variable post-processing overhead completely dependent on the query and the user's self-defined post-processing function. It is advised that precautions are taken to optimize this function to ensure acceptable performance.

3.5 Adding New Capabilities

To add a new capability to IRIS, the following modifications must happen:

- The **Python interface:** needs to be updated, either by creating a new API or modifying an existing one, to enable the user to interact with the new capability.

- The **Controller** must be modified to generate a JSON representation of the new capability from the user-defined query.

- The **Agent** should be modified in two aspects:

1. The JSON server should be able to parse the JSON representation of the query with the added capability
2. The agent should then be modified to configure BroadScan to perform the new capability

– Based on how BroadScan outputs the results with the added capability, the controller should modify the **IPFIX parser** to support the new capability, and pass it to the collector when a query is generated

3.6 Available Switch Layout

The ONRG has two distinct deployments of the IRIS system: one in an isolated lab environment and the other co-located with the network infrastructure of the University of Oregon. Both deployments use Trident3 switches with the BCM56470 SVK which contains the BroadScan module.

1. **Border deployment:** The border switch in this setting gets a mirror feed of all traffic traversing the border of the campus of the University of Oregon. While it is not used for actual switching and forwarding operations, it can be used to monitor campus traffic via its BroadScan module. In this setting, the remote server is deployed on a virtual machine (VM), which introduces the overhead of shared resources. Moreover, instead of a 40Gb ethenter, the remote server and the border switch communicate through a VLAN. Figure 9 shows the border setup of IRIS.

2. **Lab deployment:** The lab switch in this setting is completely isolated and has no real traffic running through it. This setting is typically used for testing purposes with synthtic traffic, which can be generated with libraries such as Scapy, could be a replay of packet traces with tools such as tcpreplay, or

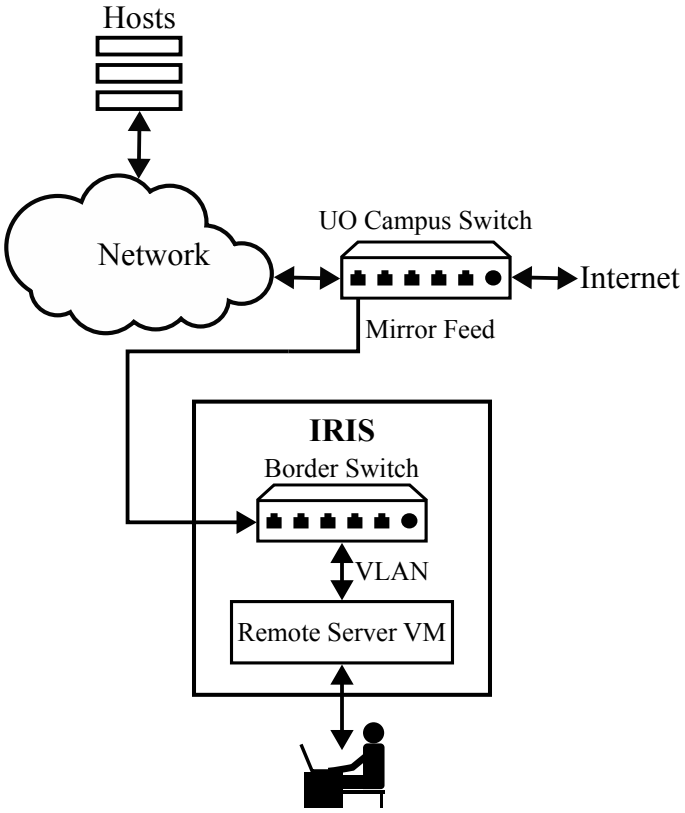

Figure 9. Border Deployment of IRIS

could be a mirror feed from another source. The programmable lab switch is connected to the local network through a bottom-of-the-rack switch, and an additional 40Gbps link directly connects it to the remote server to allow for optimal speed when sending queries and receiving exported IPFIX records. This link can be used to send the synthetic traffic to the switch as well. This switch does not perform any actual forwarding action. In addition, the remote server has a bare-metal setup, allowing maximum utilization of its resources. Figure 10 demonstrates this setup.

Each switch executes a separate instance of IRIS and is managed independently. The lab switch is mostly used for testing purposes with synthetic traffic during software development. The border switch, on the other hand, can be

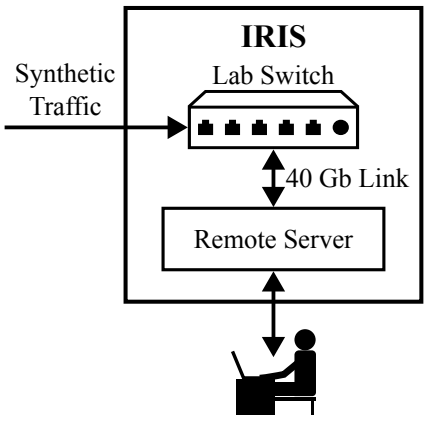

Figure 10. Lab Deployment of IRIS

used for feasibility checks during the software development process as well as for actual telemetry purposes.

3.7 Required System Configurations

To ensure IRIS can function to the best of its abilities, there are certain environment factors that have to be considered:

- **Interface MTU:** The IPFIX packets received by the collector from BroadScan can contain up to 64 records, each record 128 bytes long, as configured in the current IRIS settings. Including the added headers, the IPv4 packets carrying these IPFIX packets can be as large as 8250 bytes. If the MTU is smaller than the size required to accommodate these packets, they will not be delivered to the collector. Therefore, all interfaces between BroadScan and the collector must have their MTU set to a value large enough to ensure these packets can be delivered successfully.

- **UDP socket receive buffer:** Given the fast speed of the IRP and the slower speeds of the parser and post-processor described in section 3.4, it is essential to set the UDP socket receive buffer size to a sufficiently large value.

This ensures that the parser and post-processor have enough time to read the packets and prevents them from being dropped. The UDP receive buffer size of the remote servers is currently at 39321600 bytes.

– **Remote server and switch variables:** To enable communication between the remote server and the switch for sending and receiving queries and IPFIX packets, details such as their IP addresses, MAC addresses, designated ports, and other relevant information must be provided for each new environment. This is done by creating a binary file containing these details and placing a copy on both the SVK and the remote server.

CHAPTER IV

EVALUATION

This section aims to evaluate IRIS's performance in multiple ways. First, its ability to configure BroadScan to its maximum capacity is tested. Afterward, a series of experiments are performed to show IRIS's collector performance, as it contains a few bottlenecks. Lastly, a few telemetry use cases are deployed on IRIS to show its potential as a telemetry solution.

4.1 BroadScan Flow Table Capacity

IRIS should allow users to use BroadScan to monitor the maximum number of flows it can track, which is 262143, fill the entire width of the flow table (256 bits of data to update), and collect all results that the switch exports.

For filling all available entries in the flow table, the following query can be used with custom synthetic traffic sent to the isolated lab switch: The following query tracks all 5-tuple flows, and every five seconds reports the number of packets per flow.

```
q = Query(q_name) \
    .group_by([
        ipv4_src(mask=0xFFFFFFFF),
        ipv4_dst(mask=0xFFFFFFFF),
        src_port(mask=0xFFFF),
        dst_port(mask=0xFFFF),
        protocol(mask=0xFF)
    ]) \
    .process([count()]) \
    .export(periodic(5))
```

To assess if the maximum number of flows can be tracked, synthetic traffic including 280 thousand flows was sent to the switch. Given that this number is larger than the maximum size of the flow table, the switch should report the packet count for the 262143 flows it can track per epoch.

The above query was executed over 20 five-second epochs, and the expected 262143 flows were observed to be reported every epoch.

IRIS should also be able to track all the reducers its can to fill the width of the data section of the flow table. Since the data section of the flow table is 256 bits wide, it should be able to fit 8 32-bit reducers. The following query, targeted towards the size of the flow packets, can be used to evaluate the feasibility of this:

```
q = Query(q_name) \
  .group_by([
        ipv4_src(mask=0xFFFFFFFF),
        ipv4_dst(mask=0xFFFFFFFF),
        src_port(mask=0xFFFF),
        dst_port(mask=0xFFFF),
        protocol(mask=0xFF)]) \
  .process([
        count(),
        count(ipv4_len().in_range(100, 1000)),
        byte_count(),
        min_value(ipv4_len()),
        ewma_value(ipv4_len()),
        max_value(ipv4_len()),
        latest_value(ipv4_len()),
        add_value(ipv4_len())]) \
  .export(periodic(EPOCH_LEN))
```

For evaluation purposes, synthetic traffic consisting of packets from a single flow, with payload sizes drawn from a normal distribution, was sent for twenty epochs to the lab switch to monitor whether all reducers could effectively track the defined byte size statistics. Results presented in Figure 11 show IRIS can configure BroadScan to track the maximum number of reducers it can fit in the flow table, receive exported records, and produce results.

It should be mentioned that IRIS is able to use all flow table entries with reducers filling the flow table to the maximum width at the same time as well.

4.2 Collector Performance Analysis

To evaluate the performance of IRIS's collector, which contains the majority of IRIS's bottlenecks, this section presents statistics on IPFIX record collection

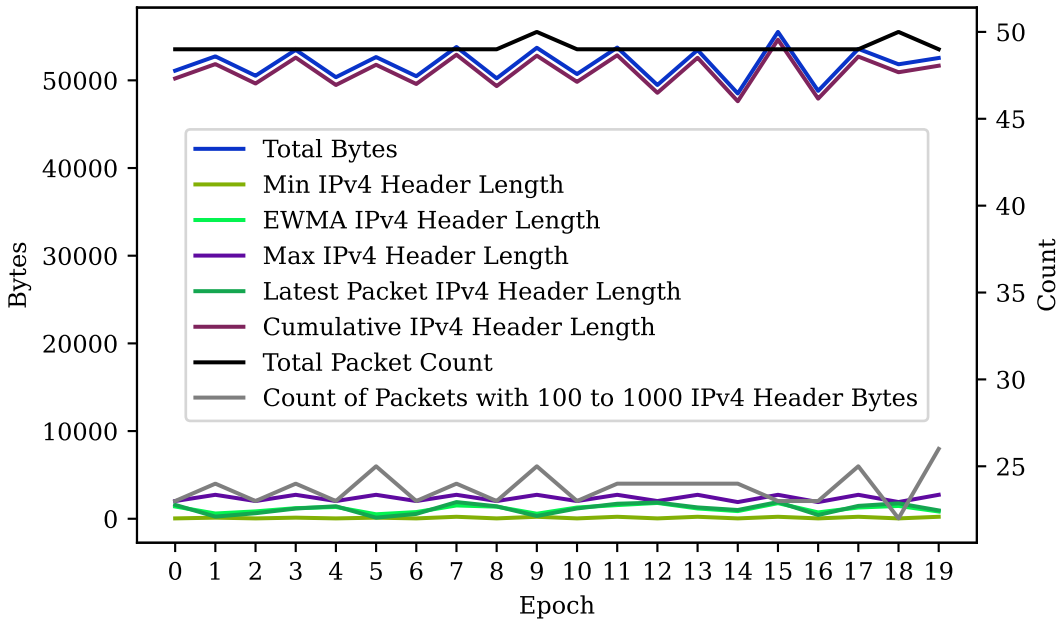

Figure 11. Flow Byte Count and Packet Count Analysis

time, IPFIX record loss percentage, and post-processing time. These statistics are automatically presented by IRIS after each stop of the Switch instance. The query selected for these assessment is a simple one that reports the packet count per five-tuple flow every five seconds with minimal post-processing, with the goal of demonstrating the actual overhead of IRIS without the influence of intensive custom post-processing methods. The query used on IRIS for these assessments is presented here.

```
q = Query(q_name) \
.group_by([
    ipv4_src(mask=0xFFFFFFFF),
    ipv4_dst(mask=0xFFFFFFFF),
    src_port(mask=0xFFFF),
    dst_port(mask=0xFFFF),
    protocol(mask=0xFF)
]) \
  .process([count()]) \
  .export(periodic(EPOCH_LEN))
```

To have a controlled assessment of IRIS with ground truth available for comparison, synthetic traffic is used on the lab switch. The traffic is designed to have a set number of flows, ranging from 20k to 280k, with 20k increments per experiment. For each experiment, the target number of distinct flows is created using the Scapy library in Python, and the flows are then exported as pcap files, prepared to be sent to the switch using the tcpreplay tool. On the lab switch, the query is executed for ten epochs per experiment.

To assess IRIS's performance in a real-world deployment, an experiment is also conducted border switch. The traffic in this experiment is real traffic fed to the switch from the border of the University of Oregon and cannot be controlled. For that reason, the query is simply executed for 100 consecutive epochs instead to observe results.

4.2.1 IPFIX Record Collection Time. The IPFIX record collection time refers to the time it takes IRIS to read all exported records from the IPFIX receiver socket by the controller. To asses the performance of this module in the collector, the IPFIX record collection time is measured and reported based on the total number of records per epoch.

The first experiment for measuring IPFIX record collection time was carried out on the lab switch. The query was executed over ten epochs for each variable number of records to obtain a more accurate measurement. Figure 12 illustrates the results of this experiment.

The second experiment was conducted on the border switch using the same query except the query had IPv4 length added to its groupers to increase number of flows stored in the flow table, and the query was executed for 100 epochs. Figure 13 shows the results of the experiment on the border switch.

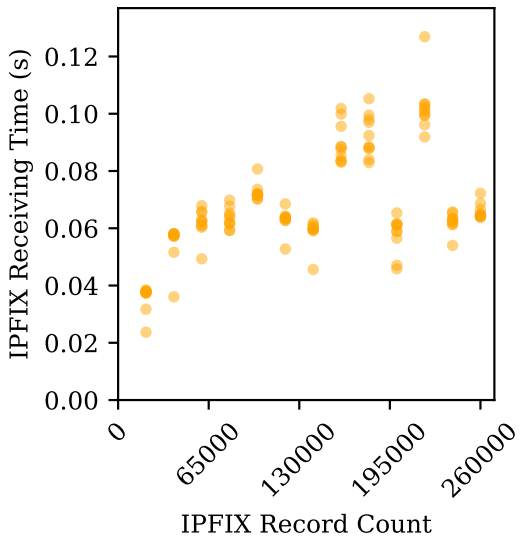

Figure 12. Lab Switch IPFIX Record Collection Time Based on Record Count

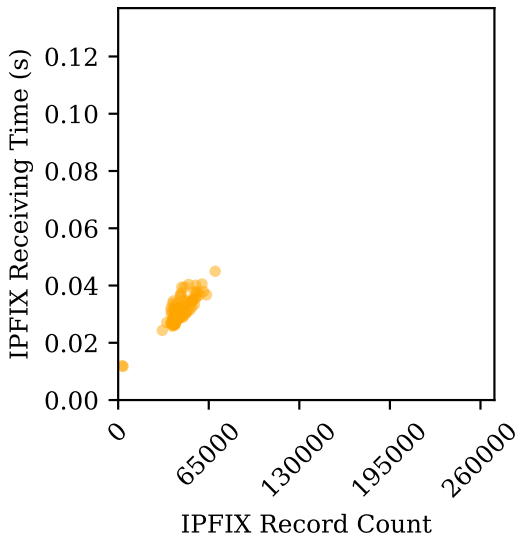

Figure 13. Border Switch IPFIX Record Collection Time Based on Record Count

The figure suggests that the duration of the record collection is, although typically increasing, rather variable even for the same number of records. This can be due to the burstiness of IPFIX packet receiving. However, it is observable that even in the worst-case scenario in this experiment, when the lab switch receives the maximum possible number of flows flow table, IRIS takes just under 130 milliseconds to collect all IPFIX records, showcasing its efficiency.

4.2.2 IPFIX Record Loss Percentage. IPFIX packet loss in IRIS can occur due to reasons such as congestion on the network. In the current lab switch setup, where the switch is connected to a dedicated 40Gbps link, packet loss is extremely rare and is therefore not reported. However, packet loss is possible on the border switch. Figure 14 shows experiment results, and the percentage of IPFIX record loss across all records per epoch, while Figure 15 illustrates the record loss percentage based on tital record count.

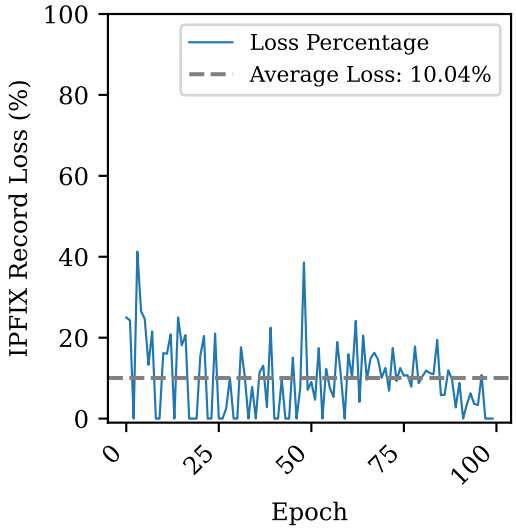

Figure 14. Border Switch IPFIX Record Loss Percentage Per Epoch

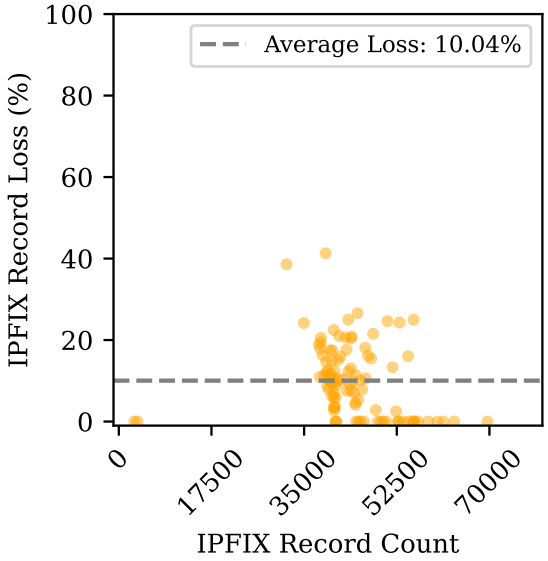

Figure 15. Border Switch IPFIX Record Loss Percentage Based on Record Count

This record loss is most likely due to the VLAN settings on the border vs the direct and dedicated 40Gb link setting in the lab.

4.2.3 IPFIX Record Parsing Time. With IRIS's settings, an IPFIX packet can contain a maximum of 64 records, and the triggering of a periodic or threshold-based export on the switch may result in thousands of IPFIX packets being sent to the collector. While the previous experiment demonstrated that receiving these records from the IPFIX receiver socket buffer is not particularly time-consuming, the parsing process involves extracting the records from all the collected IPFIX packets and formatting them according to the structure of the sent queries for each query, which although implemented in Cython for better performance, can take significant time.

To evaluate IRIS's performance during the parsing process, the parsing time of IPFIX records was measured while executing the query on the lab switch. Figure

16 illustrates how the parsing time increases with the added number of records per epoch. The figure suggests that the parsing time for records is very dependent on the number of records, and can accumulate to significant amounts, which could cause IRIS to fall behind in scenarios where the flow table is nearly full, especially if the epoch length is shorter than five seconds.

Figure 17 presents the parsing time of the query when executed on the border switch. The figure shows a pattern of parsing time increasing as the record count grows, similar to the lab switch . Additionally, within the observed range of record counts, the parsing time on the border switch appears consistent with the parsing time measured on the lab switch for similar number of records.

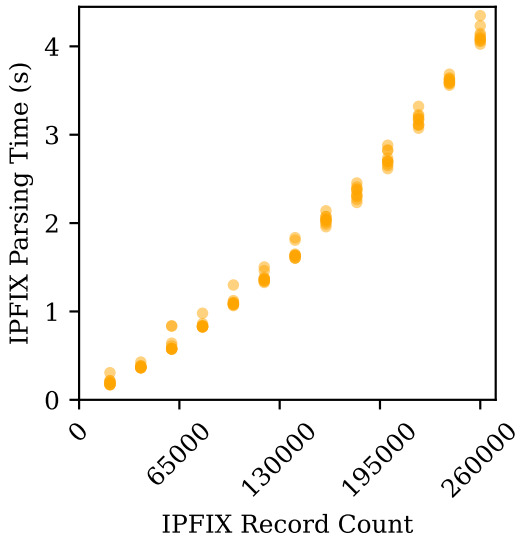

Figure 16. Lab Switch IPFIX Record Parsing Time Based on Record Count

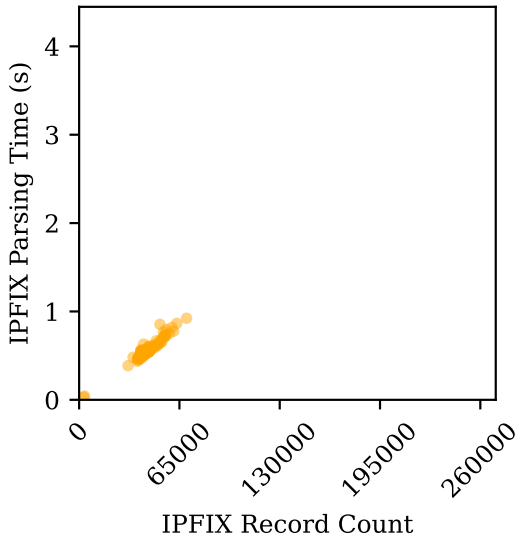

Figure 17. Border Switch IPFIX Record Parsing Time Based on Record Count

This experiment shows that IRIS's IPFIX parsing can take up to over 4 seconds (maximum number of entries in the flow table is 262143), meaning that the main parsing process does not fall behind the receiving process with epoch lengths over 5 seconds.

4.2.4 IPFIX Record Post-Processing Time.

Post processing in IRIS is very query dependent, as it can be fully customized by the user. For this reason, to show the actual overhead of IRIS itself, the query selected for these experiments has little to no post-processing involved. Therefore, running the post-processing function here is quite quick.

To illustrate this point, the query was executed on the lab switch and varying number of flows per second were sent as synthetic traffic to the switch, with results present in 18, and then this experiment was repeated on the border switch, with results demonstrated in 19.

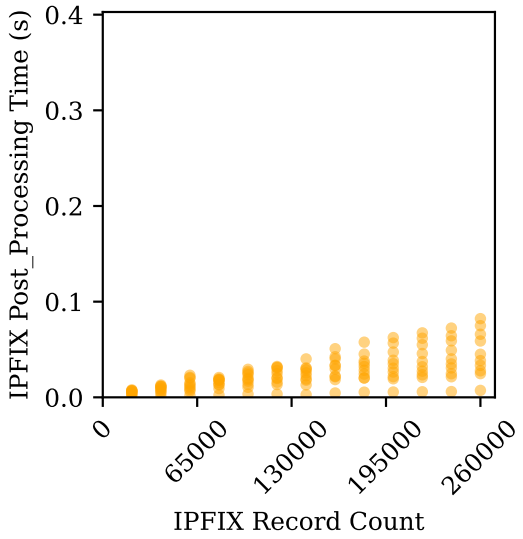

Figure 18. Lab Switch IPFIX Record Post-processing Time Based on Record Count

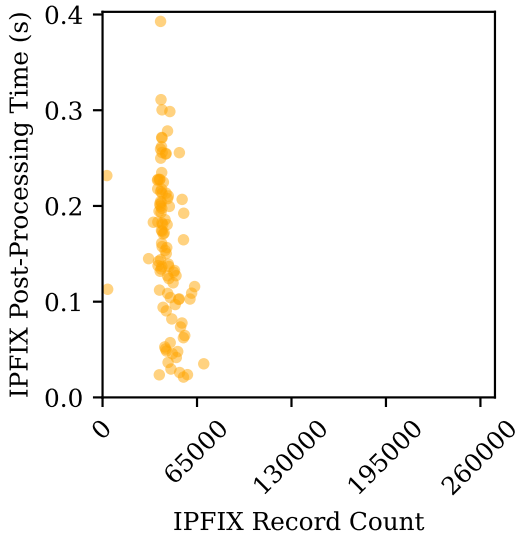

Figure 19. Border Switch IPFIX Record Post-processing Time Based on Record Count

Figure 18 presents the results of this experiment on the lab switch, showing that although dependent on the number of queries, post-processing in its simplest form takes only a small fraction of a second to complete.

It can be seen in figure 19 that on the border switch, post processing for the same number of records can vary greatly. This can be attributed to the virtual machine (VM) deployment of the border remote server, which introduces additional overheads compared to the bare-metal implementation of the lab remote server.

4.3 Example Use Cases

The goal of this section is to demonstrate several use cases for IRIS and provide solutions on how to utilize IRIS to achieve those use cases.

4.3.1 Tuned Byte Count per Source and Destination IPs. This example calculates byte count per host and destination IP pairs with tuned epoch to monitor up to 220k pairs, reports it periodically, and visualizes the results.

It uses a simple query to count bytes and average packet count per source IP and destination IP pairs and reports them periodically. Initially, it is quick-tuned with an initial epoch length value of 2 seconds to an epoch length (less than or equal to the initial 2 seconds) that would result in fewer per-epoch reports than the target record count. After determining the appropriate epoch length, the query is executed for 60 epochs. Post-processing for this execution involves aggregating the results to show the success of the tuning by plotting the number of learned source IP and destination IP pairs per epoch, which corresponds to the number of records. The results are all visualized using VIZ.

```
q = Query(q_name) \
    .group_by([ipv4_src(mask=0xFFFFFFFF), ipv4_dst(mask=0xFFFFFFFF)]) \
    .process([byte_count()]) \
    .export(periodic(EPOCH_LEN))
```

To test this query, it is first executed on the lab switch. The synthetic traffic consists of packets from 270k different source IP and destination IP pairs, continuously sent to the switch. As observable in the code, the initial epoch length for the query is set to two seconds, and the maximum target record count is set to

220k. Figure 20 and 21 show the epoch tuning process and the run of the updated query respectively.

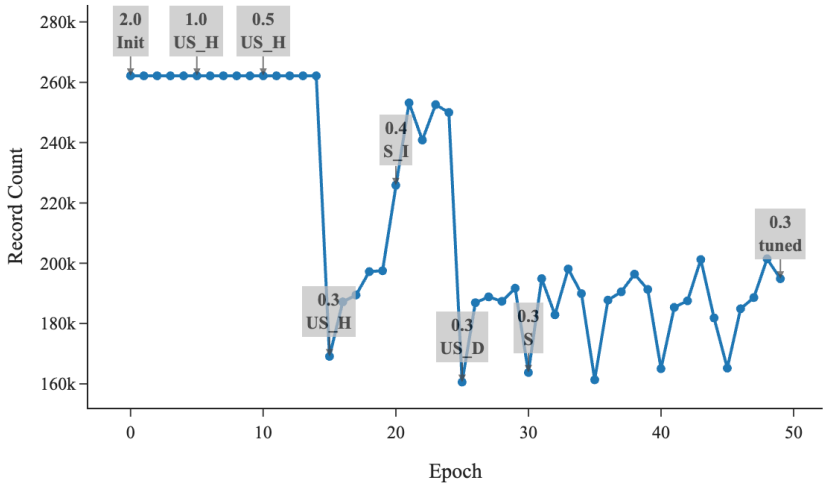

Figure 20. Query Tuning Progress

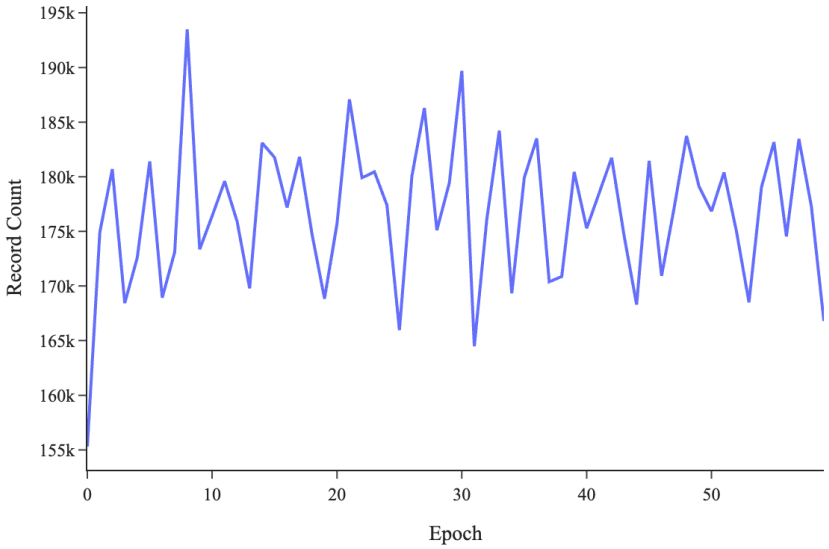

Figure 21. Time Series of Record Count for the Executed Tuned Query

This program was then executed on the border switch. Based on the network conditions, the target record count was modified to be 35k instead. Moreover, the number of epochs per state was increased to 10 for more stability.

The results of this experiment are demonstrated in figures 22 and 23 for the epoch tuning process and the run of the updated query respectively.

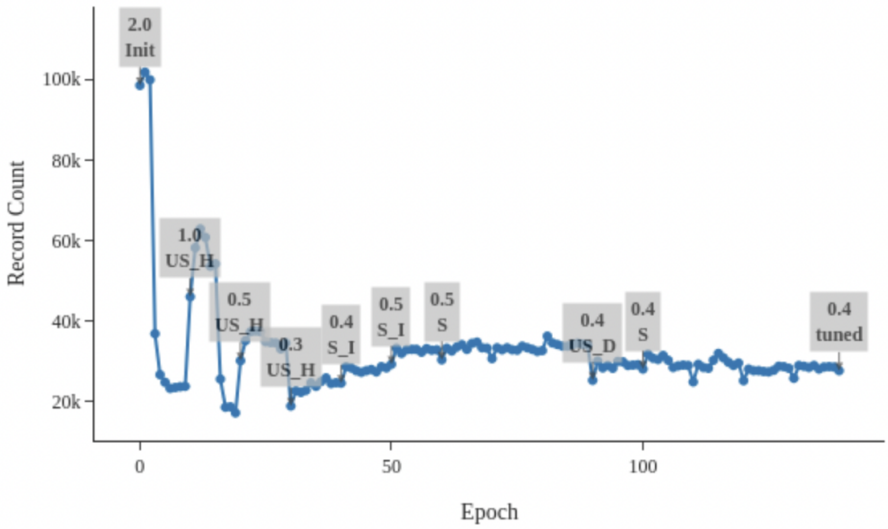

Figure 22. Query Tuning Progress on the Border Switch

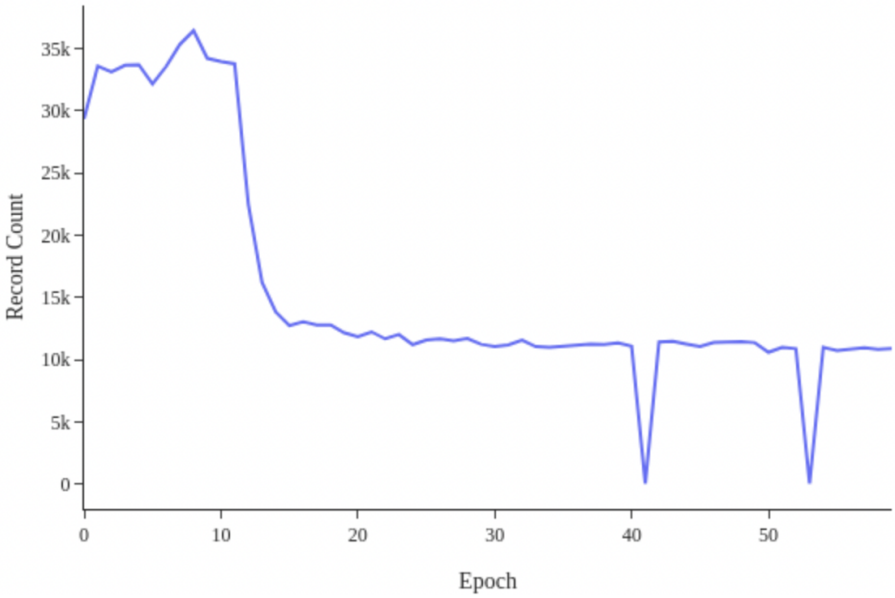

Figure 23. Time Series of Record Count for the Executed Tuned Query on the Border Switch

4.3.2 Top Destination Host by Prefix Zooming.

IRIS provides the flexibility to update the configurations on Broadscan on the fly. To demonstrate this capability, this example use case presents uses a prefix zooming strategy to find the top destination host based on byte count. Such as strategy can be useful for networks with too many connections to keep in the flow table, as each time only a maximum of 255 flows would need to be tracked. The query used for this use case and its pseudo code for the zooming algorithm can be seen below.

```
q = Query(q_name) \
.select([ipv4_dst(str(dst_prefix))]) \
.group_by([
    ipv4_dst(
        mask = int(next(
            dst_prefix.subnets(BITS_PER_ITERATION)
        ).netmask)
    )
]) \
.process(byte_count()) \
.export(periodic(1))
s.run(q)
```

- **Initialize** $dst\_prefix \leftarrow 0.0.0.0/0$
- **Initialize** $BITS\_PER\_ITERATION \leftarrow 8$
- **Do**
  - Define the Query object with current dst_prefix
    - Select destination IPs masked based on dst_prefix
    - Group destination IPs masked based on dst_prefix but with its prefix length + $BITS\_PER\_ITERATION$
  - *top_host* $\leftarrow$ Top destination IP with the highest byte count extracted from the query results
  - Update dst_prefix to the next subnet based on the *top_host* and:

· Increase the prefix length by $BITS\_PER\_ITERATION$

· Generate next subnet from dst_prefix with the updated prefix length

– **While** the prefix length of dst_prefix is less than 32

– **Return** *top_host*

To test the results, it was deployed on the isolated switch with synthetic traffic consisting of packets continuously sent to 160000 different destination IP addresses, with the ones to the destination IP 10.10.9.10 having the heaviest payloads. With this strategy, BroadScan has to monitor and report only up to 20 records each epoch, instead of the whole 160000. This not only allows IRIS to operate in networks that have more flows than what BroadScan's flow table is able to keep track of, but also reduces the collection and post processing overhead. The progress of the top prefix selected at each step can be seen in table 1, showing that the correct destination IP address was recognized.

epoch	**top_prefix**
0	10.0.0.0
1	10.10.0.0
2	10.10.9.0
3	10.10.9.10

Table 1. Top Destination Prefix Based on Byte Count per Epoch with the Zooming Strategy

4.3.3 Anomaly Detection. IRIS's Python interface allows seamless integration with other Python libraries to build more complex programs. In this use case, we demonstrate this capability by employing a Long Short-Term Memory (LSTM) machine learning model, built using TensorFlow and Keras. The model

is trained on flow-level features: flow count, average IPG, average packet count, average byte count, total packet count, and total byte count. IRIS generates these same features, which are then passed to the model for anomaly detection by calculating the reconstruction error. Since this model uses a context length of 512 and a prediction length of 196, the experiment in this use case was carried out for 800 epochs with an epoch length of 1 second. The implemented Query and a section of the post-processing function to generate the required features for this experiment are presented below.

```
q = Query(q_name) \
.group_by([
        ipv4_src(mask=0xFFFFFFFF),
        ipv4_dst(mask=0xFFFFFFFF),
        src_port(mask=0xFFFF),
        dst_port(mask=0xFFFF),
        protocol(mask=0xFF)]
    ) \
.process([max_value(ipg()), count(), byte_count()]) \
.export(periodic(EPOCH_LEN))

def post(state, x):
    df = x[q.id]
    res = df.groupby('epoch', as_index=False).agg(
        time=('time', 'first'),
        flow_cnt=('time', 'count'),
        avg_ipg=('reducer_0', 'mean'),
        avg_pkt_cnt=('reducer_1', 'mean'),
        avg_byte_cnt=('reducer_2', 'mean'),
        tot_pkt_cnt=('reducer_1', 'sum'),
        tot_byte_cnt=('reducer_2', 'sum'),
    ).reset_index(drop=True)
    state = pd.concat([state, res])
    ...
```

In the initial test on the lab switch, synthetic traffic consisting of packets from 32k unique 5-tuple flows is sent to the switch using tcpreplay. Figure 24 shows the result of running the code described above in this scenario, and detected anomalies are colored red.

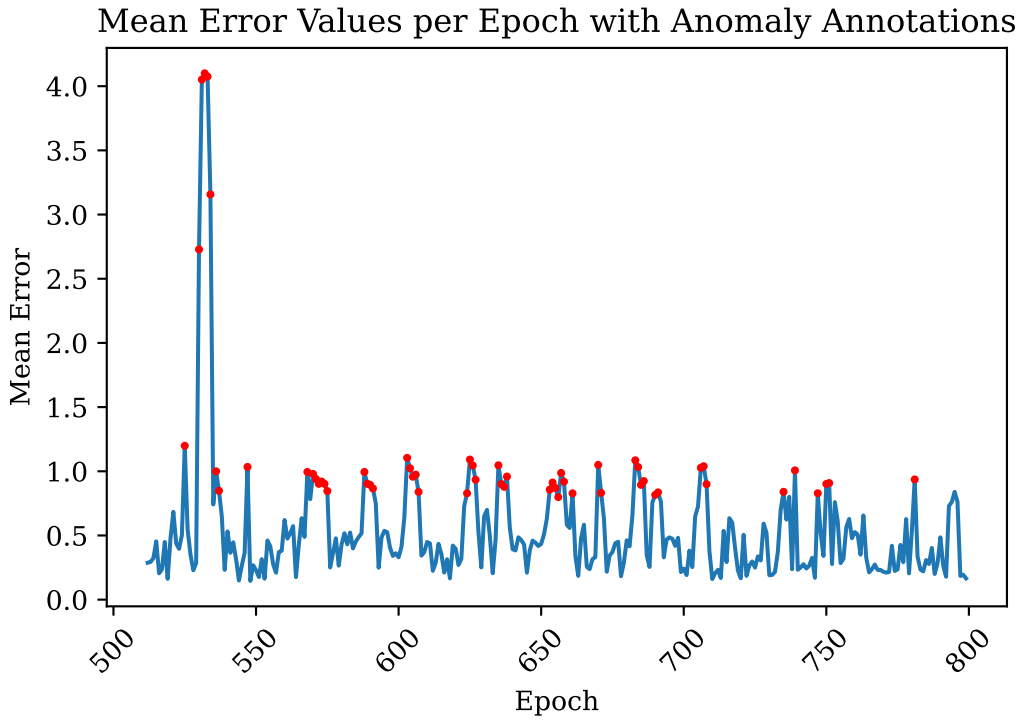

Figure 24. Lab Switch Anomaly Detection

For the next test, the query was executed on the border switch. The rest of the program is the same as the one shown above. Figure 25 demonstrates the detected anomalies on campus traffic.

4.3.4 TTL Histogram. IRIS' reducers can be used to produce histogram of various fields. This example use case presents a query to periodically report the histogram of TTL for TCP flows with detination port 443 (HTTPS).

```
histogram = [count(ttl().in_range(x, x + 31)) for x in range(0, 255, 32)]
q = Query() \
  .select([protocol(6), dst_port(443)]) \
  .group_by([ipv4_src(mask=0x00000000)]) \
  .process(histogram) \
  .export(periodic(1))
s.run(q)
```

The query was executed on the border switch to analyze the distribution of TTL values in campus traffic. The resulting TTL histogram is shown in Figure 26.

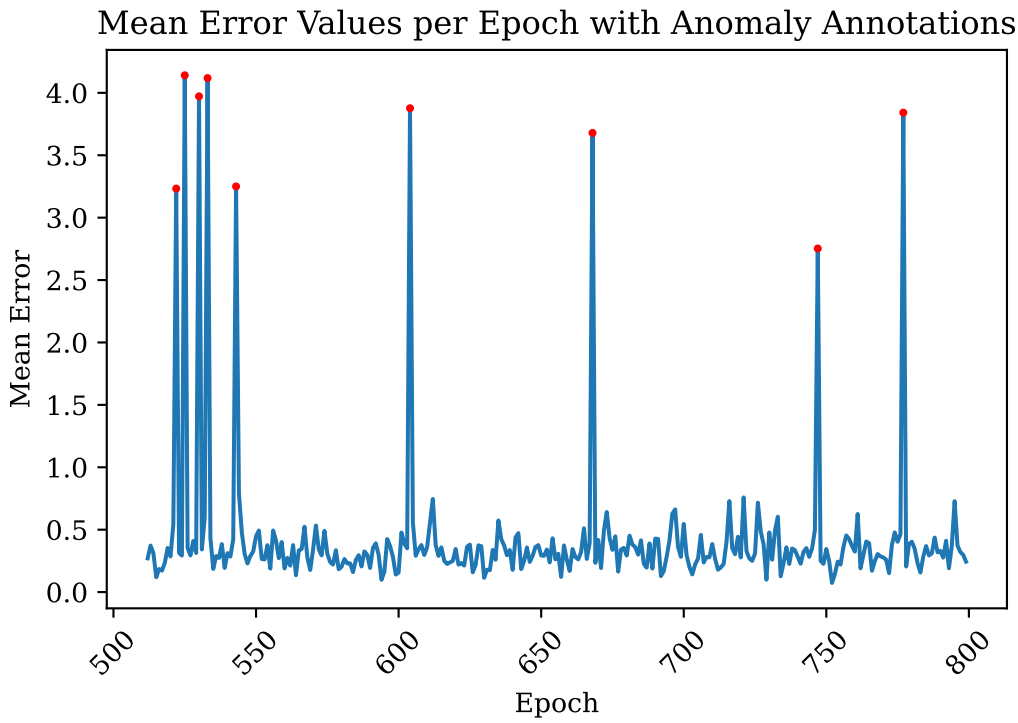

Figure 25. Border Switch Anomaly Detection

The observed values align with TTL values typically seen in networks due to the default TTL settings used by Linux and Windows hosts.

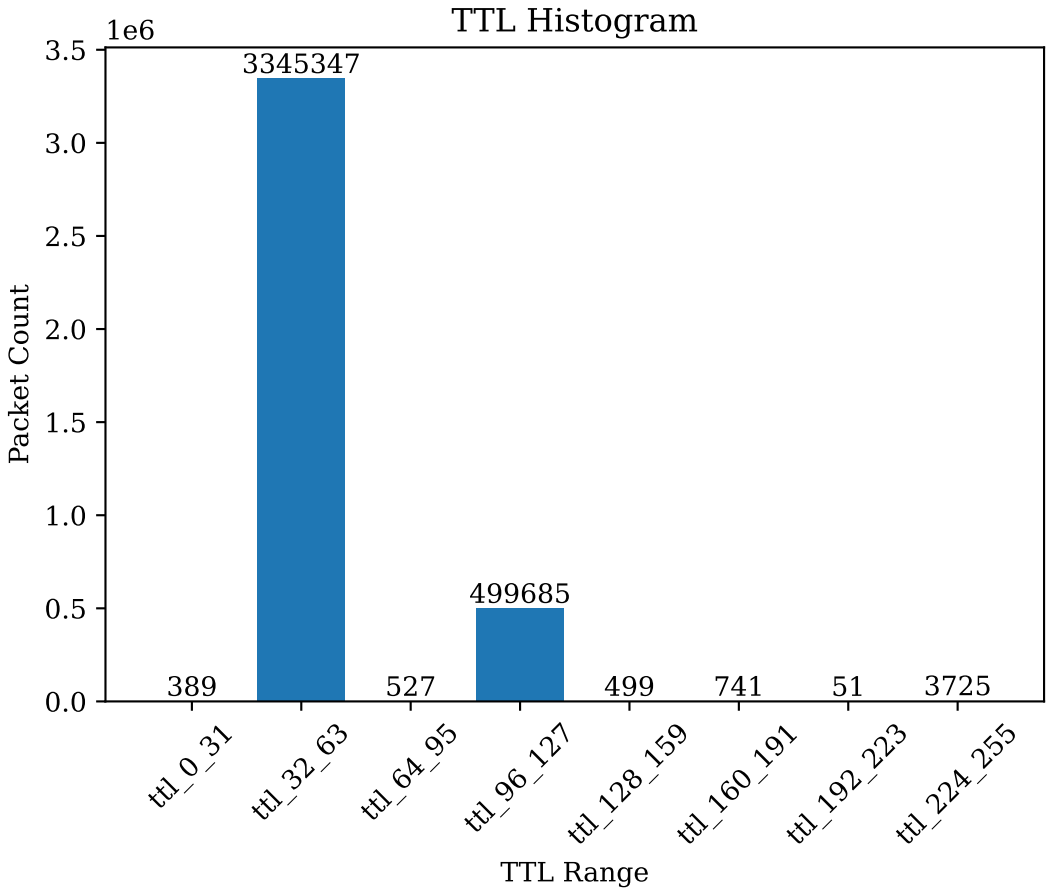

Figure 26. Border Switch TTL Histogram

CHAPTER V

CONCLUSION

Current telemetry solutions, while effective, are often limited as they either focus on specific aspects rather than providing a complete end-to-end solution, require extensive knowledge of the underlying technology and programming of dataplanes, rely on P4-based implementations that are not deployable on Broadcom ASICs, or do not provide real-world deployments.

To address this gap, this work introduces IRIS—A ready-to-use platform for interactive and dynamic programmable telemetry, which is system built on top of the BroadScan module in Broadcom ASICs. IRIS allows users to define high-level switch configurations, translates them into actual hardware configurations on the ASIC by directly managing register and memory values, and collects results from BroadScan. Evaluations show that IRIS can provide real-time results to custom queries of users with epoch durations over 5 seconds.

5.1 Future Work

IRIS can be improved in several aspects:

- Expanding the list of supported BroadScan features in the interface
- Adding security features to the IRIS and using a more secure method of communication between the remote server and the agent
- Adding multi-client support to the agent
- Improving performance for parsing IPFIX records
- Improving the border switch configurations to minimize IPFIX record loss

REFERENCES CITED

[1] Wireshark. https://www.wireshark.org/.

[2] Broadcom. Openbcm broadcom core switch software development kit (sdk). https://github.com/Broadcom-Network-Switching-Software/OpenBCM.

[3] Broadcom. Strataxgs switch solutions. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs.

[4] Chen, X., Landau-Feibish, S., Braverman, M., and Rexford, J. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 226–239.

[5] Claise, B. Cisco systems netflow services export version 9. Tech. rep., 2004.

[6] Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., and Willinger, W. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication* (2018), pp. 357–371.

[7] Harrington, D., Wijnen, B., and Presuhn, R. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. RFC 3411, Dec. 2002.

[8] Huang, Q., Sheng, S., Chen, X., Bao, Y., Zhang, R., Xu, Y., and Zhang, G. Toward {Nearly-Zero-Error} sketching via compressive sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (2021), pp. 1027–1044.

[9] Huang, Q., Sun, H., Lee, P. P., Bai, W., Zhu, F., and Bao, Y. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 404–421.

[10] Liu, Z., Mah, B., Kumar, Y., Guok, C., and Cziva, R. Programmable per-packet network telemetry: From wire to kafka at scale. In *Proceedings of the 2021 on Systems and Network Telemetry and Analytics*. 2020, pp. 33–36.

[11] Michel, O., Bifulco, R., Retvari, G., and Schmid, S. The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Computing Surveys (CSUR) 54*, 4 (2021), 1–36.

[12] Misa, C., Durairajan, R., Rejaie, R., and Willinger, W. Dynatos +: A network telemetry system for dynamic traffic and query workloads. *IEEE/ACM Transactions on Networking* (2024).

[13] Misa, C., O'Connor, W., Durairajan, R., Rejaie, R., and Willinger, W. Dynamic scheduling of approximate telemetry queries. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 701–717.

[14] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the conference of the ACM special interest group on data communication* (2017), pp. 85–98.

[15] Ponomarev, S., and Atkison, T. Industrial control system network intrusion detection by telemetry analysis. *IEEE Transactions on Dependable and Secure Computing 13*, 2 (2015), 252–260.

[16] Ponomarev, S., Wallace, N., and Atkison, T. Detection of ssh host spoofing in control systems through network telemetry analysis. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference* (2014), pp. 21–24.

[17] Song, H., Qin, F., Martinez-Julia, P., Ciavaglia, L., and Wang, A. Network Telemetry Framework. RFC 9232, May 2022.

[18] Sun, H., Huang, Q., Lee, P. P., Bai, W., Zhu, F., and Bao, Y. Distributed network telemetry with resource efficiency and full accuracy. *IEEE/ACM Transactions on Networking 32*, 3 (2024), 1857–1872.

[19] Zhou, Y., Sun, C., Liu, H. H., Miao, R., Bai, S., Li, B., Zheng, Z., Zhu, L., Shen, Z., Xi, Y., et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 76–89.