Offloading INTCollector Events with P4

Jan-Olof Andersson
Jan-Olof Andersson

Offloading INTCollector Events with P4
Abstract

In-Band Network Telemetry (INT) is a new technique in the area of Software-defined networking (SDN) for monitoring SDN enabled networks. INT monitoring provides fine-grained INT data with less load on the control plane since monitoring is done directly at the data plane. The collected INT data is added as packet headers "In-band" at each INT device along the flow path. The INT data is later composed into telemetry packets which are sent to a collector that is responsible for processing the INT data. The collector of the INT data needs to have good performance since there is a large amount of data that has to be processed quickly. INTCollector, a high performance collector of INT data, is a response to this challenge. The performance of INTCollector is optimized by implementing critical parts in eXpress Data Path (XDP), enabling fast packet processing. INTCollector is, moreover, able to reduce processing of INT data and the need for storage space since it employs a strategy where only important INT data is collected, decided by an internal event detection mechanism. The event detection mechanism in INTCollector can however be offloaded to the INT device itself, with possible performance benefits for the collector. Programming Protocol-Independent Packet Processors (P4) opens up this possibility by providing a language for programming network devices. This thesis presents an implementation of INT in P4 with offloaded event detection. We use a programmable P4 testbed to perform an experimental evaluation, which reveal that offloading does indeed benefit INTCollector in terms of performance. Offloading also comes with the advantage of allowing parameters of the event detection logic at the data plane to be accessible to the control plane.

Keywords: INT, INTCollector, P4, Event detection, SDN
Acknowledgements

I would like to express my gratitude to my supervisor Andreas Kassler at Karlstad University for introducing me to this research topic and for guiding me through the thesis. This thesis would not have been possible without his advice and expertise.
Contents

1 Introduction .................................................. 1
    1.1 Motivation .................................................. 2
    1.2 Objective .................................................. 2
    1.3 Structure of the Thesis .................................... 3

2 Background .................................................. 3
    2.1 Software-defined Networking ................................ 3
    2.2 Programmable data planes and P4 ................................ 5
    2.3 In-Band Network Telemetry ...................................... 7
    2.4 eXpress Data Path .............................................. 9
    2.5 INTCollector .................................................. 11
    2.6 Apache Kafka .................................................. 13
    2.7 Elastic Stack .................................................. 14
    2.8 Open Source Network Tester ..................................... 15
    2.9 Related work .................................................. 16

3 Design .......................................................... 17
    3.1 INT Monitor .................................................. 17
        3.1.1 Overview of INT Monitor ................................... 17
        3.1.2 Message Handling ........................................... 19
        3.1.3 Database and Monitoring ................................... 20
        3.1.4 Fast and Normal Path ....................................... 20
    3.2 P4 INT Design .................................................. 21
        3.2.1 INT Design Approach ....................................... 21
        3.2.2 Implementation Language ................................... 22
        3.2.3 INT Mode .................................................. 23
        3.2.4 INT Headers ................................................ 23

iii
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The division of the network infrastructure into planes in SDN.</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>P4 abstract model</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>INT Telemetry mode</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>INT Postcard mode</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Simple overview of system running XDP</td>
<td>10</td>
</tr>
<tr>
<td>2.6</td>
<td>INTCollector in a SDN context</td>
<td>11</td>
</tr>
<tr>
<td>2.7</td>
<td>INTCollector architecture</td>
<td>12</td>
</tr>
<tr>
<td>2.8</td>
<td>Kafka overview</td>
<td>14</td>
</tr>
<tr>
<td>2.9</td>
<td>The composition of Elastic Stack</td>
<td>15</td>
</tr>
<tr>
<td>2.10</td>
<td>Typical setup with OSNT traffic generator, monitor and test device.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Design of INT monitor</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>INT monitoring interface provided as custom dashboard in Kibana.</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>The INT telemetry report format for TCP/UDP headers</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Flow latency event detection example with flow and moving average threshold</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>P4 INT flow chart</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>P4 INT parser</td>
<td>37</td>
</tr>
<tr>
<td>5.1</td>
<td>Setup used for evaluation</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Fast path packet loss count plotted against packet sending rate (Mpps)</td>
<td>46</td>
</tr>
<tr>
<td>5.3</td>
<td>Average CPU usage and number of events detected per hop</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Average CPU usage and number of events detected per flow</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>Measured average CPU usage and event ratio for moving average algorithm.</td>
<td>51</td>
</tr>
<tr>
<td>5.6</td>
<td>Event detection ratio for different values of $\alpha$ for moving average algorithm</td>
<td>52</td>
</tr>
<tr>
<td>5.7</td>
<td>Event detection ratio for each algorithm</td>
<td>53</td>
</tr>
</tbody>
</table>
List of Tables

2.1 INT instructions defined by INT v1.0. ........................................... 8
4.1 INT instructions with corresponding P4 data. ................................. 37
A.1 Average CPU usage and number of events detected per hop with pre-filtering and normal path enabled. ................................................. 59
A.2 Average CPU usage and number of events detected per hop with pre-filtering enabled and normal path disabled. ........................................ 59
A.3 Average CPU usage and number of events detected per hop without pre-filtering enabled and normal path disabled. .............................. 60
A.4 Average CPU usage and number of events detected per hop with both pre-filtering and normal path disabled. ........................................... 60
A.5 Average CPU usage and number of events detected per flow with pre-filtering and normal path enabled. ................................................. 61
A.6 Average CPU usage and number of events detected per flow with pre-filtering disabled and normal path enabled. .............................. 61
A.7 Average CPU usage and number of events detected per flow with both pre-filtering and normal path disabled. ........................................... 62
A.8 Average CPU usage and number of events detected per flow with pre-filtering enabled and normal path disabled. ........................................ 62
A.9 Average CPU usage with pre-filtering enabled and normal path disabled with moving average threshold. .................................................. 63
A.10 Average CPU usage with pre-filtering enabled and normal path enabled with moving average threshold. .................................................. 63
A.11 Number of events detected with pre-filtering enabled and with moving average threshold. ................................................................. 64
1 Introduction

Being able to monitor network devices in real-time in order to improve network decision making is an important aspect in the evolution of Software-defined networking (SDN). In-Band Network Telemetry (INT) \cite{13} is a technique that makes it possible to gather statistics about each network device at packet level granularity. Each packet in a network using INT monitoring carries monitoring data gathered from each switch along the flow path in-band, i.e. as embedded header data. This monitoring data can be stored in a database and later analyzed by a network analytics engine. By having access to such telemetry data, the SDN controller can make suitable decisions, and since the INT data has a fine granularity the decision process is improved. The control plane also experiences less load since INT operates directly on the data plane level. Using INT for monitoring networks thus provides a good choice for fast and fine-grained monitoring.

The entity that collects and stores INT data parsed from INT packets is known as the INT monitor. Typically, an INT monitor device must process a stream of INT packets that arrive at high speed. Therefore, we also refer to INT monitor as stream processor. Implementations of INT monitors include IntMon \cite{24}, Prometheus INT exporter \cite{6} and INTCollector \cite{23}. INTCollector stands out as the most high performance alternative compared to previous INT monitors. INTCollector achieves high performance by having the crucial parts implemented in XDP \cite{17}. Another performance benefit in INTCollector comes from its use of filtering out INT data packets that are deemed unnecessary for storing using event pre-filtering mechanisms. Unnecessary here being INT data that does not change significantly enough compared to previous INT data. Since the amount of INT packets becomes very large, especially in high bandwidth networks, having a collector for INT achieving high performance becomes a priority. INTCollector is a collector that fulfills that requirement.
1.1 Motivation

The event pre-filtering mechanism introduced by INTCollector is a big contributor to the improvement of performance. This, however, still means that INTCollector has to receive and do initial processing of every INT packet sent from the INT enabled network. A possible performance benefit can be gained by moving the event detection logic to the INT sink switch instead. Offloading of the event detection becomes possible with the use of programmable data planes and languages such as Programming protocol-independent packet processors (P4) [10] that targets these devices. The event detection mechanism can thus be implemented as part of the INT switch logic. The INT device can then avoid to create telemetry packets for INT data that does not fulfill the requirement of the event detection mechanism. The INTCollector stream processor will then only have to handle telemetry reports that are actually of interest. This would reduce the load on the collector and improve event processing since it receives only important telemetry event packets to process. Keeping the event filtering logic in the P4 INT switch additionally comes with the benefit of more flexibility since parameters of the algorithms can be controlled from the control plane. Controllers in the control plane can then adjust event detection thresholds based on previously collected INT metrics and detected events.

1.2 Objective

The main objective of this thesis is to demonstrate the possible performance benefits that can be gained by offloading the event detection logic found in INTCollector to the P4 programmable switch. For this purpose, we present an implementation in P4 of INT with event detection. This implementation is used for evaluating and comparing the performance of two setups: one with event detection in the P4 programmable switch and one in the stream processor. To see how the storage of INT metrics affects performance, the experiments will additionally be done with and without the components related to database storage, i.e.
Kafka and Elastic Stack. Other aims of this thesis include the exploration into the behavior of different event detection algorithms and the implementation of a distributed system for handling the collected INT metrics.

1.3 Structure of the Thesis

The thesis is structured as follows. Section 2 covers tools and necessary concepts used in the thesis. Section 3 illustrates the design of the P4 INT and INT monitor and motivates the design decisions. The implementation is covered in section 4 and aims to give the necessary technical details of the implementation. Section 5 presents the results after evaluating the INT implementation with and without offloading of event detection. A discussion is found in section 6 followed by the conclusion of the thesis in section 7.

2 Background

2.1 Software-defined Networking

Software-defined Networking (SDN) is a paradigm in computer networking that addresses limitations in regards to management and design that are found in traditional network architectures. A key concept in SDN is the separation of the network infrastructure into a control plane and a data plane. In this model the control plane consists of a SDN controller that manages and makes decisions about how the network traffic should be forwarded and communicates this to the data plane. The data plane in contrast consists of the network infrastructure, e.g. network switches, which carry the responsibility for interpreting control plane messages and to forward the traffic accordingly. The switches still retains their own kind of internal control plane, but it now serves the function of interpreting the messages from the SDN control plane which mainly operate on tables in the switch. In addition to the data and control plane there is also a third plane that can be included, which is
known as the management plane. The management plane manages network policies that
the control plane in turn enforces [20]. An illustration of the planes in SDN can be seen
in Figure 2.1.

In SDN there are APIs between the planes to allow communication to be compatible
between the higher and lower-level components across the layers. The API between the
control plane and the management plane is known as the northbound interface and is
usually provided by a REST API. The southbound API, situated between the control plane
and data plane, allows controllers from the control plane to communicate with devices on
the data plane. OpenFlow [21] is the most notable standard and open implementation of
a southbound API. Management of the data plane with OpenFlows enables controllers to
configure what kind of actions and matching rules to apply for incoming packets on the
data plane switch. Each such rule is configured through OpenFlow by updating flow tables
in the data plane switch which specifies what actions to apply depending on which packet
header fields that are matched.

The separation of the infrastructure into separate layers has advantages compared to
the traditional network architecture. It simplifies network management by having a central
component that is able to control the forwarding behavior, while at the same time providing
a global view of the network. The flexibility gained from SDN simplifies various things such as network management, troubleshooting and Quality of Service support to name a few [20].

2.2 Programmable data planes and P4

Programming protocol-independent packet processors (P4) [10] is a high-level language specifically designed for programming on the data plane. Traditionally the data plane is defined by fixed ASICs according to vendor specifications, leaving no possibility for using the data plane beyond its given specification. This lack of freedom over the workings of the data plane leads to a more rigid design, forcing the control plane to adhere to the abilities of the network devices. P4 programmable devices turn this around by allowing the data plane to be programmed. This allows the control plane to define how the data plane processing is to be done to meet the needs of the application instead of vice versa.

In a SDN context, the need for programmable data planes with P4 is motivated by the inflexibility found in the OpenFlow API. OpenFlow enabled switches are required to match against header fields which are standardized in OpenFlow. To match new header fields it becomes necessary to update the OpenFlow API and data plane elements. P4 suggests a solution in which these updates to the OpenFlow API can be avoided by allowing the data plane to properly accommodate new packet parsing mechanisms through a common interface. Furthermore, any compatibility issues between interfaces of different vendors are avoided by the use of the common interface defined by P4. The idea is hence to enable the control plane to instruct the behavior of the data plane instead of continuing to revise the specification of OpenFlow.

The greater control of the inner workings of the data plane through the use of P4 opens up many new possibilities. For example, it becomes more feasible to deploy new network protocols which otherwise would have taken a significantly longer time with fixed-function devices that would require new types of hardware to be developed and deployed. Programmable NIC cards with P4 support are currently available for this purpose, in-
cluding programmable routers such as Barefoot Tofino and programmable FPGAs such as NetFPGA SUME. Each programmable device is known in P4 as a target.

The P4 language is a domain specific language that is designed after an abstract model shown in Figure 2.2. The abstract model defined by P4 is an attempt to generalize the workings of packet processors. The abstract models consists, simply put, of a programmable parser, followed by a series of matches, actions and tables. Packets are handled in the abstract model firstly by being parsed in the parser stage, which extracts the header fields from the packet. The packet parser in P4 is defined as a state machine. Each state in the state machine extracts header fields, and depending on the field value will transition to another state. After the packet has been parsed in the parser, the matching tables are applied to the parsed header values. If a match is found in a match table against a header field value a corresponding action will be called with optional parameters. Inside actions it is possible to modify the header fields of the current packet and perform stateful operations that store state persistently, i.e. for more than a single packet. P4 offers means for stateful memory storage in the form of metadata which are header-like structures that can store data, the register type that is able to store data in an array, meters for data rate measurement and counters which are used for counting events. The match and actions section are divided into special ingress and egress controllers, where the ingress matching
rules can specify egress ports and egress queues. A controller in P4 is a description of the order of matching and actions applied to the packets. Once the packet has gone through the match and action process it is finally reassembled in the deparser. Since P4 is designed to be run on restricted hardware there are some limitations to what P4 programs can do. For example, one limitation is the fact that floating point arithmetic is not a mandatory feature that is enforced by P4.

2.3 In-Band Network Telemetry

In-Band Network Telemetry (INT) \cite{INT} is a technique for monitoring networks that collects network state data, known as INT-metadata, directly at the data plane in real-time. This means that no logic regarding network state is put on the controller, instead the controller decides which INT items that the data plane should collect and for which flows. INT is an important field in the area of SDN since it provides a more accurate view of the current state of each node in the network compared to other approaches, such as using ICMP packets to probe the network. The use of INT thus enables various applications such as advanced network trouble shooting, congestion control and routing among others.

INT metadata is inserted on a per packet basis at each hop along the flow path. The INT source is the first INT device in the path and is responsible for adding the initial INT headers before adding INT metadata. The initial INT headers will contain information such as what INT metadata to collect (INT instruction mask), the current hop count, the length of the metadata fields, etc. The specification as defined by \cite{INT} defines eight different types of collectable metadata, listed in Table 2.1. The INT instruction mask can for example tell the INT device to insert the current hop latency and ingress timestamp as INT metadata headers. The rest of the nodes in the path, except the last, are INT in-transit nodes. The in-transit nodes simply look at the INT header and insert the corresponding metadata. Finally, the last node, the INT sink, creates a telemetry packet that is sent to the monitor and another packet containing the original packet headers and payload which
is sent to the original destination of the packet.

Table 2.1: INT instructions defined by INT v1.0.

<table>
<thead>
<tr>
<th>Bit (MSB)</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Switch ID</td>
</tr>
<tr>
<td>1</td>
<td>Level 1 Ingress and Egress port ID</td>
</tr>
<tr>
<td>2</td>
<td>Hop latency</td>
</tr>
<tr>
<td>3</td>
<td>Queue ID and occupancy</td>
</tr>
<tr>
<td>4</td>
<td>Ingress timestamp</td>
</tr>
<tr>
<td>5</td>
<td>Egress timestamp</td>
</tr>
<tr>
<td>6</td>
<td>Level 2 Ingress and Egress port ID</td>
</tr>
<tr>
<td>7</td>
<td>Egress port Tx utilization</td>
</tr>
</tbody>
</table>

The structure of the resulting telemetry report packet will depend on the protocols in use and which type of encapsulation that is followed in the specifications [14][15]. Generally, the outermost will be UDP with a telemetry header, which encapsulates the original packet headers and the INT header together with the INT metadata stack. The payload is not included in the final telemetry packet, since it does not contain anything useful telemetry-wise. The monitor, which in this thesis will be referenced to as the INT monitor, is the entity responsible for collecting and storing the INT metadata from the telemetry reports sent from the INT enabled switches.

![Figure 2.3: INT Telemetry mode.](image-url)
This process for collecting INT metrics is known as in-band telemetry mode \[15\] and is illustrated in Figure \[2.3\]. There is a second approach as well for producing telemetry reports, known as postcard mode, which specifies that each INT device sends telemetry reports directly to the monitor as illustrated in Figure \[2.4\]. In the postcard mode there is no need to differentiate between INT Source and Sink end-points since every node has the same task of producing telemetry reports. Postcard mode neither needs to add any INT headers to the original packets that are passed between the switches. Instead, the packets are forwarded as usual and any INT data is packed into a telemetry packet and sent directly to the monitor.

### 2.4 eXpress Data Path

eXpress Data Path (XDP) \[17\] is a framework that provides a safe environment for packet processing programs to execute directly in the operating system kernel context. Earlier implementations of high performance packet processing \[1\] let user space programs bypass the kernel completely and interact directly with the network hardware for high performance processing. While these earlier approaches lead to increased performance, they come with a cost. Many basic network stack functionalities need to be reimplemented by the user space application and functionalities from the kernel become unused. This is an unsafe and inflexible approach to custom packet processing. XDP addresses these issues with
a new kind of approach. In XDP it is possible to write packet processing programs in a subset of C which are compiled and run in an extended Berkeley Packet Filter (eBPF) virtual machine. Figure 2.5 depicts how XDP interacts with different components in a system. Whenever a new packet is received in the device driver from the NIC, a hook executes which will run the XDP program. The XDP program in the eBPF virtual machine can then process the packet data and decide what to do with the packet, such as dropping it or passing it over to the network stack. The XDP program is analyzed during load-time to ensure that the execution of the program will be done safely and not cause trouble with the kernel. Thus, we get a high performance packet processor that executes in a safe manner in the kernel context where it can access parts of the kernel API. The Linux kernel and several NIC drivers currently support XDP and evaluations of XDP show that XDP can reach a processing rate of 24 million packets per second [17].

Figure 2.5: Simple overview of system running XDP.
2.5 INTCollector

INT monitors in INT supported networks require an INT collector component to process telemetry reports that are produced by the INT devices in the network. INTCollector [23] is a collector for INT with a focus on high performance. The focus on performance stems from the fact that INT enables networks can produce large amounts of telemetry reports and therefore require a collector with good performance to process these reports. INTCollector is, in a larger context, designed to be an integral part of a SDN enabled network with INT support where INTCollector acts as the collector of INT telemetry reports from the network. INT metrics are parsed from the collected INT data and filtered by INTCollector before being stored persistently into a database. These stored metrics are thus readily available for entities that wish to do analysis and monitoring of the network. Notably, a SDN controller becomes able to query the INT metrics database and the controller can thereby obtain a highly granular, real-time perspective over the network. Figure 2.6 depicts how INTCollector could be used for this purpose in a SDN context.

![INTCollector Diagram](image)

Figure 2.6: INTCollector in a SDN context.

Parsing of INT report packets in INTCollector is divided up into a fast path and a
normal path as shown in Figure 2.7. The division is made to separate low-level, performance critical parts of INTCollector into the fast path and less performance critical parts with higher abstraction level into the normal path. The fast path receives all the incoming INT report packets and extracts metrics from the INT metadata, which it later uses for event detection. The fast path is implemented in XDP to give high performance to the fast path. Programs written for XDP have several restrictions, which is also another reason why only parts of the collector is implemented as fast path. The normal path is responsible for collecting the parsed metrics from the fast path and to store these metrics persistently into a database. The operation of the normal path is less performance critical and can be done at periodic intervals.

One aspect of the fast path that gives it an even bigger performance advantage is the use of event detection. An event is defined as a significant change in the metric values or the discovery of new flow paths. A significant change in metric value, in turn, is defined as one of either two cases:

1. Threshold: the current metric value changes, compared to the last stored metric value, by an amount that exceeds a certain configurable threshold.

2. Interval: the current time interval compared to the timestamp of the last stored
metric is equal or larger than a certain periodic value.

The fast path has both approaches implemented and either one can be used for the purpose of event detection. The main benefit of using event detection algorithms is that unimportant metrics are filtered out, leading to a less load on the collector. Although, one should keep in mind the fast path will still parse all incoming packets before filtering them which may still cause a high load on the collector with enough packets. Another benefit with event detection is that the collector requires less storage space for metrics. The trade-off of using event detection is less precise information of the network, since metrics are filtered out.

2.6 Apache Kafka

Apache Kafka [19] is an open source stream processing platform that provides both low latency and high throughput. Kafka run as several instances in a cluster among a set of multiple servers. Each Kafka cluster stores records into a set of categories known as Kafka topics. Each record in turn is stored as key-value pairs together with a timestamp. Each topic contains multiple partitions which are distributed between the Kafka instances for fault tolerance, scalability and high throughput.

Apache Kafka has four different API:s available that interacts with the Kafka cluster[2]. The producer API is used for publishing records to Kafka topics in the Kafka cluster. The published records can be read from subscribed topics in the Kafka cluster through the consumer API. The streams API is used for processing streams between topics, consuming one topic and producing to another. Finally, the connector API allows for external applications to connect to and consume or produce to Kafka topics.

An overview of the Kafka platform is shown in Figure 2.8. In the simple case, there will be a set of Kafka producers writing records into partitions for a set of topics. Each partition of a topic are stored across several nodes in the Kafka cluster. In this way it becomes possible for several consumers to consume a topic in parallel with high throughput since
the topic records are distributed over the Kafka cluster. Whenever the Kafka consumers are ready, they will read a set of records from the record sequence of their subscribed topic.

2.7 Elastic Stack

Elastic Stack \[5\] is a platform for collecting, analyzing, and visualizing data. It consists of three different open source software projects that are deployed together: Elasticsearch, Logstash, and Kibana. Logstash is an engine that processes data, e.g., log files or metrics, and filters or transforms the data into a format usable for storing and analyzing before outputting it. Elasticsearch is an engine that stores and indexes data. Queries can be sent to Elasticsearch over a REST API to get and filter data. Kibana is a plugin to Elasticsearch that provides means of visualizing data via a web interface, which becomes useful for analytics of data since it can be done in real-time.

Together they form the Elastic Stack. Figure 2.9 shows how each layer in the Elastic Stack interacts with each other. The logstash acts as the indata pipeline, aggregating and transforming the data representation, and sends the data to Elasticsearch. The Elasticsearch stores the data coming from logstash and indexes the data. The interface of Kibana
provides several options of visualizing data, which Kibana transparently transforms into queries to Elasticsearch and displays the results, e.g. as graphs or data tables.

2.8 Open Source Network Tester

Open Source Network Tester (OSNT) \cite{8} is a open source system for network testing. It addresses the expensive and slow development cycle involved in traditional ways of network testing. OSNT relies on the use of programmable cards, i.e. FPGA devices. Thus, it is possible to do network testing with OSNT at line-rate without having to adhere to hardware limitations and specification by vendors. The OSNT architecture includes the OSNT Traffic Generator and the OSNT Traffic Monitor. A typical setup is depicted in Figure 2.10. The OSNT Traffic Generator is able to send packets at line rate and small packet sizes. The packets to send can be stored as pcaps, i.e. replayable packet data, and loaded into memory for quick sending. With each packet it is possible to add timestamp for delay calculation. The OSNT Traffic Monitor acts as a traffic capturer and is optimized for high processing, low-loss and 6.26ns time stamp precision. The packets captured by the OSNT Traffic Monitor are handed over to the host system for additional processing.

The benefits of the open source and high performance nature of OSNT is seen in the ability for flexible and reusable solutions in regards to network testing. Additionally, the OSNT architecture fulfills the most important needs of network testing when it comes to both packet generation and packet capturing.
2.9 Related work

Other work related to reducing the workload on the stream processor caused by large numbers of network telemetry packets include INTCollector [covered in section 2.5], Sonata [16] and BurstRadar [18].

Sonata, a query driven network telemetry system, recognizes the scalability issue with having a stream processor process all telemetry packets from the network. The other approach of offloading the query system to the data plane however comes with limitations, such as memory space, and lack of query expressiveness compared to executing queries in the stream processor. Sonata proposes a compromise between the expressiveness of the stream processor and the scalability of the data plane by combining them. Thus, Sonata queries are partitioned into subset elements that span both the stream processor and the data plane with a focus on putting as much as possible of the query processing in the data plane. As a result, the reduction of the amount of network telemetry packets to the stream processor reduces the workload of the stream processor with as much as 7 orders of magnitude. The query system itself consists of expressive high level operations, such as filter, map and reduce, making it a powerful domain specific query interface that simplifies network analysis and management.
BurstRadar is an efficient monitoring system, running on the data plane, for detecting microbursts. In environments with high speed, i.e. at 10 Gpbs or more, the latency caused by microbursts becomes a substantial issue. Using an approach such as INT would require that every packet is collected and processed by the stream processor even though the amount of packets related to microbursts overall are few. BurstRadar proposes a system that implements a microburst detection mechanism on the data plane which ensures that only packets that contribute to microbursts are processed by the stream processor. The BurstRadar system works by detecting and then marking packets involved in microbursts with a snapshot algorithm. The marked packets are stored in a ring buffer before being finally being copied into a courier packet and sent to the monitor. With this technique, BurstRadar is able to reduce the amount of telemetry data to collect by a factor of 10.

3 Design

This section covers the design decisions involved in realizing a setup with event detection offloaded from INTCollector to the P4 INT device. The overall aim of the design is to create a setup that provides as high performance as possible. The design is divided into two larger sections. Section 3.1 covers the design of the INT monitor. The INT monitor consists of various components, and each addition or modification of its design is elaborated upon. Section 3.2 covers the design of a P4 based INT implementation with event detection. There we motivate the approach of the design, the choice of INT mode and headers types and the choice of algorithms used for event detection.

3.1 INT Monitor

3.1.1 Overview of INT Monitor

Figure 3.1 shows the overall design of the INT monitor. The INT monitor runs an instance of INTCollector, with additional components, that listens on incoming packets from the
P4 INT sink node. A notable difference to the original design of INTCollector is that the event detection mechanism has been offloaded to the P4 INT sink. The event detection mechanisms in the original INTCollector design runs in the XDP part of INTCollector after the parsing process of the received INT data. In this new design, the sole responsibility of the fast path is therefore to parse the INT data from the telemetry packet and then notify the normal path. The normal path will get an event notification and then read the metrics that are stored in eBFP tables in the fast path. The normal path acts as a Kafka producer and therefore, after doing format conversion of the metrics, will send the metrics to a Kafka cluster. The Kafka cluster in turn passes the messages to an Elastic Stack instance which stores and provides tools for monitoring and visualizing the data.

Figure 3.1 provides a diagram of a simple design since it only shows how to deploy the INT monitor as a monolith. The INT monitor can also be run as multiple instances to balance out the load of INT reports from the INT network. This setup would however also require some kind of load balancing mechanism that ensures that no collector instance gets overloaded. There is also some challenges involved in large INT networks that contain multiple INT sinks. This is however out of scope of this thesis. For our purposes we only assume a network consisting of a single INT sink device and a single INT monitor.
The Kafka cluster is meant to run on a cluster of machines and by running it on a single machine on the host one loses many of its benefits. The same goes for Elastic stack that can be run on another machine, even having each layer of the Elastic Stack spread across different machines. Setting up a fully distributed cluster of Kafka and Elastic Stack was considered, but to avoid complexity in the design and since the monitor runs on high end hardware it was designed as a monolith.

3.1.2 Message Handling

An additional component was added to the INT monitor which is placed between the normal path and the database. The normal path in INTCollector originally sends the collected messages with INT data from the fast path directly into a database. This however means that there is only one path of communication from the normal path to the database. In the INT monitor design, a middle component is introduced to buffer messages sent from the normal path before storing the messages into the database. This component is the Kafka cluster which provides several Kafka instances that the normal path can send messages to. Using the Kafka cluster has several benefits for the overall performance of the INT monitor. Kafka is distributed, meaning we can run Kafka instances over several different machines. Several instances of INTCollector can therefore push data to the Kafka cluster that will evenly distribute the load, leading to an increase in throughput. This design also leads to a system with higher redundancy, by having copies of messages, and fault tolerance, since a Kafka node that fails is quickly replaced by another Kafka instance. Kafka is also scalable and thus can adapt to high loads imposed by the collector. The downside of adding a Kafka cluster middle layer is the increase in complexity and the need for additional resources for running the Kafka cluster.
3.1.3 Database and Monitoring

The messages from the Kafka cluster is processed, stored and visualized in the Elastic Stack. INTCollector provides normal path implementations for use with Influxdb or Prometheus as the database. Grafana is then applied on top of the database as an interface for visualizing the data in the database. The database and monitoring tool choice fell however on the Elastic Stack. The Elastic Stack provides all the components needed into a single coherent software stack. The stack comes both with a database with a powerful data query interface, Elastic search, and a visualization interface, Kibana. Elastic Stack is easier to setup than the Prometheus/Influxdb and Grafana combination since everything comes in a bundle with all the components well tuned to each other per default while also bringing powerful features, e.g. querying, out-of-the-box. The full stack may however be excessive if no human-friendly interface for monitoring is needed, nor the powerful query capabilities of Elastic search. The INT monitor was however designed to be able to monitor INT data in real-time and to provide powerful data visualization features. Figure 3.2 shows the Kibana interface in action with a custom dashboard showing INT data.

3.1.4 Fast and Normal Path

As mentioned earlier, INTCollector is divided up into a fast and a normal path. The removal of the event detection logic from the fast path leaves only the processing of INT data of the telemetry packets left in the fast path. Extra space therefore becomes available in the XDP fast path that can be used for other purposes. The extra memory is however not utilized by the normal path since the normal path logic is closely tied to application level APIs.

The normal path is designed to send the INT metrics from the fast path to the Kafka cluster and it therefore runs a Kafka producer instance which communicates over the Kafka API. The messages sent to the Kafka cluster and the rest of the processing must be in a format that each component is able to understand. The JSON format is used
for communication from the normal path since it is a common, well-supported and highly readable data format. The metric data is read into user space by the normal path from the XDP program and the metric data is encoded as JSON data. The JSON encoded data can then be sent, knowing that the Kafka cluster and other components will be able to decode the message.

3.2 P4 INT Design

3.2.1 INT Design Approach

There have been previous efforts to realize an implementation of INT with P4 \[4\] \[3\]. The implementation of the previous efforts however contain incompatible elements that are not in line with the INT design required for this thesis. For example, they have a different choice of INT headers and provide a limited implementation of INT features. A new
implementation of INT was therefore decided to be implemented to fully meet our design criteria. While the P4 INT implementation in this thesis tries to be a general solution, it is primarily designed for a small network setup with three INT devices. Having three INT devices is just enough to cover the three types of INT devices found in an INT network: the INT source, INT in-transit and INT sink devices. The challenges involved with larger INT networks and different devices is not taken into consideration in the design. For example, adding more in-transit nodes will constrain the header parsing process which has a limited range of bytes that it is able to parse. Not having to add general solutions to the implementation also simplifies the development process and it becomes easier to optimize against our specific setup.

3.2.2 Implementation Language

Offloading the event detection from INTCollector means that we need to design INT firmware that provides the event detection mechanism. The platform chosen for doing this is on a set of programmable Netronome SmartNICs. The P4 language was chosen for writing the INT implementation for these cards. P4 is specially designed for developing on the data plane, making it suitable for writing an INT implementation, and its toolchain supports the SmartNICs as compilation targets. The version of P4 is version 16 [11]. P4 version 16 differs syntactically from the previous release, P4 version 14, and the two versions are largely incompatible with each other. P4 16 was chosen since it is the latest release and has syntactical improvements compared to that of P4 14. For example, P4 14 uses special syntax for modifying header fields while P4 16 allows using an assignment operator. P4 14, on the other hand, does have the advantage of having good support for targets and developer environments. The targets used were found to support P4 16 and the developer environment provided all necessary features needed, thus the choice fell on P4 16.
3.2.3 INT Mode

The P4 INT implementation process telemetry reports in INT telemetry mode. The alternative, INT postcard mode, sends telemetry reports at every hop, which means that a lot of telemetry packets will be sent over the network. The Telemetry mode instead lets the INT sink node create a compact telemetry packet containing all the INT data collected from the previous INT devices. The advantage of postcard mode lies in its simplicity since it does not require modifying packets by adding INT headers and, by extension, the concept of INT source and INT sink becomes unnecessary. INT telemetry mode is a more standard mode and while it is more complex to implement it produces a less total amount of telemetry packets compared to postcard mode. Since all the data of a flow path is contained into a single telemetry packet is also becomes easier to process the telemetry on a per flow basis.

3.2.4 INT Headers

INT require two types of header formats to be available. The first header is the INT header format which is inserted at the INT enabled switches. This header will contain various options, such as the number of hops traversed and which INT metadata to collect. The second format is the telemetry header format which is added to the packet before the INT switch sends it to the INT monitor. There are currently two versions, version 1.0 and 0.5, of the INT header\[14\][12] and Telemetry header format[15][13] specifications. The specification for version 1.0 introduces some change in the INT header fields compared to that of version 0.5. Notably, version 1.0 replaces the queue congestion status INT instruction with level 2 egress and ingress port IDs. INTCollector provides implementations for both the 1.0 and 0.5 version of the INT and Telemetry specifications. The P4 INT implementation was decided to support INT headers with the newer 1.0 version of the INT and Telemetry specification. The INT header specification [13] specifies several types of encapsulations, e.g. INT over VXLAN and INT over Geneve. INT over UDP/TCP was
chosen for INT header encapsulation since they are the most common protocols. Figure 3.3 shows all the headers contained in the final telemetry report packet, ordered from the outermost to the innermost headers, that will be sent to the monitor from the INT sink. The main reason for the requirement of the separation between outer and inner headers is due to the fact that telemetry packets should use UDP as transport protocol. The original transport headers are therefore encapsulated as inner headers.

3.2.5 Event Detection Algorithms

The purpose of the event detection algorithms is to reduce the amount of INT reports at the INT sink that are sent to the INT collector instance. This is an important task since INT gives information with a packet-level granularity, which quickly impair the performance of the collector if it is required to process all of the INT packets. It is also worth considering
that all INT data is not worth collecting. Many of the metrics from INT data does not change much over time and therefore provide no interesting information. If we, for example, are interested in monitoring the latency in the network, it is mainly interesting to see when the latency starts to increase of decrease significantly. We can therefore detect when these events happen, e.g. latency spikes, and only collect metrics that contribute to these events.

INTCollector provides two categories of event detection algorithms for filtering among INT data, known as interval and threshold. Both the threshold and interval algorithms reduce the quantity of collected metrics, in the former case with thresholds and the latter with interval periods. The threshold algorithms have the advantage over interval algorithms in terms of the quality of the collected metrics. A higher quality of metrics here being metrics that gives better information about how the values of the metrics change over time. In contrast, the interval algorithm simply sends metrics at intervals, even if these metrics provides unnecessary information and may miss important changes in the metric values. The threshold approach sends an INT report if the difference between the current metric of choice and the previous one exceeds a configurable threshold. Thus, only the threshold algorithms were decided to be offloaded to the P4 INT sink.

Three different event detection algorithms with thresholds were chosen for handling event detection in the P4 INT sink: per-hop event threshold, flow event threshold and moving average threshold. The per-hop and flow threshold algorithms are the same algorithms found in the fast path of INTCollector. The choice to include these two algorithms were also taken with consideration in regards to comparison with and without having them offloaded to the P4 INT sink during evaluation. An additional event detection algorithm, moving average threshold, was decided to be included. The use of an average value for comparison means that short-term changes in metric values will not trigger events, since the average of the previous values will smooth them out. The moving average has the advantage of being simple to implement with no larger memory and computation requirements than that of the per-hop and flow event threshold algorithms.
The following terminology will be used in the following sections in order to properly elaborate on the event detection algorithms:

- A metric is defined as the tuple \((\text{ids, } m)\) where \(\text{ids}\) is the identification, e.g. switch id, and \(m\) is a measurement, e.g. hop latency.
- \(t_n\) is the timestamp at time \(n\), where \(t_n > t_{n-1}\) for \(n \in \mathbb{N}\).
- \(V_{\text{ids, } m}(t)\) is the metric value for the metric \(m\) at for id \(\text{ids}\) at time \(t\).
- \(F\) is the set of all switch IDs that are part of the current flow path.

1. **Per-hop Event Threshold** The per-hop event threshold algorithm detects events on a per hop basis. An event is detected if the relation in equation (3.1) holds for any \(\text{swid} \in F\) (any switch in the current flow path), where \(T\) is the threshold value and \(m\) is the metric.

\[
|V_{\text{swid, } m}(t_{n+1}) - V_{\text{swid, } m}(t_n)| > T \tag{3.1}
\]

The per-hop threshold algorithm is sensitive to change in metric values since it compares the threshold for each hop in the flow path. For example, a significant increase in metric value that only occurs for a single packet would produce an event with the per-hop algorithm.

2. **Flow Event Threshold** The sum of the metric values for each hop in the flow is used as the metric value in the flow event threshold algorithm. An event is detected if the relation in equation (3.2) holds.
\[
\left| \sum_{swid \in F} V_{swid,m}(t_{n+1}) - \sum_{swid \in F} V_{swid,m}(t_n) \right| > T
\]  
(3.2)

The flow event threshold algorithm is, just as the per-hop threshold algorithm, sensitive to changes in metric values.

3. **Moving Average Threshold** The moving average threshold algorithm uses an exponential moving average as defined in equation (3.3). For each measurement the moving average \( S_{\text{avg}} \) at \( t_{n+1} \) is set to the value of the metric at \( t_{n+1} \) weighted in with the moving average at the previous time \( t_n \). The \( \alpha \) coefficient decides how much weight the new values will have on \( S_{\text{avg}}(t_{n+1}) \). The absolute value of the difference between the moving average at \( t_{n+1} \) and the current metric value at \( t_{n+2} \) is then compared to the threshold value \( T \) as show in equation (3.4). If the relation in (3.4) holds then an event will be produced.

\[
S_{\text{avg}}(t_{n+1}) = \alpha \sum_{swid \in F} V_{swid,m}(t_{n+1}) + (1 - \alpha) S_{\text{avg}}(t_n)
\]  
(3.3)

\[
|S_{\text{avg}}(t_{n+1}) - \sum_{swid \in F} V_{swid,m}(t_{n+2})| > T
\]  
(3.4)

The moving average threshold will theoretically produce less events compared to the per-hop and flow threshold algorithms. The reason for the lower number of produced events is the influence of the old metric values that are weighted into the comparison. Depending on the value of the \( \alpha \) coefficient chosen, sudden spikes in metric values becomes smoothed out, i.e. has less impact on the moving average value.

4. **Example of Event Detection with the Flow and Moving Average Threshold Algorithms**

An example of event detection with the flow event and moving average threshold
algorithms is depicted in Figure 3.4. The flow latency line shows the latency of the flow and the moving average line shows the current moving average value at each point in time. The algorithms are configured to have a threshold value of 10 ns and the moving average constant $\alpha$ has a value of 0.75. Initially, both algorithms will detect an event since, in this case, the initial difference to compare with will be 0 and 743 ns against a threshold value of 10 ns. Between time 0 and 60 $\mu$s the flow latency varies which causes three events to be detected by the flow threshold algorithm since the difference of the current latency value to that of the last detected event is larger than the threshold. The moving average, on the other hand, smooths out these small variations and only one event is detected in this time span. When it comes to large latency spikes, such as at time 110 and 220 $\mu$s, both algorithms detect events since the latency in both cases greatly exceeds the threshold. The per-hop event threshold algorithm works in the same way as the flow event threshold algorithm illustrated in this section, but applied on a per-hop, i.e. hop latency, basis.
3.2.6 INT Event Detection

The implementation of event detection on the data plane is written in P4 as part of the P4 INT implementation. The event detection logic is located at the INT sink, since it is the INT device that will generate telemetry reports. Each of the three threshold algorithms defined in section 3.2.5 have been implemented in the INT sink. It is assumed that the choice of algorithm will not need to be altered during run-time. The decision of which algorithm to use is therefore defined as a compile time decision. The choice of algorithm could instead have been configurable with a table to dynamically set event detection algorithms, but that would mean additional resource usages since all algorithms would be included in the switch. Since the target hardware has limited resources, doing such design choices is of importance. It is possible to combine the different event detection algorithms by enabling multiple algorithms at the same time. In our case, we will however only activate one algorithm at a time exclusively since we are interested in comparing the algorithms one-by-one. One resource to take into consideration is the limited memory. However, the algorithms intentionally require only small amounts of memory for storing state. Each of the algorithms only need to store state for the values of the previously collected metrics.

Since the event detection now is implemented on a data plane device it is possible to set parameters used in the threshold algorithms as table entries that can be read and written from an external SDN controller. This opens up interesting possibilities for the management of telemetry reporting. Whenever the need for adjusting the rate of telemetry reports sent appears, a SDN controller can adjust the table containing the parameters of the algorithm to control the algorithms behavior and event detection rate. One important parameter that is put into such a table is the threshold parameter. By adjusting this parameter it becomes possible to adjust the sensitivity of the event detection dynamically, with a higher threshold value yielding less events and a lower threshold value more events.
4 Implementation

This section covers the implementation of the normal path with Kafka producer and the
P4 INT implementation with Event detection [7].

4.1 Normal Path with Kafka Producer

The normal path of INTCollector is written in the Python language. Since Python is a high-
level language it is easy to adapt the normal path to use another API processing and storing
INT metrics from the fast path. The kafka-python API[22] is used for communication with
a Kafka cluster instance. Specifically, the normal path uses the Kafka producer module
from kafka-python that provides an API for sending messages to a running Kafka cluster.
Listings [1] shows the essential parts of the normal path involved in the sending of metrics
to Kafka.

```python
from kafka import KafkaProducer
# (37 lines omitted)
def _event_push():
    producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
                            value_serializer=lambda x:
                            json.dumps(x).encode('utf-8'))

    while not push_stop_flag.is_set():
        time.sleep(args.event_period)

        collector.lock.acquire()
        data = collector.event_data
        collector.event_data = []
        collector.lock.release()

        for d in data:
            future = producer.send('headers', value=d)
```

Listing 1: Normal path Kafka producer in Python.
As explained in section 3.1.1, the fast path will signal the normal whenever new events arrive and store the metrics in a shared table from which the normal path can access the telemetry data. The normal path remains idle for \textit{event\_period} seconds before entering a critical section where it checks the state of the shared \texttt{event\_data} variable (lines 11-14). This variable contains collected metrics from the fast path, which if non-empty, is sent to the "headers" topic in the Kafka cluster via the Kafka producer API (lines 16-17).

```
1 def _process_event(ctx, data, size):
2     cdef uintptr_t _event = <uintptr_t> data
3     cdef Event *event = <Event*> _event
4     # (12 lines omitted)
5     event_data = {
6         "src_ip" : self.int_2_ip4_str(event.src_ip),
7         "src_port" : event.src_port,
8         "dst_ip" : self.int_2_ip4_str(event.dst_ip),
9         "dst_port" : event.dst_port,
10        "protocol" : event.ip_proto,
11        "flow_latency" : event.flow_latency,
12        "flow_sink_time" : event.flow_sink_time,
13        # (6 lines omitted)
14     }
15
16     for i in range(0, event.num_INT_hop):
17         event_data["sw\{0\}_sw_ids\".format(i)] = event.sw_ids[i]
18         event_data["sw\{0\}_in_port_ids\".format(i)] = event.in_port_ids[i]
19         event_data["sw\{0\}_e_port_ids\".format(i)] = event.e_port_ids[i]
20         event_data["sw\{0\}_hop_latencies\".format(i)] =
21             event.hop_latencies[i]
22         event_data["sw\{0\}_queue_ids\".format(i)] = event.queue_ids[i]
23         event_data["sw\{0\}_queue_occups\".format(i)] = event.queue_occups[i]
24         event_data["sw\{0\}_ingr_times\".format(i)] = event.ingr_times[i]
25         event_data["sw\{0\}_egr_times\".format(i)] = event.egr_times[i]
26         event_data["sw\{0\}_queue_congestion\".format(i)] =
27             event.lv2_in_e_port_ids[i]
28         event_data["sw\{0\}_tx_utilizes\".format(i)] = event.tx_utilizes[i]
```

Listing 2: Normal path metric collection in Cython.
The event_data variable is updated in the manner shown in Listings 2. This part of the normal path reads event data into user space from the fast path and is written in Cython [9]. Cython is a programming language that provides the syntax of Python and compiles the program down to C. It is thus possible to write code with higher performance in Cython compared to Python and to easily integrate external C code. Whenever a new event is signaled from the fast path, the event data will be available as a pointer to a C structure (line 3). The pointer to the event data is converted into a dictionary data type. Lines 5-14 read the flow information and lines 16-26 read the INT metadata for each switch and add it to the dictionary. The data for each switch has to be stored as separate fields in the dictionary and not in an array. This is not a limitation in the normal path per se, it is instead a limitation in Kibana which has no support for fields that are stored as arrays, and therefore the data has to be represented in a flattened structure. The event data then becomes available to be read by the code in Listings 1 which will be converted by the kafka-API into JSON format before it is sent to the Kafka cluster.

4.2 P4 INT with Event Detection

The flow chart in figure 4.1 gives a high level illustration of how INT packets are processed by the P4 implementation. The ingress section implements a basic forwarding mechanism for IPv4 packets. Each packet received by the Ingress section are matched in a forwarding table. A packet is forwarded if any forwarding rule matches the IP addresses of the packet. Otherwise the packet is dropped at the ingress. The forwarding is performed by updating the packet header with the new MAC destination and source addresses. The packet then enters the egress pipeline. The egress pipeline of the P4 implementation contains the main INT logic. It is implemented to cover the necessary operations required by an INT device. This includes adding the initial INT headers, adding INT metadata during the in-transit phase, and finally to prepare and send a telemetry report packet for the INT monitor. It also includes the event detection logic, which implements the algorithms defined in section
Figure 4.1: P4 INT flow chart.
3.2.5

4.2.1 Headers and Constants

```c
header int_hdr_t {
    bit<4> ver;
    bit<2> rep;
    bit<1> c;
    bit<1> e;
    bit<1> m;
    bit<7> rsvd1;
    bit<3> rsvd2;
    bit<5> hop_m_len;
    bit<8> remain_hop_cnt;
    bit<16> instruction_mask;
    bit<16> reserved;
}
```

Listing 3: INT header defined in P4.

All the headers as seen in Figure 3.3 are defined in the P4 implementation with the P4 header construct. An example of the INT header defined in the implementation is given in Listing 3. The only exception is the definition of the INT metadata stack. INT metadata is the data added at each switch, as instructed by the INT instruction mask seen on line 11 in Listing 3. INT metadata is defined as a P4 header stack, i.e. as a stack data structure of INT metadata. The header stack is declared by using an array notation:

```
int_metadata_t[MAX_HOP_COUNT*NUM_INSTRUCTIONS] int_metadata;
```

The int_metadata_t is data type defined as a 32 bit header, which is the size for a single INT metadata. The MAX_HOP_COUNT and NUM_INSTRUCTIONS constants define the maximum number of hops along the flow path and the number of INT data types to embed into the packet header respectively. The values of these constants are constrained since the
P4 SDK puts a limit to the size of the header stack and to the total size of header data. Our target specifies a limit of approximately 820 bytes for combined header sizes and a limit of 16 headers in a header stack. The maximum number of hops and INT instructions therefore have to be adjusted with this in mind. With a total amount of 3 hops, we can store INT metadata for a maximum of 5 different INT instructions. The additional size added by INT headers may also cause trouble with the MTU size limit. This can be solved by increasing the MTU of the links to accommodate extra room needed for the INT headers.

### 4.2.2 Collection of INT metadata

Switch information is read from special P4 structures known as intrinsic metadata. Intrinsic metadata is metadata that has special significance to the operation of the target. A set of default intrinsic metadata is defined in standard metadata which contain mandatory information of packets, e.g. ingress port, egress port, and packet length among others. P4 targets are allowed to define their own fields of intrinsic metadata that is specific for the targets architecture. The architecture targeted by this implementation has support for the following non-standard intrinsic metadata fields:

- **ingress_global_timestamp**
- **current_global_timestamp**

The *ingress_global_timestamp* metadata contains the timestamp at the packet ingress of the MAC chip and *current_global_timestamp* contains the current timestamp in the MAC block.

The values of these fields are obtainable in P4 by defining it as a struct as shown in Listings 4. This set of intrinsic metadata is somewhat limited since it does not completely cover all of the available INT instructions. The INT v1.0 specification gives 8 different instructions of INT data that can be collected. The list of instructions, given in the
struct intrinsic_metadata_t {
  bit<64> ingress_global_timestamp;
  bit<64> current_global_timestamp;
}

Listing 4: Intrinsic metadata.

"INT data" column, and the corresponding available P4 primitives, given in the "P4 data" column, are given in table 4.1. The INT instructions lacking P4 primitives are marked as "not implemented". Most notable is the lack of queuing data available from the P4 architecture. P4 metadata for the ingress port, egress port and ingress timestamp are on the other hand readily available. The INT data for switch id is obtained through a table lookup. The hop latency for a packet is defined as the time the packet spends being switched inside the device. The value of current_global_timestamp is read at the egress, which when subtracted by ingress_global_timestamp approximately gives the hop latency. Note that current_global_timestamp is the time spent in the device without queuing, since queuing in this device architecture occurs after the egress.

4.2.3 Parsing

The parser, which parses out header data of incoming packets, is implemented as shown in figure 4.2. The starting state of the parser starts at the Ethernet state and continues through all the headers as specified by INT over UDP/TCP [14]. Since the transport protocol can be either TCP and UDP, the transition after the IPv4 state will depend on the value of the protocol field in the IPv4 header. The value of the destination port number of the UDP packet decides if the packet is an INT packet, and to transition to parser inner Ethernet. The UDP destination port is configurable, but must be the same for all telemetry reports. Since all packets in our case are treated as INT packets, the port check is unnecessary, but if we would wish to only apply INT to certain flows or types this check helps us to filter them out. The telemetry header is included in the parsing
Table 4.1: INT instructions with corresponding P4 data.

<table>
<thead>
<tr>
<th>Bit (MSB)</th>
<th>INT data</th>
<th>P4 data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Switch ID</td>
<td>Table lookup</td>
</tr>
<tr>
<td>1</td>
<td>Level 1 Ingress &amp; Egress port ID</td>
<td><code>standard_metadata.ingress_spec</code> and <code>standard_metadata.egress_spec</code></td>
</tr>
<tr>
<td>2</td>
<td>Hop latency</td>
<td><code>current_global_timestamp - ingress_global_timestamp (at egress)</code></td>
</tr>
<tr>
<td>3</td>
<td>Queue ID and occupancy</td>
<td>Not implemented</td>
</tr>
<tr>
<td>4</td>
<td>Ingress timestamp</td>
<td><code>ingress_global_timestamp</code></td>
</tr>
<tr>
<td>5</td>
<td>Egress timestamp</td>
<td><code>current_global_timestamp (at egress)</code></td>
</tr>
<tr>
<td>6</td>
<td>Level 2 Ingress &amp; Egress port ID</td>
<td>Not implemented</td>
</tr>
<tr>
<td>7</td>
<td>Egress port Tx utilization</td>
<td>Not implemented</td>
</tr>
</tbody>
</table>

Figure 4.2: P4 INT parser.
process since all headers used in the P4 program are required to be included in the parser. After parsing the inner headers the parser will reach the state where it needs to parse INT metadata. The INT metadata stack however contains a variable amount of INT metadata. This is solved by adding a metadata for the parser that keeps track of the current number of parsed INT metadata headers. When the number of INT metadata reaches zero the parsing process is finished and transitions to the accept state. Packets that are accepted by the parser are passed to the ingress pipeline with all the parsed header fields available for read and write access.

4.2.4 Ingress and Egress Pipeline

The first check done by the egress pipeline is to see if the packet is a clone. Packet cloning is achieved with the `clone()` operation in P4, which will create a copy of the current packet when it reaches the egress and send the clone of the packet into the ingress. Cloning is needed since the original packet must still be sent at the INT sink without the INT headers, while the clone can be sent with the accumulated INT data to a collector. If the packet is not a clone, then it means that the arriving packet has not gone through the INT sink yet. If no initial INT header is found it is added before adding the INT data for the current device. The actions for inserting the INT data are called corresponding to the value of the INT instructions set in the initial INT header. An example for adding INT data is the action for adding hop latency as INT data shown in Listings 5. Since INT data is stored as a header stack, the header stack must call the `push_front` method. This pushes an empty space to the front of the header stack that can be used for storing 32 bits of INT data. If the device is not the INT sink, then the packet can be passed to the next hop.

The event detection mechanism needs to be able to identify metrics, e.g. by switch or by flow identification, to allow the event detection to be applied on a per flow basis. The implementation uses a matching table to acquire the flow id, which is defined as a five tuple: (IPv4 source address, IPv4 destination address, IPv4 protocol, UDP/TCP source
The P4 code for the per-hop latency threshold algorithm is shown in Listings 6. Lines 3-5 define registers containing the last hop latency for each switch. This is followed by three if-statements at lines 11-30 which check if the new hop latency value exceeds the threshold. The \texttt{ABS\_DIFF} macro, defined on line 1 and applied in the threshold comparisons, takes the absolute value of the difference of two values. This macro works for unsigned integers since it always subtracts the larger with the smaller value. If the absolute difference between the old value \texttt{prev\_hop\_latency} and \texttt{cur\_hop\_latency} exceeds the threshold then the \texttt{meta.flow_metadata.is\_update} is set to true, which means that an event is detected. P4 lacks support for iteration statements and each use of the threshold check therefore has
```c
#define ABS(x,y) (((x) < (y)) ? (y)-(x) : (x)-(y))

register<bit<32>>(NUM_FLOWS) reg_port_hop_lat_swid1;
register<bit<32>>(NUM_FLOWS) reg_port_hop_lat_swid2;
register<bit<32>>(NUM_FLOWS) reg_port_hop_lat_swid3;
bit<32> prev_hop_latency = 0;
bit<32> cur_hop_latency = 0;

bit<32> flow_id = meta.flow_metadata.id;

reg_port_hop_lat_swid1.read(prev_hop_latency, flow_id);
cur_hop_latency = hdr.int_metadata[HOP_LATENCY_INDEX].data;
if (ABS_DIFF(prev_hop_latency, cur_hop_latency) > meta.flow_metadata.thresh) {
    meta.flow_metadata.is_update = true;
    reg_port_hop_lat_swid1.write(flow_id, cur_hop_latency);
}

reg_port_hop_lat_swid2.read(prev_hop_latency, flow_id);
cur_hop_latency = hdr.int_metadata[HOP_LATENCY_INDEX + NUM_INSTRUCTIONS].data;
if (ABS_DIFF(prev_hop_latency, cur_hop_latency) > meta.flow_metadata.thresh) {
    meta.flow_metadata.is_update = true;
    reg_port_hop_lat_swid2.write(flow_id, cur_hop_latency);
}

reg_port_hop_lat_swid3.read(prev_hop_latency, flow_id);
cur_hop_latency = hdr.int_metadata[HOP_LATENCY_INDEX + NUM_INSTRUCTIONS*2].data;
if (ABS_DIFF(prev_hop_latency, cur_hop_latency) > meta.flow_metadata.thresh) {
    meta.flow_metadata.is_update = true;
    reg_port_hop_lat_swid3.write(flow_id, cur_hop_latency);
}
```

Listing 6: Per hop latency threshold.
register<bit<32>>(NUM_FLOWS) reg_latency_sum;
bit<32> cur_latency_sum = 0;
bit<32> prev_latency_sum = 0;
bit<32> flow_id = meta.flow_metadata.id;

cur_latency_sum = hdr.int_metadata[HOP_LATENCY_INDEX].data +
                hdr.int_metadata[HOP_LATENCY_INDEX+NUM_INSTRUCTIONS].data +
                hdr.int_metadata[HOP_LATENCY_INDEX+NUM_INSTRUCTIONS*2].data;

reg_latency_sum.read(prev_latency_sum, flow_id);

if (ABS_DIFF(prev_latency_sum, cur_latency_sum) > meta.flow_metadata.thresh) {
    meta.flow_metadata.is_update = true;
    reg_latency_sum.write(flow_id, cur_latency_sum);
}

Listing 7: Flow latency threshold.

register<bit<32>>(NUM_FLOWS) reg_avg;
bit<32> flow_id = meta.flow_metadata.id;
bit<32> t_prev = 0;
bit<32> t_avg = 0;
bit<32> t_cur = 0;

    t_cur = hdr.int_metadata[HOP_LATENCY_INDEX].data +
            hdr.int_metadata[HOP_LATENCY_INDEX+NUM_INSTRUCTIONS].data +
            hdr.int_metadata[HOP_LATENCY_INDEX+NUM_INSTRUCTIONS*2].data;

reg_avg.read(t_prev, flow_id);
if (ABS_DIFF(t_prev, t_cur) > meta.flow_metadata.thresh) {
    meta.flow_metadata.is_update = true;
}
    t_avg = (3 * (t_prev >> 2)) + (t_cur >> 2);
reg_avg.write(flow_id, t_avg);

Listing 8: Moving average threshold.
to be defined in its own if-statement. This could be accomplished in an external function in micro C (a subset of C designed for use in microcontrollers) where iteration statements are available. But since we only deal with 3 switches this was not a priority issue.

Listings 7 shows the implementation of the flow latency threshold algorithm. In this case, the sum of the hop latencies are used in the threshold comparison. This works in the same manner as in the per-hop threshold case, but instead of comparing each hop latency, we sum the hop latencies (line 6) and use the hop latency sum for comparison with the threshold (line 10).

The implementation of the moving average threshold algorithm is shown in Listings 8. Line 1 defines a register for the moving average flow latency sum values. The current metric value for the flow latency is calculated on line 7. The threshold comparison is done on lines 10-12 similarly as in the previous Listings 6 7. Finally, at line 13 the new moving average flow latency is calculated, with the \( \alpha \) coefficient set to 0.75, and stored in the register on line 14. Note that bit-shifting is used on line 13 since this target lacks floating point arithmetic.

The threshold code in Listings 6, 7 and 8 only do event detection based on hop latency. INTCollector additionally checks queue occupancy and queue congestion status. However, as shown earlier in table 4.1, the architecture of the implementation has no support for collecting queue related data. Thus only hop latency data is used for metrics in the event detection implementation.

5 Evaluation

In this section the performance of the INT monitor is evaluated and comparisons between the different threshold algorithms are made. Section 5.2 evaluates the processing rate of the INT monitor. This tells us at what rate INTCollector is capable of processing telemetry packets before it starts to drop packets. The rate at which INTCollector can process INT
packets without loss will then be used as the sending rate for the other tests. Section 5.3 evaluates the number of events produced and the CPU usage for each of the three threshold algorithms with and without the offloading of event detection. This will show if the offloading does lead to less CPU usage at the INT monitor or not. If the event detection mechanism in the INT implementation works as intended can be verified by comparing the event ratio for the INT with that of the INTCollector implementation. How much more INT traffic we potentially can send based on the increase of the threshold value is covered as well. Section 5.4 compares the event ratio for each of the different algorithms when implemented at the INT sink. The comparison will give an idea of how the event detection of each algorithm behaves in relation to each other.

5.1 Evaluation Setup

The evaluation setup, as seen in Figure 5.1, consists of a topology with three switches and two end-hosts. Each switch has a Netronome Agilo SmartNIC with the P4 INT implementation from section 4.2 loaded into it. The SmartNIC cards in the switches each has two 40 Gpbs ports, which are split into 4 separate 10 Gpbs ports in a breakout module. The two end-hosts in the topology of the evaluation setup consists of a packet generator and an INT monitor. The INT monitor runs with the operating system Ubuntu GNU/Linux 4.14.105 with 32G RAM and a 20 core Intel Xeon Silver 4114 CPU at 2.20GHz. The INT monitor is equipped with a 4× dual SFP+ 10 Gbps Intel NIC that is connected to the INT network. The packet generator runs OSNT and sends packets from a given set of pcap files over a NetFPGA card. A total of 4 pcap files, one for each interface on the NetFPGA card, are used for the evaluation. Each pcap file is generated with a Scapy Python script and contains 64 different TCP flows giving a total of 256 different flows sent to the INT monitor. The pcap packets are standard TCP over IP and Ethernet packets without any payload. The INT source is configured to enable three INT instructions: switch ID, hop latency and ingress timestamp. The total size of the telemetry packets therefore becomes
no more than 172 bytes. The P4 INT sink uses a round robin algorithm for sending INT report packets evenly across each port at the INT monitor, thus keeping an even load on the separate INTCollector instances.

Figure 5.1: Setup used for evaluation.
5.2 INT Packet Processing Rate

Figure 5.2 shows the percentage of packets that are lost at the four collectors as the total transfer rate in packets per second (pps) is varied from 0.5 Mpps upto 31 Mpps. No event detection mechanism was active at the INT sink nor at the INTCollector during this test. All packets therefore were sent to the INTCollector which parsed the packets and then dropped the collected metrics directly at the normal path, i.e. without any Kafka and Elastic Stack processing. Packets got processed without any loss at the INT monitor until the data transfer rate surpassed 1 Mpps. The transfer rate 1 Mpps therefore became the sending rate for the rest of the evaluation phase since it is the highest transfer rate at which the INT monitor can process INT packets without any loss. This is needed for giving accurate measurements of event detection in the evaluation tests, where dropped packets otherwise could be mistaken for not meeting event detection criteria. Sending at higher rates than 1 Mpps means that we will lose metric data, e.g. 83% at 31 Mpps, which would give us a insufficient representation of events detected.

The capabilities of XDP on its own was tested as well with a simple XDP program that parses the IP header of each packet, drops the packet and increases a counter. The counter is checked each second from user pace to measure the rate in packet per second. At full capacity a total rate of 3 Mpps without loss was measured across all 4 XDP instances.

5.3 Average CPU Usage, Event Offloading Ratio and Throughput

Each of the threshold algorithms were evaluated for a set of threshold values in regards to average CPU usage, the number of events detected and the possible throughput. The average CPU usage was measured as the average of the CPU usage taken over intervals of 20 seconds. The number of produced events are measured as event ratio, which is defined
as the ratio of packets that generate events to the total number of packets processed:

\[
\text{event ratio} = \frac{\text{events detected}}{\text{total packets received}}
\]

The potential throughput is derived from the event ratio. For example, an event ratio of 50% would allow the sending rate to be doubled (since only half of the packets will generate telemetry packets that are sent to the INT monitor). An increased threshold value therefore allows a higher sending rate, which eventually will allow the sender to send at line rate without overloading the INT monitor. The tests were made with a sending rate of 1 Mpps in total across the ports of the OSNT packet generator. The INT source was configured to use three INT instructions, i.e. to collect three INT metrics at each switch. The threshold value for the algorithms, which is readily accessible as a table value in the data plane, was varied from 0 ns to 60 ns. At 0 ns all events will exceed the threshold value and generate telemetry reports, which is equivalent as the case without any event detection mechanism. As the threshold value increases less events will be detected and will eventually stop triggering events, which in this test setup was found to be after 60 ns.

Figure 5.2: Fast path packet loss count plotted against packet sending rate (Mpps).

Figure 5.3 and 5.4 show CPU usage and event ratio of the per-hop and per-flow event
detection algorithms respectively for four different setups. The setups vary by the use of event pre-filtering at the P4 INT switch and by the inclusion of the Kafka producer normal path. In the case when event pre-filtering is enabled the event detection logic is enabled at the INT sink while the event detection at INTCollector has a constant threshold value of zero. The threshold code in INTCollector was not removed since setting the threshold to zero has the same effect. The threshold algorithm is disabled in INT sink if the pre-
filtering is disabled, but enabled at the INTCollector, i.e. the INTCollector threshold is varied to allow event detection. The pre-filtering case is used to compare the setup when the event detection is offloaded to the data plane vs. the original setup where it is part of INTCollector. The exclusion of the Kafka producer normal path means that metric data from the fast path is dropped directly when it reaches the normal path. Therefore when the metric data is read from the XDP table into user space the Python code will
immediately discard the data instead of calling the Kafka producer API. By excluding the normal path we can compare the performance when the whole setup is used (normal path enabled), i.e. with INTCollector, Kafka and Elastic Stack, compared to only INTCollector (normal path disabled).

Specifically, each of the four test cases are as follows:

1. **Event pre-filtering enabled and normal path enabled.** Event detection algorithms is enabled at the P4 INT sink switch which filter out events directly at the data plane. Only INT packets that produce events will in this case reach INTCollector in the INT monitor, which will only have to parse the INT data from the telemetry reports. The normal path is enabled, which means that the INT data will be read into user space, sent via the Kafka producer API to the Kafka cluster and finally be stored in the Elastic Stack.

2. **Event pre-filtering disabled and normal path enabled.** In this case the event pre-filtering is disabled, which means that the event detection is handled at INTCollector instead of at the P4 INT sink switch. INTCollector thus becomes responsible to perform event detection on all packets from the INT sink and to parse the INT data. The normal path is as described in test case 1.

3. **Event pre-filtering enabled and normal path disabled.** Enabled event pre-filtering works as described in test case 1. A disabled normal path means that no Kafka or Elastic stack processing is involved. The normal path simply reads the INT data from XDP and discards it immediately.

4. **Event pre-filtering disabled and normal path disabled.** No pre-filtering is as described in test case 1 and no normal path is as described in test case 3.

The evaluation of the per-hop and flow event threshold algorithms show similar results. Graph (a) in Figure 5.3 and 5.4 shows the measurements of the average CPU load for each
of the 4 setups. The square and diamond shaped lines show the average CPU load when the normal path is enabled, while the diamond and circle shaped lines below are for the cases without the normal path. The extra CPU load when the normal path is enabled is due to the processing done by the Kafka cluster and Elastic Stack which require extra CPU usage. When the event pre-filtering is enabled there is less CPU usage compared to when the pre-filtering is disabled. Graph (b) in Figure 5.3 shows the event ratio for the per-hop algorithm. In this case the event ratio quickly dips between 5 and 10 ns. The per-flow algorithm in contrast, shown in graph (b) in Figure 5.4, has a more smooth event ratio line. The flow algorithm is more smooth since it looks at the sum of the hop latencies, while the per-hop looks at individual hops and therefore is more sensitive to change and will trigger events more easily. Finally, the maximal potential throughput that the packet generator can send without overloading the INT monitor is shown in graph (c). The per-hop algorithm allows more potential throughput as the threshold value is increased since the per-hop algorithm filter out events more effectively. Eventually both algorithms reach line rate, which is 30 Mpps. The max limit is reached by the per-hop algorithm when the threshold is configured to be 30 ns (except for one test case which reach it at 40 ns). The flow algorithm reaches the max limit at little later, namely at the 40 ns threshold value. Without event detection it is therefore only possible to send 1 Mpps from the packet generator, but if the event detection is enabled and has a threshold value of 40 ns or more it becomes possible to send at line-rate.

Figure 5.5 shows the average CPU usage, event ratio and possible throughput with different threshold values for the moving average algorithm. The test case with event pre-filtering disabled was not measured for the moving average since INTCollector does not implement the moving average algorithm. Instead, only two test cases were involved: enabled and disabled normal path. The moving average with normal path line in graph (a) Figure 5.5 is evaluated with the normal path enabled, i.e. with Kafka and Elastic Stack, and with the event detection in the INT sink, but not at the INTCollector. The second
Figure 5.5: Measured average CPU usage and event ratio for moving average algorithm.

line has event detection at the INT sink, but has the normal path disabled, i.e. packets are dropped in the normal path instead of being sent to Kafka and Elastic Stack. The graphs (b) and (c) only include one line, with normal path, since both are evaluated at the INT sink there is no need to measure the non-normal path case which yields essentially the same event ratio data. The potential throughput is depicted in graph (c) and shows that line-rate for throughput was achieved when the threshold value reached 30 ns. How different values of $\alpha$ affects the event ratio for the moving average algorithm is shown in
Figure 5.6: Event detection ratio for different values of $\alpha$ for moving average algorithm.

The tabularized values of the measurements for the hop latency, flow latency and moving average algorithms can be found in Appendix A.

### 5.4 Event Detection Algorithm Comparison

Figure 5.7 shows a comparison of the event detection ratio for the per-hop, flow and moving average threshold algorithms for different values of the threshold value. This test case was run with event detection located at the INT sink. Since only the behavior of the event detection was of interest the test cases were run without any INT monitor involved. Since no INT monitor was involved the tests could be run at line-rate. The pcap files used for generating traffic were the same as in earlier tests. The results show how the per-hop latency line has a high initial threshold, but then quickly dips. The flow latency and moving average are more smooth, with the moving average having a much lower event ratio. The reason for the lower event ratio of the moving average algorithm is most likely
being due to how the moving average smooths out small and sudden latency spikes (which can be seen in the algorithm example in Figure 3.4).

6 Discussion

The goal of this thesis was to investigate the benefits of offloading the event detection from INTCollector to an INT sink. The main benefit of the event detection offloading was assumed to be improved performance of the INT monitor since packets would be pre-filtered at the INT-sink before reaching the INT monitor. The implementation of the offloading of the event detection mechanism to INT proved itself to be possible. That the event detection algorithms in the INT implementation works as intended can be inferred from the close match of the event ratio curves in Figure 5.3 and 5.4. The results in figure 5.3, 5.4 and 5.5 also suggest that INTCollector benefits in terms of performance since in the case when offloading is in use the CPU usage level is lower. The result that the CPU load
at the INT monitor decreases with event detection offloading does, on the other hand, not take the extra load put on the switch into consideration. However, the extra load imposed on the switch is likely negligible. The event detection mechanism at the INT sink requires a few extra instructions for threshold comparison, but at the same time the sink will not have to do the copy operation for every packet which is a expensive operation to perform.

The comparison of the event detection algorithms, as presented in section 5.4, shows a differences in the amount of events they produce in relation to the threshold value. This finding shows that the choice of algorithm has an impact on the distribution of the number of events generated as a function of the threshold value. It also suggests that using average values has a smoothing effect on the variance of latency values. This leads the moving average algorithm to produce less events compared to the algorithms that compare the current value to the previously stored value.

The sending rate at which INTCollector could receive INT reports without loss was found to be limited to 1 Mpps. Tests made with a simple XDP packet counter program was, in contrast, able to receive packets at 3 Mpps. Several factors contribute to the comparable decrease in throughput, such as the delay cause by the instructions in the XDP program, the pace that the normal path reads from the XDP tables and the time taken by the Kafka processing. Each message sent to the Kafka cluster imposes a delay since the producer API waits for an acknowledgment for messages sent. Another aspect is the fact that XDP only utilizes one core. This was partly compensated by running four instances of INTCollector at the destination host instead of attaching a single INTCollector instance to all interfaces. The test machine had 20 cores and with 4 INTCollectors only 4 of the twenty cores were used. This can be seen in the graphs, e.g. Figure 5.3 and 5.4 (a), where the case without normal path only reaches 20% total CPU usage without event pre-filtering (100% CPU usage at 4 out of 20 cores gives a total average CPU usage of 20%).

The results from the evaluation can be improved upon further. The test cases do,
for example, not explore factors such as the number of flows and the number of INT instructions used. Further work would include larger test suites to cover several aspects that could impact the performance. Moreover, there are more options to be explored when it comes to the implementation of the event detection, such as the design and combination of event detection algorithms. The normal path implementation with Kafka, with the goal of achieving high throughput, was not properly evaluated. Work could be done in exploring the advantages in a distributed approach with Kafka. The normal path improvement was however not the main objective of the thesis and more focus was put into the performance advantages in the P4 INT fast path offloading. The results in this thesis have in the end been sufficient to show that there are benefits to gain from offloading event detection and hopefully this will encourage further work on event detection in the data plane.

7 Conclusion

This thesis presented an implementation of INT in P4 which offloads the event detection mechanism from INTCollector to the data plane with the aim to reduce the load on the stream processing entity. In addition, three threshold algorithms were implemented at the INT sink to examine the impact different algorithms have on the ratio of produced events. Based on the results of the evaluation, it can be concluded that offloading the event detection reduces the CPU usage of INTCollector. Furthermore, the choice of algorithm were shown to impact the number of events produced, with each approach showing different distributions of the ratio of events in relation to the threshold value. Depending on the threshold value, the throughput to the stream processor was reduced and it became possible to send more INT traffic over the INT sink without overloading the INT monitor with telemetry reports. Offloading the event detection to the data plane additionally comes with the advantage of making parameters of the algorithms accessible to the control plane which can control the behavior of the event detection. Further work is still required with
the evaluation of offloading in order to fully understand how different factors may affect the performance. All the possibilities with event detection on the data plane was only covered partly in this thesis and future research, e.g. in the design of event detection algorithms, is still open.

References


# Evaluation data tables

Table A.1: Average CPU usage and number of events detected per hop with pre-filtering and normal path enabled.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of detected events</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4800000</td>
<td>31.20</td>
</tr>
<tr>
<td>5</td>
<td>4245822</td>
<td>30.27</td>
</tr>
<tr>
<td>10</td>
<td>2074157</td>
<td>27.99</td>
</tr>
<tr>
<td>20</td>
<td>299576</td>
<td>23.70</td>
</tr>
<tr>
<td>30</td>
<td>96708</td>
<td>22.35</td>
</tr>
<tr>
<td>40</td>
<td>86913</td>
<td>18.10</td>
</tr>
<tr>
<td>50</td>
<td>23241</td>
<td>13.53</td>
</tr>
<tr>
<td>60</td>
<td>14</td>
<td>5.63</td>
</tr>
</tbody>
</table>

Table A.2: Average CPU usage and number of events detected per hop with pre-filtering enabled and normal path disabled.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of detected events</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4800000</td>
<td>19.80</td>
</tr>
<tr>
<td>5</td>
<td>4218373</td>
<td>18.83</td>
</tr>
<tr>
<td>10</td>
<td>1896738</td>
<td>16.35</td>
</tr>
<tr>
<td>20</td>
<td>266862</td>
<td>6.84</td>
</tr>
<tr>
<td>30</td>
<td>63855</td>
<td>2.56</td>
</tr>
<tr>
<td>40</td>
<td>54705</td>
<td>2.73</td>
</tr>
<tr>
<td>50</td>
<td>51219</td>
<td>2.59</td>
</tr>
<tr>
<td>60</td>
<td>214</td>
<td>0.62</td>
</tr>
</tbody>
</table>
Table A.3: Average CPU usage and number of events detected per hop without pre-filtering enabled and normal path disabled.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of detected events</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4800000</td>
<td>32.10</td>
</tr>
<tr>
<td>5</td>
<td>3773594</td>
<td>31.86</td>
</tr>
<tr>
<td>10</td>
<td>1261426</td>
<td>30.16</td>
</tr>
<tr>
<td>20</td>
<td>334303</td>
<td>27.82</td>
</tr>
<tr>
<td>30</td>
<td>100022</td>
<td>27.02</td>
</tr>
<tr>
<td>40</td>
<td>91849</td>
<td>27.56</td>
</tr>
<tr>
<td>50</td>
<td>64201</td>
<td>24.03</td>
</tr>
<tr>
<td>60</td>
<td>829</td>
<td>10.95</td>
</tr>
</tbody>
</table>

Table A.4: Average CPU usage and number of events detected per hop with both pre-filtering and normal path disabled.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of detected events</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4800000</td>
<td>20.10</td>
</tr>
<tr>
<td>5</td>
<td>3888835</td>
<td>19.47</td>
</tr>
<tr>
<td>10</td>
<td>1565746</td>
<td>18.34</td>
</tr>
<tr>
<td>20</td>
<td>326676</td>
<td>12.17</td>
</tr>
<tr>
<td>30</td>
<td>173557</td>
<td>7.88</td>
</tr>
<tr>
<td>40</td>
<td>84920</td>
<td>6.02</td>
</tr>
<tr>
<td>50</td>
<td>61047</td>
<td>5.48</td>
</tr>
<tr>
<td>60</td>
<td>938</td>
<td>3.54</td>
</tr>
</tbody>
</table>
### Table A.5: Average CPU usage and number of events detected per flow with pre-filtering and normal path enabled.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of detected events</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4800000</td>
<td>33.09</td>
</tr>
<tr>
<td>5</td>
<td>3599883</td>
<td>31.02</td>
</tr>
<tr>
<td>10</td>
<td>2203186</td>
<td>28.11</td>
</tr>
<tr>
<td>20</td>
<td>604410</td>
<td>27.30</td>
</tr>
<tr>
<td>30</td>
<td>225580</td>
<td>25.78</td>
</tr>
<tr>
<td>40</td>
<td>26957</td>
<td>23.04</td>
</tr>
<tr>
<td>50</td>
<td>18252</td>
<td>15.29</td>
</tr>
<tr>
<td>60</td>
<td>1488</td>
<td>6.26</td>
</tr>
</tbody>
</table>

### Table A.6: Average CPU usage and number of events detected per flow with pre-filter disabled and normal path enabled.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of detected events</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4800000</td>
<td>34.13</td>
</tr>
<tr>
<td>5</td>
<td>3210424</td>
<td>32.18</td>
</tr>
<tr>
<td>10</td>
<td>2212919</td>
<td>30.92</td>
</tr>
<tr>
<td>20</td>
<td>674967</td>
<td>31.21</td>
</tr>
<tr>
<td>30</td>
<td>168312</td>
<td>30.23</td>
</tr>
<tr>
<td>40</td>
<td>85789</td>
<td>29.10</td>
</tr>
<tr>
<td>50</td>
<td>56043</td>
<td>27.30</td>
</tr>
<tr>
<td>60</td>
<td>9804</td>
<td>19.49</td>
</tr>
</tbody>
</table>
Table A.7: Average CPU usage and number of events detected per flow with both pre-filtering and normal path disabled.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of detected events</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4800000</td>
<td>19.90</td>
</tr>
<tr>
<td>5</td>
<td>4001281</td>
<td>19.62</td>
</tr>
<tr>
<td>10</td>
<td>2140928</td>
<td>17.82</td>
</tr>
<tr>
<td>20</td>
<td>611310</td>
<td>14.73</td>
</tr>
<tr>
<td>30</td>
<td>217266</td>
<td>9.88</td>
</tr>
<tr>
<td>40</td>
<td>151035</td>
<td>7.56</td>
</tr>
<tr>
<td>50</td>
<td>53263</td>
<td>5.27</td>
</tr>
<tr>
<td>60</td>
<td>2812</td>
<td>3.42</td>
</tr>
</tbody>
</table>

Table A.8: Average CPU usage and number of events detected per flow with pre-filtering enabled and normal path disabled.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of detected events</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4800000</td>
<td>19.60</td>
</tr>
<tr>
<td>5</td>
<td>3762673</td>
<td>18.85</td>
</tr>
<tr>
<td>10</td>
<td>2668714</td>
<td>16.25</td>
</tr>
<tr>
<td>20</td>
<td>493819</td>
<td>9.81</td>
</tr>
<tr>
<td>30</td>
<td>162355</td>
<td>5.43</td>
</tr>
<tr>
<td>40</td>
<td>91993</td>
<td>4.17</td>
</tr>
<tr>
<td>50</td>
<td>86390</td>
<td>3.39</td>
</tr>
<tr>
<td>60</td>
<td>454</td>
<td>1.60</td>
</tr>
</tbody>
</table>
Table A.9: Average CPU usage with pre-filtering enabled and normal path disabled with moving average threshold.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20.39</td>
</tr>
<tr>
<td>5</td>
<td>18.39</td>
</tr>
<tr>
<td>10</td>
<td>16.46</td>
</tr>
<tr>
<td>20</td>
<td>5.4</td>
</tr>
<tr>
<td>30</td>
<td>2.3</td>
</tr>
<tr>
<td>40</td>
<td>2.3</td>
</tr>
<tr>
<td>50</td>
<td>1.04</td>
</tr>
<tr>
<td>60</td>
<td>0.56</td>
</tr>
</tbody>
</table>

Table A.10: Average CPU usage with pre-filtering enabled and normal path enabled with moving average threshold.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Average CPU usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24.13</td>
</tr>
<tr>
<td>5</td>
<td>23.15</td>
</tr>
<tr>
<td>10</td>
<td>21.33</td>
</tr>
<tr>
<td>20</td>
<td>20.11</td>
</tr>
<tr>
<td>30</td>
<td>14.58</td>
</tr>
<tr>
<td>40</td>
<td>8.05</td>
</tr>
<tr>
<td>50</td>
<td>5.76</td>
</tr>
<tr>
<td>60</td>
<td>3.57</td>
</tr>
</tbody>
</table>
Table A.11: Number of events detected with pre-filtering enabled and with moving average threshold.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Number of detected events</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4800000</td>
</tr>
<tr>
<td>5</td>
<td>2855898</td>
</tr>
<tr>
<td>10</td>
<td>1408641</td>
</tr>
<tr>
<td>20</td>
<td>214126</td>
</tr>
<tr>
<td>30</td>
<td>47391</td>
</tr>
<tr>
<td>40</td>
<td>7294</td>
</tr>
<tr>
<td>50</td>
<td>4787</td>
</tr>
<tr>
<td>60</td>
<td>1456</td>
</tr>
</tbody>
</table>