Towards traffic change detection in the network data plane

Gonçalo Filipe Oliveira Matos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Fernando Manuel Valente Ramos
Prof. Salvatore Signorello

November 2021

Acknowledgments

I would first like to thank my advisor Professor Fernando Ramos for the opportunity to work on this project, for pushing me to improve and to contribute to the scientific community, and for all the support throughout my dissertation.

A great special thank you to my co-advisor Professor Salvatore Signorello for all the exceptional support, advice, motivation, and feedback he gave me, which allowed me to improve and learn in every step of this project.

A word of gratitude to my parents for their patience and for supporting me through my academic life and to my older brother for always setting the best example and for being someone I can look up to.

A warm thank you to my girlfriend, Ana Rita Ferreira, for all the support and for being the smartest, funniest, prettiest, and most awesome person in the world. Thank you for baking cakes and waffles, and for the very important help with writing every single one of my emails and even this acknowledgements section. Thank you for always being here for me and for never letting me give up.

This work was partially supported by national funds through FCT via UIDB/50021/2020 and PTDC/CCI-INF/30340/2017 (uPVN) project.

Resumo

A identificação de alterações de tráfego é fundamental para a execução de várias tarefas em redes, desde análise de congestão à deteção de intrusões. Os detetores de alterações mais modernos utilizam mecanismos baseados em estruturas de dados chamadas *sketches* que, ao utilizarem descrições compactas do tráfego, atingem melhores compromissos entre memória e precisão. Estas técnicas são utilizadas para detectar *heavy-hitters* (fluxos de tráfego que representam as maiores fontes de congestão em redes) mas, apesar de poderem ser adaptados para detetar alterações neste tipo de tráfego, falham na *generalidade*. Como se focam na monitorização dos fluxos ”pesados”, ignoram os fluxos pequenos que podem também ser a causa de alteração na rede (e.g., micro-rajadas ou ataques de pequeno volume).

Neste trabalho apresentamos o K-MELEON, um sistema *in-network* de deteção de alterações de tráfego em redes informáticas que identifica todas as alterações relevantes de tráfego, em vez de apenas as alterações provocadas por *heavy-hitters*. A contribuição principal deste trabalho é uma variante do k-ary sketch (uma solução genérica de deteção de alterações) que é executada no plano de dados de *switches* programáveis. O maior desafio baseia-se no facto de o k-ary ser *offline*, baseado na análise de dados em *bulk*. O K-MELEON utiliza técnicas de *streaming* que se enquadram no modelo computacional restrito dos *switches* e se adaptam às suas limitações. A avaliação do K-MELEON mostra que este obtém o mesmo grau de exatidão *online* que o k-ary *offline*, e que deteta alterações em redes para qualquer tipo de fluxo: grande ou pequeno.

Palavras-chave: deteção de alterações, sketch, plano de dados, redes definidas por software

Abstract

Identifying traffic changes accurately sits at the core of many network tasks, from congestion analysis to intrusion detection. Modern systems leverage sketch-based data structures that achieve favorable memory-accuracy tradeoffs by maintaining compact summaries of traffic data. Mainly used to detect heavy-hitters (usually the major source of network congestion), some can be adapted to detect traffic changes, but they fail on *generality*. As their core data structures track elephant flows, they miss identifying mice traffic that may be the main cause of change (e.g., microbursts or low-volume attacks).

In this work, we present K-MELEON, an in-network online change detection system that identifies heavy-changes – *instead of changes amongst heavy-hitters only*, a subtle but crucial difference. Our main contribution is a variant of the k-ary sketch (a well-known *heavy-change* detector) that runs on the data plane of a switch. The main challenge was the batch-based design of the original. To address it, K-MELEON features a new stream-based design that matches the pipeline computation model and fits its tough constraints. The preliminary evaluation of this prototype shows that K-MELEON achieves the same level of accuracy for *online* detection as the offline k-ary, detecting changes for any type of flow: be it an elephant, or a mouse.

Keywords: change detection, sketch, data plane, software-defined networks

Contents

List of Tables

List of Figures

Acronyms

API	Application Programming Interface.
ARIMA	Auto-Regressive Integrated Moving Average.
ASIC	Application-Specific Integrated Circuit.
BMv2	Behavior Model version 2.
CDF	Cumulative Distributed Function.
COTS	Commercial Off-the-shelf.
CPU	Central Processing Unit.
CRC	Cyclic Redundancy Check.
DDoS	Distributed Denial of Service.
DoS	Denial of Service.
DRAM	Dynamic Random-Access Memory.
ET	Elastic Trie.
EWMA	Exponentially Weighted Moving Average.
HP	Hewlett-Packard Development Company.
ID	Identifier.
IPFIX	Internet Protocol Flow Information Export.
IP	Internet Protocol.
MAC	Media Access Control.
MA	Moving Average.
MMT	Multiple Match Tables.
NSHW	Non-Seasonal Holt-Winters.
OF	OpenFlow.

P4c	Programming Protocol-independent Packet Processors compiler.
P4	Programming Protocol-independent Packet Processors.
PoF	Protocol-Oblivious Forwarding.
RAM	Random-Access Memory.
RMT	Reconfigurable Match Tables.
SDE	Software Defined Environment.
SDN	Software Defined Networking.
SMA	S-Shaped Moving Average.
SNMP	Simple Network Management Protocol.
SRAM	Static Random-Access Memory.
Tbps	Terabit per second.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol.
VLAN	Virtual Local Area Network.

Chapter 1

Introduction

In a network, *the only constant is change*. While sometimes inconsequential, significant traffic changes are commonly associated with events that require special attention from the operator: they may be an indicator of a malicious attack to the network [1], of a bottleneck caused by a flash crowd [2], or can be a sign of persistent congestion [3]. The ability to detect traffic changes fast and efficiently is therefore a fundamental requirement of many network operation tasks.

Some traffic changes are known to occur, and can be coped with straightforwardly. Take anomaly detection as an example. Signature-based detectors [4, 5] look for patterns that match signatures of known anomalies, enabling detection and reaction to these changes. For the general case of unknown anomalies and unexpected changes, however, operators resort to statistical-based approaches of various kinds [6]. These systems devise a model of normal behavior based on past traffic, and generate alerts when *significant changes* that are inconsistent with the model occur. A core building block of these systems is therefore a mechanism of ***change detection***. The applicability of this technique is not restricted to anomaly detection, however, but to an array of other applications, from network measurement to traffic engineering.

Ideally, a change detection mechanism would analyse all packets from every flow and maintain all flow-related information. Such fine-grained approach does not scale well, incurs in very high overheads, and is effectively not tractable without dedicated hardware [7]. The typical change detection mechanisms therefore either rely on coarse-grained counters, like those provided by SNMP, or are sampling-based. The former provide useful aggregate statistics to detect major problems, but give up information that is often necessary to detect relevant changes, which are left sunk inside the aggregated traffic. Further, typical SNMP collection granularities are on the order of several minutes. The latter, a common solution deployed in practice today, consists in packet sampling (e.g., NetFlow). However, the processing and bandwidth overheads of processing packets make it infeasible to sample at sufficiently high rates. Sampling just 1 in several thousand packets is therefore common. Facebook, for instance, typically samples packets with a probability of 1 in 30,000 [3].

An alternative to sampling that has been much explored recently, particularly spurred by the emergence of programmable networking hardware, is the use of sketches [8, 9]. Sketches are a probabilistic summary technique proposed in the database community for analysing data streaming datasets. They have very interesting properties for the networking context too: sketches are space-efficient and provide probabilistic memory-accuracy guarantees. They thus enable efficient and scalable monitoring solutions that in some cases can run entirely in the network data plane. Several sketch-based systems have been recently proposed, running different network monitoring tasks at line rate in commodity switches [10–16]. These modern systems are, however, restricted to heavy-hitter variants, and none has considered the general problem of change detection[1].

This dissertation describes K-MELEON, a sketch-based change detection system. The starting point is the k-ary sketch [18], a variant of the sketch data structure, which uses a constant, small amount of memory, and has constant per-packet updates. On top of the sketch summaries runs a time series forecast model (EWMA, ARIMA, etc.) that detects significant changes by looking for flows with large forecast errors. The k-ary is an efficient and accurate solution for the general change detection problem, but has some limitations that preclude its deployment. First, *it is an offline solution* based on the analysis of bulk traffic traces. Network operators, however, need detection solutions that enable real-time decisions. Second, *it is software-based*, limiting its performance, response time, and scalability. For instance, the k-ary [18] is able to detect changes only in the order of minutes. As a result, it would fail to detect relevant events in today's networks, including microbursts [3] and short duration attacks [19].

1.1 Contributions

K-MELEON addresses the above challenges with k-ary by running the sketch and forecasting modules of its change detection mechanism entirely in the data plane of programmable switches [20, 21]. As a result, it enables ***online change detection***, with ***sub-second responses***, at ***Terabit scales***. The challenge is to develop functionally equivalent algorithms to the batch-based k-ary ones, while fitting the switch pipeline computation model and its harsh compute and memory constraints. The main novelty of this solution is a *streaming-based* algorithm that carefully computes the change detection estimates *continuously, as packets traverse the switch*.

In summary, the contributions of this dissertation are:

- The design of K-MELEON, an online change detection system that speeds and scales up detection by leveraging programmable switches.
- The implementation of a prototype in P4 [22, 23], which we make available open-source (in [24]).
- An evaluation that demonstrates K-MELEON achieves the same level of accuracy as the offline k-ary solution, and detects changes from *any* type of flow (not only heavy-hitter traffic as existing work [16, 17]).

A short version of this work was accepted to appear at the CoNEXT'21 Student Workshop. An extended version of that paper was also accepted to appear at the EuroP4'21 workshop.

[1]To be clear, a few works [16, 17] have considered *heavy(-hitter)* change detection, a subset of the problem, as will be clarified later. In short, these systems detect traffic changes among heavy-hitters only, but in practice not all flows that experience significant changes are "heavy". This dissertation investigates the more general problem.

1.2 Structure of the document

The rest of this document is organized as follows:

- Chapter 2 - Background

  This Chapter describes the background and related work. The first section introduces the works which enabled the programmability of the control and data planes. Then, the most common methods for network traffic measurement are described in Section 2.2. Finally, Section 2.3 explores the state of the art in the more specific problem of traffic change detection.

- Chapter 3 - Design

  This Chapter details the design steps taken to build the stream-based change detection solution we propose, K-MELEON. The first sections define the problem tackled by this thesis and describe the starting point of the design phase: the k-ary sketch [18]. Section 3.3 sets the requirements for change detection and Section 3.4 discusses the challenges faced by the k-ary algorithm with on-line change detection. Finally, Section 3.5 describes the design of K-MELEON, by overcoming, step by step, each of the challenges specified for on-line change detection.

- Chapter 4 - Implementation

  This Chapter describes the implementation of K-MELEON. Section 4.1 reports our implementation of the k-ary algorithm in python, by closely following the description in the original paper [18]. We use this as baseline for our evaluation in Chapter 5. Section 4.2 describes the implementation of K-MELEON in the data plane using P4, and the interactions required with the control plane. Lastly, Section 4.3 presents the set of tools created to support the evaluation of both programs.

- Chapter 5 - Evaluation

  This Chapter presents the evaluation of K-MELEON. Section 5.1 presents the testing environment, including the datasets and the experimental setup used. Section 5.2 validates the k-ary implementation in python and explores the configuration parameters for the use case of attack detection. Lastly, Section 5.3 measures the ability of K-MELEON to detect network attacks and microbursts in the data plane.

- Chapter 6 - Conclusions

  This Chapter summarizes the work developed and discusses future work.

Chapter 2

Background

This chapter describes previous work related to the subject of this dissertation. Section 2.1 introduces the concept of Software Defined Networking (SDN), which breaks the tight coupling of the control and forwarding planes in network devices. At first, this Section highlights the management problems inherent to conventional networks which the SDN paradigm has contributed to resolve. Afterwards, it presents the OpenFlow protocol and the P4 language that have enabled a standard programmability of the control and data plane, respectively, in SDN. As the work in this thesis focuses on traffic measurements, Section 2.2 presents an overview of traditional techniques in monitoring systems based on packet sampling. Furthermore, it presents some of the most relevant solutions leveraging sketching algorithms within legacy and SDN-enabled networks. Finally, Section 2.3 outlines the importance of traffic change detection for several critical monitoring tasks, with a focus on the anomaly detection use case for security. Then, it presents the main challenges and techniques for performing traffic change detection in today's programmable networks.

2.1 Programmable Networks

Conventional computer networks are typically complex and hard to manage, and include different kinds of equipment like routers, switches, network address translators, server load balancers, firewalls, and intrusion detection systems. This kind of equipment is typically fixed-function and has individual configuration interfaces that vary across manufacturers and require manual configuration by knowledgeable network administrators.

Historically, those networks have been vertically integrated; the (1) control plane - that determines how to handle network traffic - and the (2) data plane - that forwards packets according to the rules defined by the control plane, are bound together inside the networking devices. This has represented a barrier for innovation, hindering the testing and deployment of new ideas into production networks.

Software Defined Networking (SDN) [25, 26] has recently become a widely adopted network paradigm to resolve the above problem. SDN changes the way networks are designed and managed by separating the control plane from the data plane and by consolidating the former, allowing for a logically central-

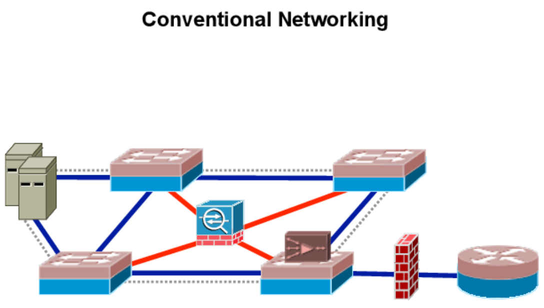


Figure 2.1: Conventional networking.

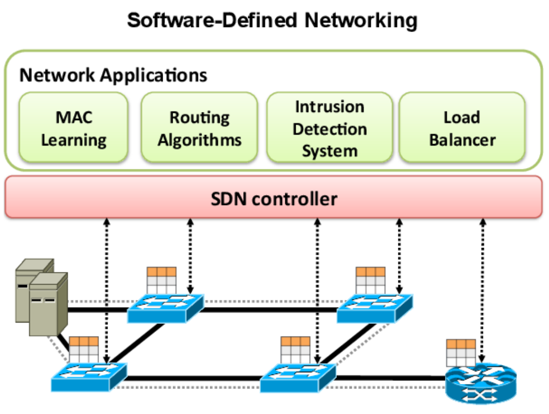


Figure 2.2: Software-Defined Networking [26].

ized control plane program to control multiple data plane components. Figures 2.1 and 2.2 illustrate the difference between conventional networking, where control functions are performed in each network device, and SDN, where functions are now logically centralized into a controller entity, while the network data plane elements act like simple forwarding devices.

2.1.1 Programmable Control Planes

Software Defined Networks provide a great advantage against conventional networks by introducing the ability to program the control plane. In fact, by separating the control from the data plane, SDN infrastructures allow for the creation of a flexible and decentralized controller where network applications can be executed. With SDN, network managers can remove the decision process from the network devices, that become simple forwarding elements and build more scalable and flexible networks. Although a centralized controller might induce the idea of a physically centralized system, SDN does not mandate it. Indeed, SDN network designs often resort to physically distributed control planes to achieve sufficient levels of performance, scalability and reliability.

The OpenFlow (OF) protocol, proposed in [27], first described how such an SDN system could be designed for researchers to experiment with new protocols in production networks. The design of OF was rooted on the observation that a common forwarding abstraction was available across several modern routers, despite their differences with vendors and models. In fact, these devices featured hardware flow-tables to implement functions like firewall-ing, NAT-ing, and statistics collection, at line-rate. And, by abstracting the flow-table mechanism present in many of the commercially-available switches and routers, OF could serve as an open protocol to uniformly program all of these devices.

An OF-compliant switch features at least three components: (1) A Flow Table, (2) A Secure Channel and (3) The OpenFlow protocol. The Flow Table associates an action to a certain flow, instructing the switch how to process such flow. A flow could be a TCP connection, for example, or all packets from a particular MAC/IP address, or all packets with the same VLAN tags, etc. The Secure Channel allows a remote-control process to exchange commands and packets with the switch. The OF protocol provides an open and standard way for a controller to communicate with a switch.

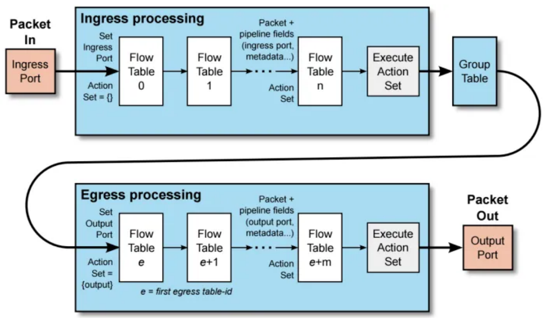


Figure 2.3: The OpenFlow pipeline [28]

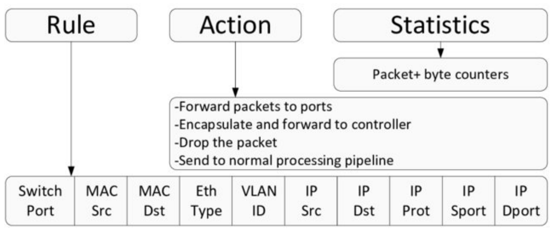


Figure 2.4: An OpenFlow Flow-Rule Table [29].

Figures 2.3 and 2.4 illustrate the structure of the OF logical processing pipeline and of the flow table, respectively. The OF processing pipeline consists of two stages: ingress processing (for incoming packets) and egress processing (for outgoing packets). Each pipeline consists of a series of sequentially numbered flow tables. When a packet reaches the ingress port, it is matched by selected protocol header fields with the flow entries of the first flow table (table 0). Each entry contains a set of actions that it can apply to matched packets and associated counters it can increment. A flow entry can steer the packet to any flow table with a number larger than its own flow table number. The last flow table, besides executing actions associated to its entries, can specify the output port for the forwarding of the current packet. Flow tables can be populated with specific entries at run-time by an SDN controller through the OF protocol.

Since its proposition in 2008, OpenFlow has grown to support more protocols and expose more of the underlying switches' capabilities to the controller. Today, OF is the most used communication protocol for software defined networks. It is widely used across different kind of networks, e.g., in universities [30] and at big companies (Google [31], HP [32]), only to name a few. Its continuous development is overseen by a very diverse and active community across industry and academia.

2.1.2 Programmable Data Planes

Although the OpenFlow (OF) specification keeps adding support for new protocols, the current programming model shows two main limitations: (1) match+action processing on only a fixed set of fields and (2) a limited repertoire of packet processing actions. Supported protocols and available actions, which are defined in the OF specification, are mostly dictated by off-the-shelf fixed-function switches. A new programmable switch model, the Reconfigurable Match Tables (RMT) switch architecture, was proposed [20] with the aim to overcome those two limitations with fixed-function switches. RMT consists of a line-rate pipelined switch architecture featuring a minimal set of generic packet processing primitives. Each primitive specifies a packet processing operation executed in a piece of hardware which can be *reconfigured* in the field.

The RMT model improves over the mainstream OF's Multiple Match Tables (MMT) model where multiple match tables in a pipeline match against a subset of *only pre-defined* protocol fields in a packet. In more detail, RMT enhances the MMT data plane programming model in four ways: (1) new protocols can be added or existing protocols can have their definitions altered, (2) multiple match tables with

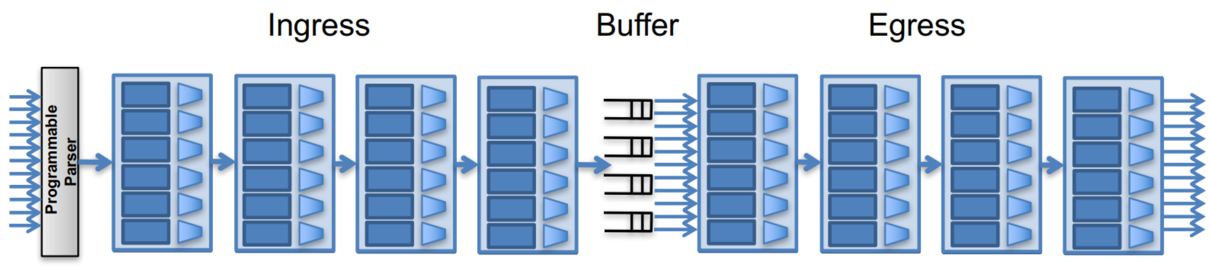


Figure 2.5: Match-action tables pipeline in PISA.

arbitrary width and depth can be specified, subject only to an overall resource limit, (3) new actions can be defined, and (4) arbitrarily modified packets can be placed in specified queue(s), for output of any subset of ports, with a queuing discipline specified for each queue.

The scientific advances brought by the RMT design and other related works [33] led to the definition of several new abstractions for data-plane programming, such as the Protocol Independent Switch Architecture, which is now used by the most recent programmable switches. The PISA architecture generalizes the Reconfigurable match tables model to implement an ingress and egress pipeline consisting of a series of match-action tables arranged in stages that execute synchronously as illustrated in Figure 2.5. More specifically, whenever a packet enters the pipeline, it is processed by each stage and delivered to the next, until it exits the pipeline. This architecture is also protocol-independent because it does not model any specific packet parsing until its parser block is programmed. This novel switch target allows non-standard protocols and packet processing behaviors to be configured into the switch through a program at compile time.

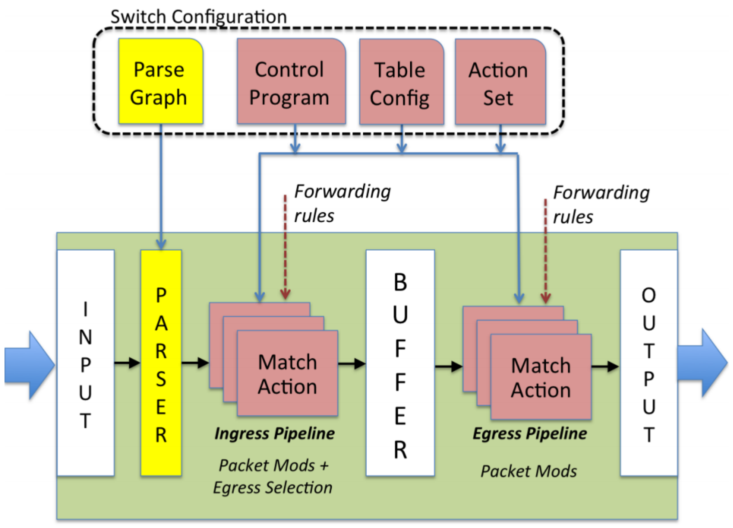


Figure 2.6: The P4 abstract forwarding model [20].

```
/* -*- P4_16 -*- */
#include <core.p4>
#include <v1model.p4>

#define SKETCH_BUCKET_LENGTH 28
#define SKETCH_CELL_BIT_WIDTH 64
#define SKETCH_REGISTER(num)
register<bit<SKETCH_CELL_BIT_WIDTH>>
(SKETCH_BUCKET_LENGTH) sketch##num

#define SKETCH_COUNT(num,algorithm) . . .

const bit<16> TYPE_IPV4 = 0x800;

typedef bit<9>  egressSpec_t;
typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
   macAddr_t dstAddr;
   macAddr_t srcAddr;
   bit<16>   etherType;
}

header ipv4_t { . . . }

struct metadata { . . . }

struct headers {
   ethernet_t   ethernet;
   ipv4_t       ipv4;
}
```

```
parser MyParser(packet_in packet,
            out headers hdr,
            inout metadata meta,
            inout standard_metadata_t standard_metadata) {
   state start {
      transition parse_ethernet;
   }

   state parse_ethernet {
      packet.extract(hdr.ethernet);
      transition select(hdr.ethernet.etherType) {
         TYPE_IPV4: parse_ipv4;
         default: accept;
      }
   }
   state parse_ipv4 { . . . }
}

control MyVerifyChecksum(inout headers hdr, inout
metadata meta) {
   apply { . . . }
}

control MyIngress(inout headers hdr,
             inout metadata meta,
             inout standard_metadata_t standard_metadata)
{
   action sketch_count(){
      SKETCH_COUNT(0, crc32_custom);
      SKETCH_COUNT(1, crc32_custom);
      SKETCH_COUNT(2, crc32_custom);
   }
```

```
   table forwarding {
      key = {
         standard_metadata.ingress_port: exact; }
      actions = { . . . }
      default_action = drop;
   } apply {
      if (hdr.ipv4.isValid() && hdr.tcp.isValid()){
         sketch_count(); }
      forwarding.apply();
   }}

control MyEgress(inout headers hdr,
             inout metadata meta,
             inout standard_metadata_t standard_metadata)
{
   apply { . . . }
}

control MyComputeChecksum(inout headers  hdr, inout
metadata meta) {
    apply { . . . }}

control MyDeparser(packet_out packet, in headers hdr) {
   apply { . . . }}

/*****  S W I T C H  *****/

V1Switch(
   MyParser(), MyVerifyChecksum(),
   MyIngress(), MyEgress(), MyComputeChecksum(),
   MyDeparser()
) main;
```

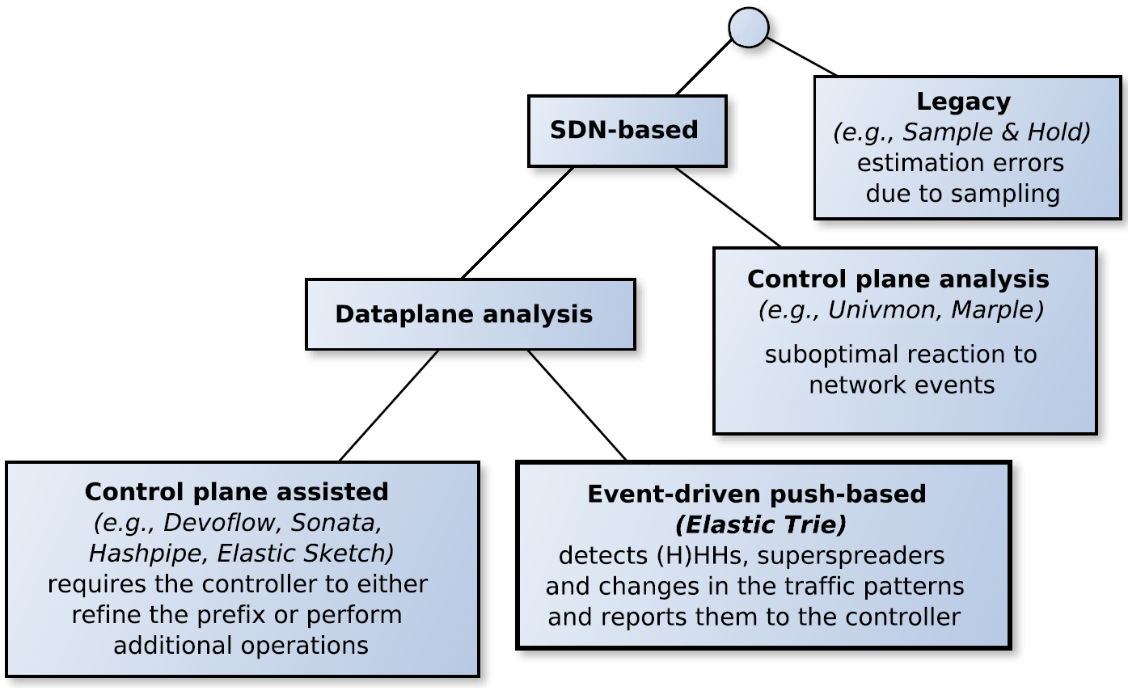
Figure 2.7: Excerpt of a P4 program implementing the Count-Sketch [34].

2.1.3 The P4 Language

Programming Protocol-independent Packet Processors (P4) [22] is a domain-specific language used for programming data-plane algorithms across different targets. The P4's abstract forwarding model is illustrated in Figure 2.6. It assumes switches forwarding packets through a programmable packets parser followed by multiple programmable match+action stages arranged in series, parallel, or both. The P4 language and programming model for packet processing envision programmers to create target- and protocol-independent programs which target-specific compilers can then map onto a variety of different forwarding devices. This mode of operation evolves from the OF classical programming model that assumed a fixed parser and worked with a limited set of standard protocols and processing functions on match+action stages arranged in series.

Figure 2.7 shows a P4 implementation of a simple count-sketch [34] algorithm. This example helps illustrate through a program the main components of the P4 forwarding model which enable a programmer to specify custom data plane behaviors. The **header** definition, which through **header** types, describes a format for each header within a packet. The **parser** block describing allowed sequences of headers within a packet through a state machine, helping identify and extract those sequences in incoming packets. Ingress and egress processing pipelines define how packets are processed through tables and actions, involving operations with packet metadata, header fields and possibly values stored in stateful memory (e.g., registers). Finally, a deparser block declares how packet headers are serialized after egress processing. The full list of constructs available with P4 can be found in the official language specification [35].

Since the initial straw-man proposal, P4 has considerably evolved and it is today the most prominent high-level abstraction for describing data-plane behaviors across hardware and software network



SDN-based
Legacy
(e.g., Sample & Hold)
estimation errors due to sampling
Dataplane analysis
Control plane analysis
(e.g., Univmon, Marple)
suboptimal reaction to network events
Control plane assisted
(e.g., Devoflow, Sonata, Hashpipe, Elastic Sketch)
requires the controller to either refine the prefix or perform additional operations
Event-driven push-based
(Elastic Trie)
detects (H)HHs, superspreaders and changes in the traffic patterns and reports them to the controller

Figure 2.8: Design space in traffic aggregates detection [36].

devices, like network cards, switches and routers.

2.2 Traffic Measurements

The advances in programmable networks have enabled the experimentation and deployment of improved techniques for collecting and analyzing network traffic measurements. In fact, many network applications, e.g., for accounting, resources provisioning, and security, increasingly rely on accurate network traffic measurements. These applications typically used a flow-based approach to measure network traffic but, as link speeds, and consequently the number of flows, have increased in today's networks, it becomes impossible to store information for every flow. As a result, traditional flow-based network measurement approaches have become typically slow, inaccurate, and/or resource-intensive. As an alternative, methods based on finely-tuned traffic approximations, which leverage insights from the theory of data stream analysis, have been explored recently. This Section covers the most common techniques available for network measurement, which span from legacy methods to newer SDN-enabled ones. As a guide to the rest of this section, Figure 2.8 presents an overview of the design space for a specific measurement task meant to help with the detection of traffic aggregates in networks.

2.2.1 Traditional Methods

Conventional routers offer flow-based measurements such as NetFlow [37], sFlow [38], IPFIX [39], or NetStream [40], providing flow-level information about network traffic. In order to store a constantly increasing number of flows, flow measurement devices have to resort to cheaper memories, namely

DRAM. However, with the increasing line speeds, those devices cannot monitor all flows. So, overall, these systems show some limitations which may be critical for certain monitoring applications. For example, in order to detect heavy flows, NetFlow collects flow counts for *sampled* packets in the data plane and then these records are exported to a NetFlow collector that processes the data and performs traffic analysis. While a high sampling rate results in too many counters not fitting in memory, having a lower sampling rate incurs in missed flows and in a loss of accuracy.

An improved NetFlow-based approach, **FlowRadar** [41], stores counters for all flows with low memory overhead and exports these counters in short time intervals. To do so, FlowRadar devises per-switch encoding of flows and their respective counters into small fixed-size memory space, called *encoded flowsets*, with constant flow insertion time. To make room for new flows, FlowRadar periodically removes old and idle flows. A remote collector then performs network-wide decoding of the flows, and temporal and flow space analysis, thus keeping both switch processing time and memory use at a minimum. FlowRadar achieves better measurement performance than purely sample-based approaches like NetFlow and sFlow with regard to two main aspects: (1) *Flow coverage*: counting all the flows without sampling; and (2) *Temporal coverage*: exporting these counters for each short time slot.

However, FlowRadar is based on manual configuration of closed-off network equipment, and requires setting a maximum number of flows to track, incurring in higher decoding times whenever the number of flows increases. Moreover, the encoding process introduces collision rates, which increase with the number of flows.

More recently, the emergence of programmable switches has generally enabled the design of improved SDN-based network monitoring frameworks and languages that extend traditional methods, such as MAFIA [42], Marple [43], and Sonata [44]. These tools aim to provide network operators with a unified query interface that compiles high-level monitoring queries into equivalent low-level code, e.g., P4 [22], that executes in programmable switches.

2.2.2 Sketch-based Methods

Traditional approaches to network measurement usually fell into two extremes: (1) versatile monitoring solutions based on packet sampling, with low fidelity, or (2) high fidelity but dedicated hardware appliances (middle-boxes) for very specific monitoring tasks.

An alternative to the use of flow-based packet sampling are sketching algorithms. A **Sketch** is a probabilistic summary of a data stream that can reduce the memory requirement of measurement tasks while preserving the required accuracy, thus allowing for the design of memory-efficient monitoring systems. The sketch data structure is generally a multi-dimensional array, like in CountMin [45] and Count Sketch [34], or an array of bits, like in Bitmap [46], where items of a data stream are stored by means of several independent hash functions. Sketches have a small and fixed memory size, making them suitable for implementation in a cache or low-latency memory (SRAM) usually available in switches and routers. Generally, any sketch-based algorithm supports two operations: (1) to update the sketch, using the key and value of each item in the data stream and (2) to query the sketch, to get statistics about the items in the data stream.

One of the first practical systems using sketches in networking was **OpenSketch** [9]. OpenSketch proposes a new API for traffic measurement using sketches. Inspired by OpenFlow, which separates the control plane from the data plane to allow for more efficient control of switches, OpenSketch implements a new software defined measurement architecture that separates the measurement data plane from the control plane. This separation allows for the design of a measurement framework that is both generic and efficient.

The OpenSketch data plane provides a three stage pipeline, which is able to support several measurement tasks in high-speed links with limited memory. The three stages of the OpenSketch pipeline are the following: (1) the hashing stage, which reduces the measurement data, (2) the classification stage, which selects flows, and (3) the counting stage, which accumulates traffic statistics. The OpenSketch data plane supports various measurement tasks by efficiently using the switch's capabilities and allocating the switch's memory to: (1) select the flows to monitor and (2) store/export the collected measurement data.

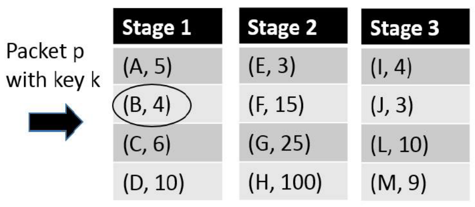
The OpenSketch control plane provides a measurement library with a sketch manager, which automatically configures the data plane pipeline to use different sketches, and a resource allocator, which allocates the switch's memory to each task to maximize accuracy. Thus, the control plane allows the programmer to choose the flows to monitor, the traffic metrics to collect, and the storage of the collected flow data. Jointly with OpenFlow, OpenSketch can provide a complete measurement and control platform for the deployment and management of many sketch-based measurement tasks.

While OpenSketch developed an FPGA-based solution, recent works started exploring production-level programmable switches, namely for the detection of heavy hitters, that is, flows carrying large traffic volumes. **Hashpipe** [15] manages to track the *k* heaviest flows with high accuracy within the features and constraints of programmable switches. More precisely, Hashpipe leverages the advantages of P4-programmable switches to retrieve heavy hitters by requiring an amount of memory proportional only to the number of heavy flows.

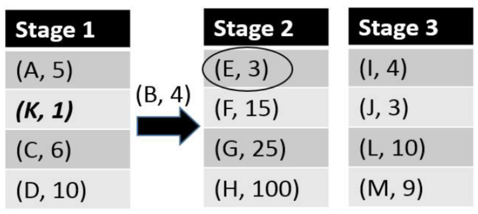
Hashpipe is a sketch-based solution that hashes and counts all packets in a pipeline of hash tables. By adapting the Space-Saving algorithm [47], Hashpipe retains counters for heavy flows while discarding the lighter flows over time. Figure 2.9 illustrates the Hashpipe pipeline through an example.

Although Hashpipe manages to achieve high performance in heavy hitter detection, an ideal solution for traffic measurement should be generic in terms of the monitoring task while still providing good accuracy when compared to systems designed ad-hoc for specific monitoring tasks.

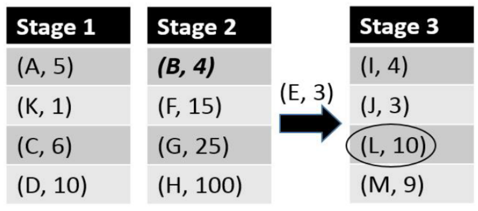
UnivMon [14] is a sketch-based monitoring framework which offers: (1) generality, by delaying the binding of the traffic measurements to specific applications and (2) high fidelity, with regard to the metrics. UnivMon builds upon three fundamental requirements for a generic monitoring framework: (i) Fidelity for a broad spectrum of applications: assuring high accuracy for any set of metrics to be estimated; (ii) One-Big-Switch abstraction for monitoring: merging all traffic estimations to appear as if all traffic is being monitored by one big switch; (iii) Feasible Implementation road map: implementing in programmable switch hardware.



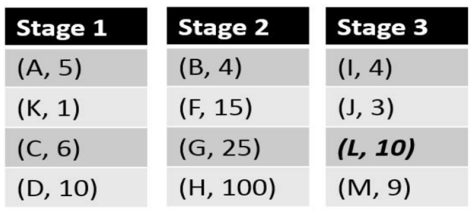
(a) Initial state of table



(b) New flow is placed with value 1 in first stage



(c) B being larger evicts E



(d) L being larger is retained in the table

Figure 2.9: An illustration of HashPipe [15]. When (a) a packet enters stage 1, if its key is present, its counter is incremented. Otherwise, it replaces the key with the lowest counter. Then, (b) the replaced key will be hashed to a slot in stage 2. If its counter is smaller than the key in that slot, the key is removed, otherwise, it will replace the key. This process will repeat (c,d) until a key is removed or the final stage is reached.

The UnivMon data plane leverages recent advances on universal streaming [48, 49] to build universal sketches that require no prior knowledge of the metrics to be estimated. The UnivMon control plane collects sketch information from the network components, runs simple estimation algorithms for every application of interest, and provides APIs and libraries for running estimation queries. The control plane also generates sketching manifests that specify the set of universal sketch instances each router needs to maintain, while taking the network topology, routing policies and knowledge of the hardware resource constraints into account.

Another common problem in network monitoring is that when a network is subject to transient effects such as congestion or attacks, the unusual traffic variations experienced may affect the performance of the monitoring system. Although systems like, e.g., HashPipe [15] and UnivMon [14], have improved network monitoring across several dimensions like accuracy, speed, memory usage, and generality, those systems still do not take dynamic traffic variations into consideration.

Elastic Sketch [50] proposes a sketching algorithm to compute generic sketches which are able to adapt to the network characteristics. Elastic Sketch achieves elasticity through three main mechanisms: (1) compressing the sketch into a sufficient size to fit the available bandwidth, (2) changing the processing method according to the current packet rate: when the packet rate becomes too high, only information regarding heavy flows is recorded, discarding mice flows; and (3) dynamically increasing the memory size of the heavy part as the number of elephant flows changes.

The basic workflow of Elastic Sketch builds upon the separation of elephant flows from mice flows. It keeps a series of buckets for the elephant flows, each recording the information of a flow: flow ID, positive votes, negative votes and a flag. Positive votes records the number of packets belonging to this flow, negative votes records the opposite, and the flag indicates whether the light part might contain positive votes for this flow. When a packet with flow ID f1 arrives, if f1 is the same as the flow in the packet it increments the positive votes. Otherwise, it increments the negative votes and checks if the ratio between positive and negative votes exceeds a predefined threshold. If it does, the flow is removed from the bucket and the new one is inserted.

Elastic Sketch manages to be generic both in terms of measurement tasks and platforms. The former is achieved by keeping all necessary information for each packet, but discarding the IDs of mice flows. The latter is achieved by building multiple versions of the sketch for both software and hardware platforms. Along with the ability to adapt to the available bandwidth, changes in packet rate, and flow size distribution, Elastic Sketch can be directly used for flow size estimation, heavy hitter detection, heavy change detection and for estimation of flow size distribution, entropy and cardinality. Although it can be used for change detection, it (i) requires to keep multiple copies of the entire sketch structure, (ii) those entire sketch data structures must be offloaded at the end of each time interval, (iii) more importantly, only the heavy part of the sketch preserve the flow key and can be checked for changes, leaving undetected changes with respect to flows tracked in the light part of the sketch. In other words, it detects only heavy-hitter changes.

Overall, sketch-based methods prove to be theoretically sound. Sketches provide accurate per-flow summaries, which provide approximate measurement results with limited resources. However, such approaches are difficult to use in practice, because large data sources compete for the same resources and incur in measurement errors due to resource conflicts. While resources can be provisioned according to the needs of an application, the resource configurations and accuracy parameters are closely related.

SketchLearn [12] is a sketch-based measurement technique which characterizes the statistical properties of such conflicts in sketches through parameter-free inference performed in the control plane, instead of pursuing a perfect configuration. It sets four main requirements for this technique: (1) small memory usage, (2) fast per-packet processing, (3) real-time response, and (4) generality. SketchLearn builds a multi-level sketch technique, which stores multiple small sketches. Each of the small sketches tracks a specific bit from the flow-key definition. The multi-level sketch also provides one key property: If there is no large flow in it, its counter values should follow a gaussian distribution. Thus, SketchLearn iteratively infers and extracts large flows from the multi-level sketch until the remaining small flows fit this property. Such separation of large from smaller flows enables sketch collision resolution for many traffic statistics and more refined measurements for different networking tasks. Ultimately, SketchLearn constitutes the first sketch-based measurement technique that builds on the statistical properties of sketches instead of relying only on sketch configurations.

All the aforementioned techniques improve traffic measurements on programmable networks across some dimensions. However, they do not fulfil at the same time all the requirements set as important by this work for traffic change detection, which have been summarzed in Table 2.1. More specifically, HashPipe only targets the detection of heavy hitters, UnivMon is generic but does not detect changes in the data plane, Elastic Sketch ignores mice flows, and SketchLearn is not amenable to an efficient implementation on a high-speed data plane because of the considerably high amount of sketch updates required (up to l where l is the length in bits of the flow key, e.g., 104 bits for the five tuple).

2.3 Traffic Change Detection

Understanding and detecting traffic changes in networks is of paramount importance to operate networks efficiently, reliably, and securely across several dimensions. A prime example is anomaly detection. Traffic anomalies such as failures and attacks are very common in networks. A common approach to detect anomalies consists in looking for patterns that match the signatures of previously known anomalies, used, e.g., in tools like Bro [51] and Snort [52]. However, an attacker could induce variations in an attack, allowing it to proceed unmatched by known signatures. Looking for traffic changes is therefore one possible approach to detect these anomaly-based attacks.

Change detection studies have focused primarily in the context of time series forecasting and outlier analysis. Standard techniques for change detection include smoothing, such as exponential smoothing or sliding window averaging, the Box-Jenkins ARIMA modeling, and wavelet-based techniques. While these techniques have been successfully applied to network fault detection [53–55] and intrusion detection [56, 57], these change detection techniques typically only handle a rather small number of time series. These might work with smaller, highly aggregated network traffic data, but directly applying these techniques on a per-flow basis does not scale to the needs of the massive data streams given today's traffic volume and link speeds.

The study in [58] seminally proposed the use of sketches for the detection of significant changes in massive data streams with a large number of network time series. It proposes a variant of the sketch data structure called **k-ary sketch** that uses a constant, small amount of memory, and incurs in constant cost on per-record update and reconstruction of the forecast error for a given key. In k-ary, a forecasting module uses the summarized information about the input stream from the sketches in the past intervals to build a forecast sketch and to compute the error between the observed and the forecast sketches. Afterwards, a change detection module chooses an alarm threshold based on the estimates from the k-ary sketches and checks if the computed error is above that threshold.

The k-ary leverages sketches to detect any kind of heavy change, unlike other solutions [16, 17] which only allow the detection of changes amongst heavy flows, called heavy hitters. To distinguish between the two approaches, we say that the former detects heavy changes and the latter detects heavy-hitter changers.

While the k-ary detects heavy changes over massive data streams, it still struggles to find the real culprit and is subject to sketch reconstruction errors. To address this issue, a new version of the k-ary sketch [59] proposed the use of *reversible sketches*, which are designed to track specific flows in the sketch data-structures. That work tackles the irreversibility problem of the k-ary by using reverse modular hashing techniques that infer the keys of culprit flows from sketches without storing any explicit key information.

More recently, the MV Sketch [17] has been proposed for performing fast heavy flow detection online, using reversible sketches. **MV-Sketch** [60] uses an array of buckets, similarly to the Count-Min Sketch, to store information about the flows. Yet, MV-Sketch enhances the bucket structure with an additional value, per bucket, to also track the candidate heavy flow by applying a majority vote algorithm (MJRTY) [61]. MV-Sketches can be used to detect heavy-hitter changers by leveraging the linearity property of sketches - sketches can be considered vectors of buckets, to which linear algebra operations (like sum, subtraction, scalar multiplication, etc.) can be applied. For example, by subtracting the values of buckets at the same positions of two sketches computed over different time intervals and recovering the heavy flows from the buckets whose differences exceed a threshold T.

The advent of SDN and programmable networks has also enabled the development of new techniques and systems for traffic change detection. The possibility to build more accurate techniques for real-time detection at high-speed has further been enabled by the possibility to configure the network data plane [22, 62] to assist with the measurement task. However, data plane assisted measurements typically incur in frequent updates to the control plane for further inspection.

In order to minimize the amount of information transmitted from the data plane to the controller for analysis, the recent work in [36] devises a new data structure, called Elastic Trie (ET). ET enables performing the detection of high volume traffic clusters almost entirely inside the data plane, by sending notifications from the switch to the controller only if a traffic change requires further analysis.

The Elastic Trie (ET) structure can be implemented using match-action units commonly available on the pipeline of modern programmable switches. It is based on a tree structure with the aim to leverage the natural hierarchical organization of IP addresses; since by using standard longest-prefix techniques, it can find more easily the memory block where a specified prefix is stored. Each node in the prefix tree (trie) consists of three parts: (1) the counter for the left child, (2) the counter for the right child and (3) a timestamp. Each counter represents the amount of traffic for each sub-prefix associated to the node while their sum represents the amount of traffic sent by the prefix itself. The timestamp keeps the time of creation of the node or the time of the last reset. To assist in change detection, ET defines two timers: active timeout and inactive timeout, which are checked against the timestamps in each node. If, for a node, both the active and inactive timeouts have not expired, but one of the counters exceeds a threshold T, the node is expanded and reported as a heavy hitter. When the inactive timeout has not expired, the active timeout has expired and the sum of counters does not exceed the threshold T, the node is collapsed. Then, by tracking the number of nodes collapsed and expanded, ET can detect sudden changes in short-term traffic behavior and promptly report them to the controller.

The ET structure offers three new properties important to the network monitoring task: (1) A push-based approach, where the data plane only sends information to the controller when specific events occur; (2) a coarse-grained approach to the prefix responsible for network events, which gives the controller a finer or coarser prefix information depending on predefined settings; and (3) the ability to compute all the operations in the data plane, which minimizes the amount of information that needs to be exchanged with the control plane.

To summarize, all the works presented in this Section devise techniques for the collection and analysis of traffic measurements which can potentially be leveraged for traffic change detection on pro-

Approach	Tbps performance	Reversible	Heavy-Hitter Change Detection	Generic Change Detection
HashPipe [15]	X	-	-	-
UnivMon [14]	X	-	-	-
Elastic Sketch [50]	X	-	X	-
SketchLearn [12]	-	X	-	-
K-ary Sketch [58]	-	-	X	X
Reversible K-ary Sketch [59]	-	X	X	X
MV Sketch [60]	-	X	X	-
Elastic Trie [36]	X	-	X	-
K-MELEON	X	X	X	X

Table 2.1: Overview of the state-of-the-art approaches for traffic change detection.

grammable networks. However, none of those can individually achieve all of the properties we target for change detection, namely: (1) to run at line-rate, (2) to be reversible, (3) to perform generic change detection including of mice and elephant traffic. Table 2.1 presents a comparison of existing techniques and the goal of our system.

2.4 Summary

In this chapter, we have started by describing the trend towards programmable networks, from the separation between the control and data planes to the programmability of the data plane of switches. We presented the contributions from OpenFlow, which enabled the separation of the planes, the improvements provided by the RMT model, and introduced P4, a domain-specific language used for programming the data-plane. Afterwards, we investigated commonly used traffic measurement techniques, detailing the key improvements brought by state-of-the-art sketch-based methods over sampling-based ones. We finalized by presenting recent advances in traffic change detection, and by introducing the main properties targeted by the change detection module proposed in this thesis.

The next chapter dives into the design of the proposed solution, K-MELEON, which leverages the k-ary sketch algorithm to perform change detection almost entirely in the data plane of state-of-the-art programmable switches.

Chapter 3

Design

The goal of this thesis is to develop an online change detection mechanism that runs in the data plane of the most recent P4-capable switches. Towards that goal, we have leveraged the state-of-the-art change detection algorithm: the k-ary sketch [18]. The k-ary is an offline algorithm that follows a batch-based processing approach. As such, it does not fit the pipelined packet-processing model of modern switching ASICs [63]. We propose K-MELEON, a stream-based version of the k-ary algorithm. In a nutshell, the design of K-MELEON revisits the logic of k-ary and adapts it to run within the constraints of P4-programmable data planes.

This chapter is organized as follows. First, we provide the reader with the problem definition in Section 3.1, illustrate the main data structures and operations of the k-ary algorithm in Section 3.2, and define the requirements set for this data-plane module in Section 3.3. Afterwards, we provide the necessary background to understand the challenges with performing traffic change detection in the network data plane, in Section 3.4. Finally, we describe the K-MELEON data-plane module for traffic change detection, in Section 3.5.

3.1 Problem definition

We use the general Turnstile Model [64] to describe data streams. Specifically, let $I = \alpha 1, \alpha 2, \dots$ be an input stream that arrives sequentially, item by item. Each item $\alpha_i = (a_i, u_i)$ consists of a key $a_i \in [n]$, where $[n] = 0, 1, \dots, n-1$, and of an update $u_i \in \mathbb{R}$. In the networking context, the key can be defined using one or more fields in packet headers (e.g., the 5-tuple). Associated with each key $a \in [n]$ is a time varying signal $A[a]$. The arrival of each new data item (a_i, u_i) causes the underlying signal $A[a_i]$ to be updated: $A[a_i] += u_i$. The general goal of change detection is to identify all those signals with significant changes in their behavior over a certain time.

In this model of change detection we break up the packet stream into temporally adjacent chunks. We are interested in keys whose signals differ dramatically in size between two consecutive chunks. In particular, for a given ϕ, we define a key as a *heavy change key* if the difference in its signal exceeds ϕ percent of the total change over all keys. That is, for two input sets 1 and 2, if the signal for a key x

is $A_1[x]$ over the first input and $A_2[x]$ over the second, then the difference signal for x is defined to be $D[x] = |A_1[x] - A_2[x]|$. The total difference amongst all keys is $D = \sum_{x \in [n]} D[x]$. A key x is then defined to be a *heavy change key* if and only if $D[x] \geq \phi \cdot D$.

3.2 The k-ary sketching algorithm

The k-ary algorithm [18] is a generic change detection technique, which uses sketches to store per-flow information about an input stream into a compact memory-efficient structure, forecasts future values for the input stream, and performs change detection by comparing the observed data against the predicted values. This technique follows the turnstile model described in Section 3.1. More precisely, it considers input items consisting of a key (source and destination IPs, 5-tuple, etc.) and a (possibly negative) update (unitary increments, packet size, byte count, etc.). By comparing information in consecutive fixed-length intervals of time, called epochs, it identifies all those keys of the input stream with significant changes in their behavior. The k-ary algorithm can be illustrated by describing its three main modules and respective data structures: the sketch module, the forecasting module, and the change detection module.

Sketch Module - The Sketch Module is responsible for storing and updating the observed sketch, $S_o(t)$, a variant of the traditional sketch counting data structure, called k-ary sketch, that summarizes per-flow information about the traffic in each epoch. Similarly to the count sketch [34], the k-ary sketch consists of an $H \times K$ table of registers. Each row of the k-ary sketch is associated with an independent 4-universal hash function h_i. The k-ary sketch supports four different operations: UPDATE to update the sketch, ESTIMATE to estimate the value for a given key, ESTIMATEF2 to estimate the second moment $F2$, and COMBINE to compute the linear combination of multiple sketches. These operations supported by the data structure are used across the different modules of the sketching algorithm. The UPDATE operation is used by the Sketch Module to update the sketch, as illustrated in Figure 3.1 for an input k with value v. In this example, where $H = 3$, the key element k is hashed three times with the different hash functions to update cells with the value v in the respective rows.

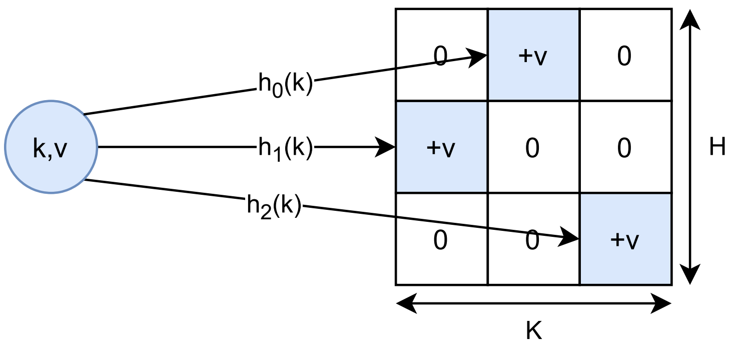


Figure 3.1: Illustration of the UPDATE operation on a k-ary sketch.

Forecasting Module - The Forecasting Module uses the observed sketches from previous epochs to create a forecast sketch $S_f(t)$ and, by comparing it to the observed values, builds the error sketch $S_e(t)$ for the current epoch. The forecasting module leverages the COMBINE operation on the sketch data structures to build different forecasting models which can be implemented on top of k-ary sketches. Hence, these models essentially leverage the linear property of the sketches, which allows linear combinations of multiple sketch data structures by combining all of their elements. For example, smoothing models, like the Exponentially Weighted Moving Average (EWMA) or the s-shaped moving Average (SMA), attribute decreasing weights to past sketch values in order to generate predictions that are weighted sums of past observations. Those models, originally proposed in [18], include four simple smoothing models, and two ARIMA models. The smoothing models are the moving average (MA), the s-shaped moving average (SMA), the exponentially weighted moving average (EWMA), and the non-seasonal holt-winters (NSHW).

Change Detection Module - The Change Detection Module leverages the error sketch $S_e(t)$ to detect significant changes across two consecutive epochs. First, this module computes an alarm threshold T_A based on the estimated second moment of the Error Sketch $F2$ and on an application-specific configurable parameter T. Afterwards, it estimates the error E_k associated with each key observed in the current epoch and compares that with the computed T_A. Whenever the estimate for a key is greater than the T_A threshold, the corresponding flow is considered to have changed significantly since the previous epoch.

3.3 Requirements for change detection

Based on the analysis of the k-ary, we set the following requirements for the K-MELEON module for online traffic change detection:

- **R1:** Achieve the same level of change detection accuracy as the k-ary sketch, including the detection of mice flows;
- **R2:** Achieve equivalent resource usage to the k-ary sketch;
- **R3:** Perform online analysis of traffic to enable near real-time responsiveness;
- **R4:** Offer fast responses (ideally, at sub-second time scales);
- **R5:** Achieve high throughput (Tbps).

K-MELEON aims to fulfill all of the above requirements by following a stream-based processing approach to the detection of traffic changes. In a nutshell, K-MELEON computes and stores the sketch and forecast values of k-ary, continuously with the packet stream, *inside* the data plane of a modern commodity switch [21].

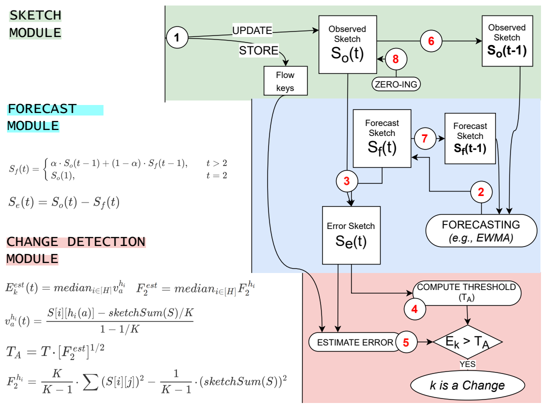


Figure 3.2: High-level block diagram of k-ary.

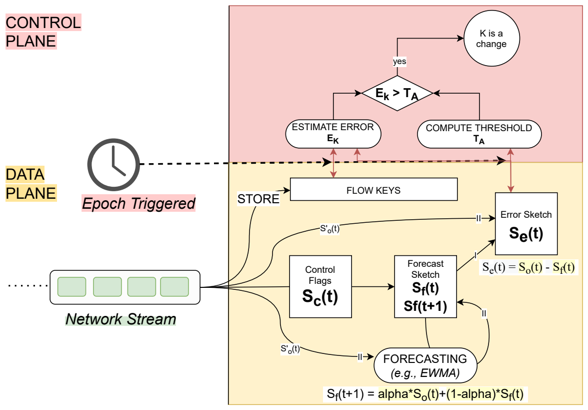


Figure 3.3: High-level block diagram of K-MELEON.

3.4 Challenges with on-line change detection

Running the logic of the k-ary sketching algorithm in the data plane of P4-capable target raises several issues, since many of its operations are not natively supported by the P4 programming model (the reader can check Section 2.1.2 for a recap about this model). First, the entire sketch data structure $S_o(t)$ must be traversed at once for the processing steps performed at the end of each epoch, while P4 enforces a limited, packet-by-packet and not event-driven [65] concurrent memory access, since that is common across high-throughput packet processing architectures [20]. Second, complex arithmetic is required by the forecasting model and for the computation of the estimates of frequency moments by the detection module, while P4 only allows for simple packet processing instructions [66].

For the sake of illustration, we have dissected the k-ary sketch algorithm and marked with numbers its main processing steps in Figure 3.2. The processing steps $(2-8)$, performed at the end of each epoch, involve operations not supported in P4. In detail, unsupported operations are required for the computation of the forecast sketch $S_f(t)$ (step 2), the computation of the error sketch $S_e(t)$ (step 3), the computation of the alarm threshold T_A (step 4), the estimate of the observed keys E_k (step 5), the copy of the observed and forecast sketches to be used as past values in the next epoch (steps 6 and 7), and the reset of the observed sketch $S_o(t)$ (step 8). Updating the observed sketch $S_o(t)$ and storing the flow keys (step 1), steps which are performed per packet, can be expressed in P4.

3.5 Stream-based change detection: K-MELEON

A high-level illustration of K-MELEON is shown in Figure 3.3 alongside the original k-ary algorithm. In summary, K-MELEON adapts the k-ary's sketch and part of the forecasting modules to run entirely inside the data plane as a P4 program, while, at present, still offloading the operations of the k-ary's change detection module to software in the control plane. Throughout the design of K-MELEON we have tested two of the different forecasting models originally proposed in the k-ary (see Section 3.2) for which we obtained very similar results, namely EWMA and NSHW. This design choice was driven by the computation and the storage requirements of these two models which, after a careful study, seemed to be the best fit for the targeted programming model. After an initial phase of testing, our design focused on the EWMA forecasting model. In fact, while EWMA performed very similarly to NSHW in all of our tests, a result that confirmed the conclusions from the original k-ary paper [18], it only requires to store two sketches from a previous epoch, rather than four (for smoothing and trend) as required by NSHW. Therefore, the design here presented focuses on the EWMA to illustrate the mechanics of K-MELEON.

3.5.1 Computing the Error and Forecast Sketches

The k-ary algorithm detects changes at every epoch, that is, every t time units. To this end, the algorithm collects summary information in the observed sketch over the current epoch and computes the error between the forecast and the observed values at the end of it. K-ary performs this operation all at once in a batch-based fashion, at the end of each epoch (see Figure 3.2). More specifically, at the end of each epoch, the k-ary must perform a linear subtraction of two sketches: the observed and the forecast. Furthermore, before computing this sketch, the k-ary must first compute the forecast sketch, which also requires performing the linear combination of the previous observed sketch and the previous forecast sketch. Performing the aforementioned operations in a batch-based fashion at the end of each epoch turns to be infeasible within the constraints of P4 programmable switches. Therefore, in what follows, we detail how we have converted these operations into equivalent ones, with some degree of approximation, that can be implemented in a stream-based fashion, per packet, within the constraints of the target P4-capable platform.

We start by presenting the equations required to compute the forecast (using EWMA), $S_f(t)$, and the error, $S_e(t)$, sketches:

$$S_e(t) = S_o(t) - S_f(t) \tag{3.1}$$

$$S_f(t) = \begin{cases} \alpha \cdot S_o(t-1) + (1-\alpha) \cdot S_f(t-1), & t > 2 \\ S_o(1), & t = 2 \end{cases} \tag{3.2}$$

As it can be seen from equation (3.1), $S_f(t)$ is required to compute $S_e(t)$. Also, while computing $S_e(t)$ only requires values from the current epoch t, computing $S_f(t)$ (see equation (3.2)) requires values from $t-1$, which means that $S_f(t)$, $S_f(t-1)$, $S_o(t)$, and $S_o(t-1)$ must be stored somewhere in the data plane at every epoch t, according to our requirements.

For $t > 2$, however, it should be noticed that $S_f(t)$ can be computed already at the beginning of epoch t, because it only uses sketch values from the epoch $t-1$. This means that, although $S_f(t)$ is required to compute $S_e(t)$, these sketches can be computed at different times t_i. Namely, $S_f(t+1)$ can be computed at the end of each epoch t according to the following:

$$S_f(t+1) = \begin{cases} \alpha \cdot S_o(t) + (1-\alpha) \cdot S_f(t), & t > 1 \\ S_o(1), & t = 1 \end{cases} \tag{3.3}$$

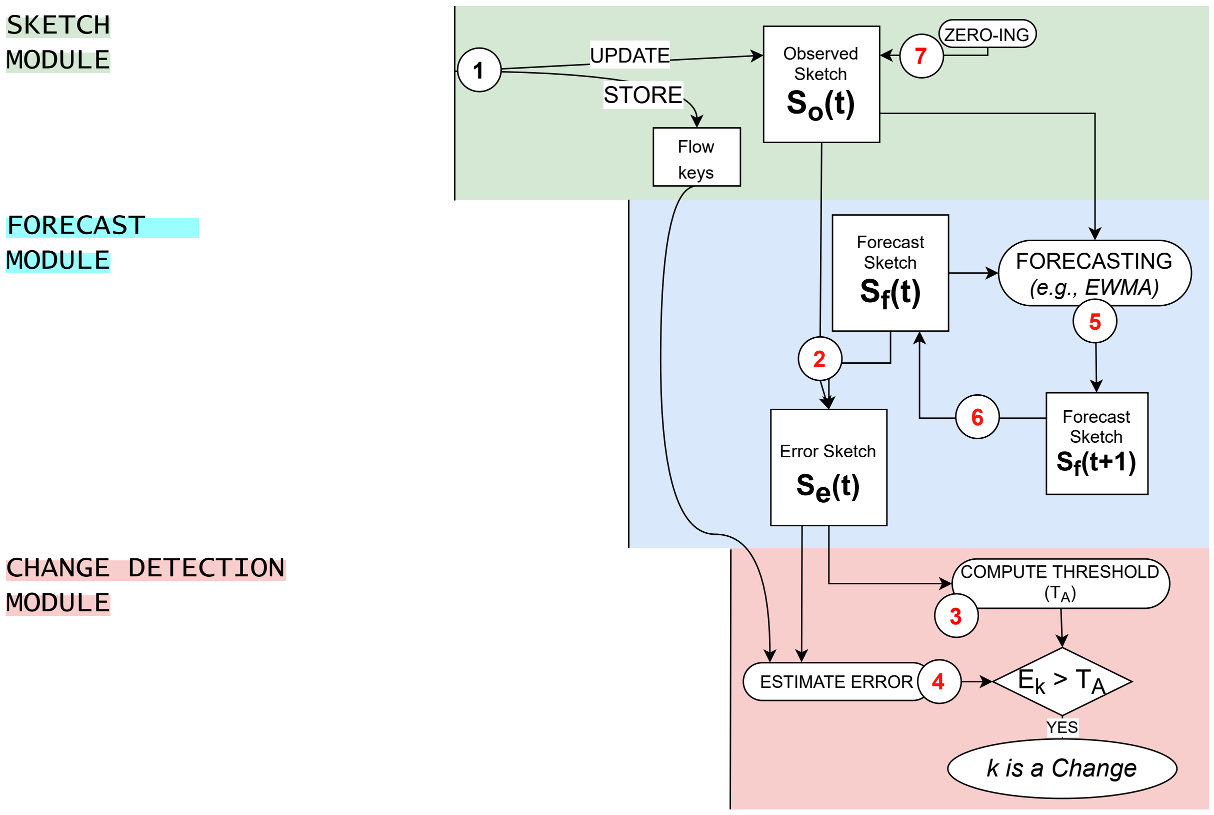


Figure 3.4: K-ary algorithm after applying the transformation in equation 3.3.

As it can be noted, the new equation (3.3) requires to store only the current observed and forecast sketches $S_o(t)$ and $S_f(t)$ to compute the forecast sketch for the next epoch $t{+}1$. Therefore, the additional $S_f(t-1)$ and $S_o(t-1)$ required by the original formula (3.2) do not need to be stored anymore after this transformation.

Figure 3.4 illustrates the transformations brought by equation (3.3), with respect to the original algorithm in Figure 3.2 and rearranged according to the processing steps $(2-8)$. More specifically, the error sketch is now computed in step 2, followed by the computation of the alarm threshold T_A (step 3), the estimate of the observed keys E_k (step 4), the computation of the forecast sketch for the following epoch $S_f(t+1)$ (step 5), the copy of the forecast sketch to be used in the following epoch (step 6), and the reset of the observed sketch $S_o(t)$ (step 7). However, even with the transformation in equation (3.3), $S_f(t+1)$ and $S_e(t)$ still need to be computed in a batch-based fashion.

To overcome the batch-based operation for the error sketch and the forecast sketch, it is sufficient to consider at first the error sketch equation (3.1) at time t, for which $S_o(t)$ and $S_f(t)$ are required. As shown previously, $S_f(t)$ can be computed at the end of the previous epoch $t{-}1$, so only $S_o(t)$ must be computed at time t. By its own nature, the observed sketch $S_o(t)$ is already computed in a streaming-based fashion, in fact it is incrementally updated with each packet traversing the switch. Thus, let $S_o^{'}(t, p_i)$ be the update in epoch t for a packet $p_i, i \in [1 : p(t)]$, where $p(t)$ is a function that returns the total number of packets in a given epoch t. The computation of the observed sketch can be expressed like:

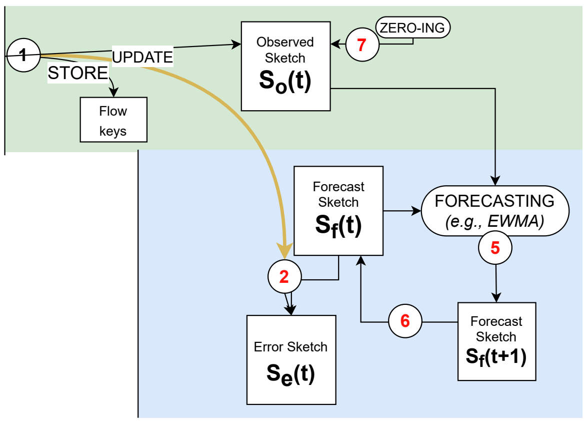


Figure 3.5: High-level block diagram of the k-ary algorithm using the incremental update of the error sketch described in equation (3.5).

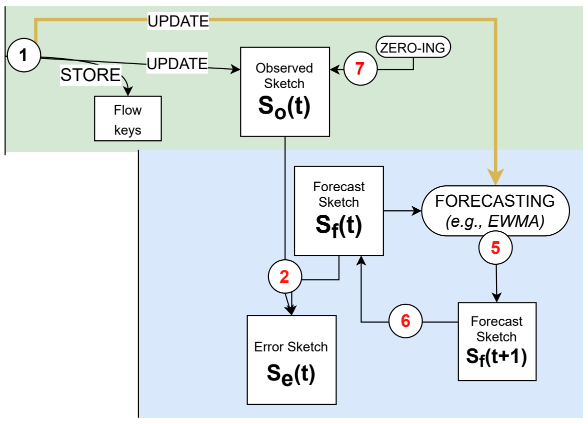


Figure 3.6: High-level block diagram of the k-ary algorithm using the forecast update described in equation (3.6).

$$S_o(t) = \sum_{i=0}^{p(t)} S_o^{'}(t, p_i) \tag{3.4}$$

Now, by replacing the $S_o(t)$ term in the error sketch equation (3.1) with the equivalent expression from equation (3.4), a new formula to compute the error sketch can be obtained in (3.5). It is evident from (3.5) that the error sketch for a given epoch can be indeed computed incrementally with each packet, just like the observed sketch, rather than being computed all at once at the end of that epoch.

$$S_e(t) = \sum_{i=0}^{p} S_o^{'}(t, p_i) - S_f(t) \tag{3.5}$$

Figure 3.5, illustrates the resulting algorithm, leveraging the above stream-based computation of the error sketch.

With regard to the forecast sketch for the next epoch $t + 1$, $S_f(t + 1)$ is computed via a linear combination of the same two sketches: the observed and the forecast sketch (see equation 3.2). Hence, by following this reasoning for the forecast sketch - replacing the $S_o(t)$ term in the error sketch equation (3.1) with the equivalent quantity from equation (3.4), a new forecast sketch equation that can be incrementally updated with each packet from the input stream is obtained.

$$S_f(t+1) = \begin{cases} \alpha \cdot \sum_{i=0}^{p} S_o^{'}(t, p_i) + (1-\alpha) \cdot S_f(t), & t > 1 \\ \sum_{i=0}^{p} S_o^{'}(t, p_i), & t = 1 \end{cases} \tag{3.6}$$

An illustration of this new mechanism for the computation of the forecast sketch $S_f(t+1)$ is presented in Figure 3.6. By leveraging equations (3.5) and (3.6) to compute respectively the error sketch $S_e(t)$ and the forecast sketch $S_f(t + 1)$, the computation of those sketches can be finally performed in a stream-based fashion.

As it can be observed in Figure 3.7, all observed values $S_o^{'}(t, p_i)$ are now updated directly into the forecast and error sketches, so it is no more necessary to store $S_o(t)$ in a separate structure. After the

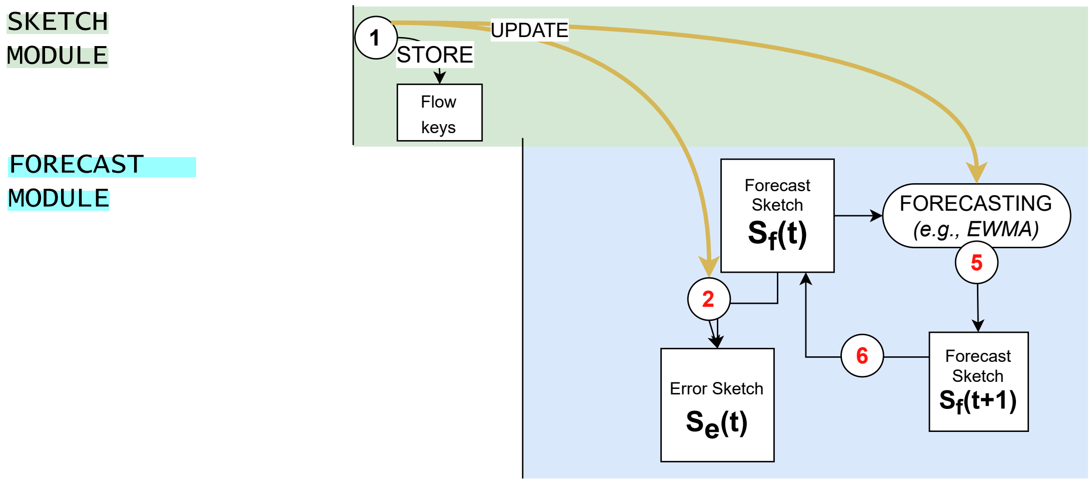


Figure 3.7: High-level block diagram of the k-ary algorithm after fully integrating the changes in the equations (3.5) and (3.6).

above modifications, K-MELEON must store three sketches at each epoch t: $S_e(t)$, $S_f(t)$, and $S_f(t+1)$.

The new equations (3.5) and (3.6) consist of two different terms, one related to the observed values in $[t-1, t]$ and one related to the forecast sketch at time t. The former term is computed incrementally with each packet, while the latter is computed at time $t-1$. This implies that a mechanism to prevent the second term of the equations from being considered multiple times, at each update $S_o^{'}(t, p_i)$ in $[t-1, t]$, must be enforced incrementally to compute (3.5) and (3.6) correctly. K-MELEON achieves that by introducing a control-flag sketch $S_c(t)$ containing 1-bit flags to be checked at each update of $S_e(t)$ and $S_f(t+1)$, meant to indicate whether or not the values of $S_f(t)$ have been already copied to $S_e(t)$ and $S_f(t+1)$ in the current epoch. More specifically, K-MELEON flips the corresponding control-flag sketch cell whenever the respective $S_f(t)$ value is copied with a certain packet p_i so that any subsequent check of that flag in the current epoch for any packet p_j with $j > i$ will indicate that the respective forecast value has already been copied. Figure 3.8 illustrates how the newly introduced control-flag sketch operates together with the other sketch data structures to perform the right updates.

This figure shows that the forecast $S_f(t+1)$ and error $S_e(t)$ sketches are still updated with the observed values for each packet but the forecast sketch $S_f(t)$ is only copied if the corresponding control flag in $S_c(t)$ enables it.

By observing Figure 3.8, it is also worth noting that the distinction between sketch module and forecast module, originally present with k-ary, is not present in K-MELEON. In fact, the corresponding values of $S_o(t)$ are directly stored into the forecast sketch $S_f(t+1)$ and the error sketch $S_e(t)$. Besides, $S_f(t+1)$ and $S_f(t)$ have been so far logically represented as two separate data structures. However, since values from $S_f(t)$ are only read once to update $S_f(t+1)$, the two sketches can be stored in the same structure that incrementally changes from $S_f(t)$ to $S_f(t+1)$ in the current epoch according to equation (3.6).

In summary, during each epoch t, K-MELEON needs to store three sketches: $S_e(t)$, $S_f(t)$ (which incrementally becomes $S_f(t+1)$), and $S_c(t)$.

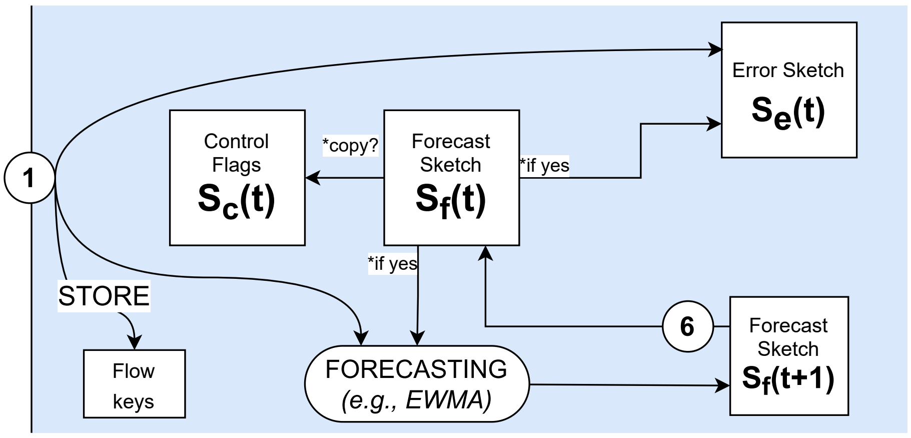


Figure 3.8: First High-level block diagram of the K-MELEON's data plane algorithm.

3.5.2 Fake Updates

The check on the control-flag sketch $S_c(t)$ ensures that values of the forecast sketch $S_f(t)$ are copied to other sketches only once per epoch. However, in the design presented so far, there is no mechanism that guarantees that *all* the values from $S_f(t)$ are copied into $S_f(t+1)$ for a certain epoch. In fact, if no packet hashes to specific buckets of the forecast sketch $S_f(t)$ during a certain epoch, the corresponding buckets will not be considered in the computation of the forecast sketch $s_f(t+1)$. This problem is better illustrated in the upper part of Figure 3.9. In the lower part, the figure also shows how K-MELEON aims at overcoming this problem by performing additional (”fake”) updates of the forecast sketch per packet.

This figure shows an epoch with only three packets, for illustration. In this example, the color blue is used to indicate changes introduced by the update function; the color red is used to indicate changes introduced by the fake updates we propose here, and the color grey is used to indicate collisions during the update operation. Without the fake update mechanism, any 0s in the control sketch $S_c(t)$ at the end of the epoch indicate values of the forecast sketch $S_f(t)$ which have not been considered in the computation of $S_f(t+1)$ for the current epoch. To address this, in K-MELEON we update one column of the sketch with each update, storing only one counter that indicates which column of the sketch to update next.

K-MELEON thus performs h additional copies of $S_f(t)$ into $S_f(t+1)$ at each packet update. As the number of updates is typically small (h represents the number of hashes used), these updates can be performed in a programmable switch at line rate, by performing one update per stage, in h stages. Similarly to a normal update operation, the control flag in $S_c(t)$ is first checked to decide whether or not to perform this copy into $S_f(t+1)$. If the buckets referenced by the counter have already been copied, no extra operation is performed for the current packet.

The fake update mechanism requires a number of packets per epoch of at least the width of the sketch to work as intended. If the number of packets in a certain epoch is lower than the width of the sketch data structure, then the number of additional copies will not be sufficient to consider the entire forecast sketch $S_f(t)$ for the computation of $S_f(t+1)$. However, in practice the number of packets per epoch is much higher than the width of the sketch.

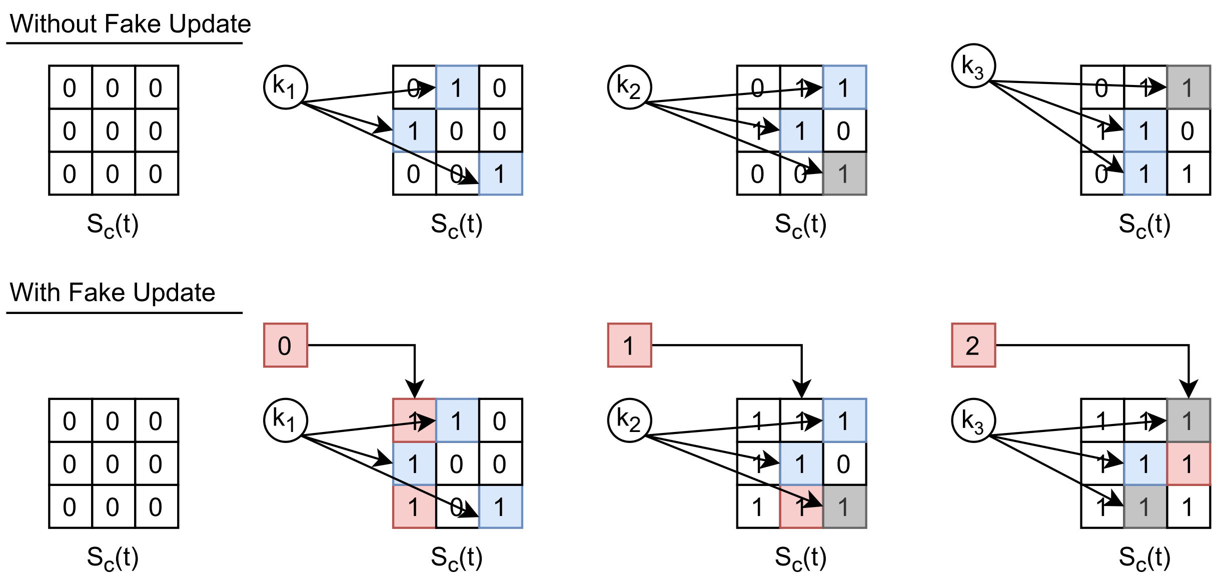


Figure 3.9: Comparison between the K-MELEON algorithm with and without the fake update mechanism.

3.5.3 Reading sketches consistently

The modifications of the k-ary algorithm introduced in Section 3.5.1 allow K-MELEON to run the sketch and forecast modules in a stream-based fashion in the data plane of a P4-capable target. However, as they are, they may also possibly introduce some consistency issues when it is necessary to interact with the control plane to perform change detection.

In detail, K-MELEON stores two sketch structures, $S_f(t)$ and $S_e(t)$, which are incrementally updated with each packet. Read operations from the control plane are much slower when compared to the packet processing happening in the data plane. Yet, a consistent snapshot of the error sketch is required by the control plane to perform change detection at the end of each epoch. By consistent we mean that the values stored in the error sketch fetched by the control plane should all belong to the same epoch. However, with the read operation happening at a slower speed, the error sketch fetched may contain values updated in the current epoch. Therefore, the data plane design of K-MELEON must feature an intrinsic mechanism to allow the control plane to perform *atomic* read operations of the error sketch data structure. This operation could easily be done in a batch-based fashion in k-ary, but in a streaming-based solution one cannot stop processing traffic to entirely fetch the error sketch from the control plane.

K-MELEON addresses this consistency issue by storing two instances of the error sketch data structure in the data plane: in each epoch one instance is meant to be updated while the other, which contains the error sketch from the previous epoch, is read only. K-MELEON uses a 1-bit mutable epoch flag that indicates the instance to be updated and the one to be read at each epoch. Figure 3.10 illustrates this mechanism through an example. In epoch $t = 2$ in the figure, the epoch flag is zero, and only the sketch on the left $S_e(2)$ is updated, while $S_e(1)$ is currently read. In the following epoch ($t = 3$), the epoch flag changes and, as a consequence, the sketch on the left $S_e(2)$ is offloaded and the sketch on the right starts to be updated with new values, becoming $S_e(3)$.

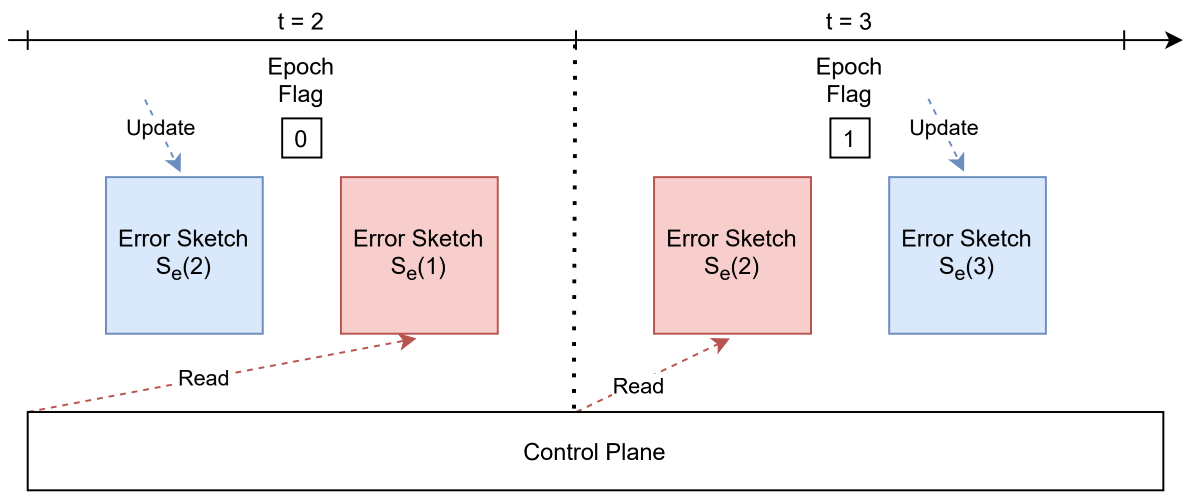


Figure 3.10: Illustration of the K-MELEON's consistency mechanism.

3.5.4 Approximating Floating-Point Arithmetic

The k-ary sketch proposed six forecasting models to build $S_f(t)$ inside the forecasting module. These forecasting models use operations (namely floating-point multiplications) and/or require storing additional data structures (e.g., to carry on smoothed past values), which are extremely challenging to implement inside the data plane of P4-programmable targets. For these reasons, and based on analysis performed throughout this work, in K-MELEON we implement only the Exponentially Weighted Moving Average (EWMA). In fact, EWMA presents a good trade-off between computational and storage resources required and prediction results achieved, when compared to other models originally proposed with k-ary.

Applying EWMA, the computation of the forecast sketch in the current epoch is obtained as $S_f(t+1) = \alpha \cdot S_o(t) + (1-\alpha) \cdot S_f(t)$, where $S_f(t)$ is the forecast sketch computed in the previous epoch and $S_o(t)$ is the observed sketch in the current epoch. This formula requires floating-point arithmetic due to the multiplicative factor α $\{\alpha \in \mathbb{R} \,|\, 0 \leq \alpha \leq 1\}$. Since floating-point arithmetic is not supported in P4 [35], K-MELEON resorts to converting the floating-point multiplications of $S_o(t)$ and $S_f(t)$ to multiplication via bit-shifts. Because α takes values between 0 and 1, this operation uses only **right** bit-shifts. However, the multiplication via right bit-shifts is limited, as it can only represent multiplications by negative powers of two, 2^{-x}, where x is the number of bit-shifts. Thus, α can only take values which can be represented as negative powers of two. An example of the transformations required by this method for the operation $\alpha \cdot S_o(t)$ is shown in equation (3.7), with the right bit-shift operation represented with the symbol $\gg$.

$$\alpha = 2^{-x} \ , \ \alpha \cdot S_o(t) = 2^{-x} \cdot S_o(t) \approx S_o(t) \gg x \tag{3.7}$$

In Figure 3.11, we illustrate how the multiplication by $\alpha = 0.5$ and $\alpha = 0.25$ can be computed using right bit-shifts as described in equation (3.7). As should be clear, only a very limited number of useful alpha values can be computed with right bit-shifts. For example, there is no way to compute the multiplication using $\alpha > 0.5$ with this technique alone. K-MELEON allows, however, selecting any α which can be decomposed into a sum of negative powers of two. Figure 3.12 shows how different α values,

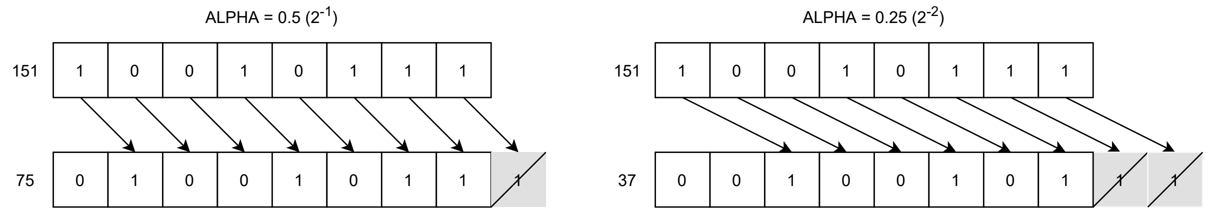


Figure 3.11: Multiplying number 151 by 0.5 (on the left) and by 0.25 (on the right), using right bit-shifts.

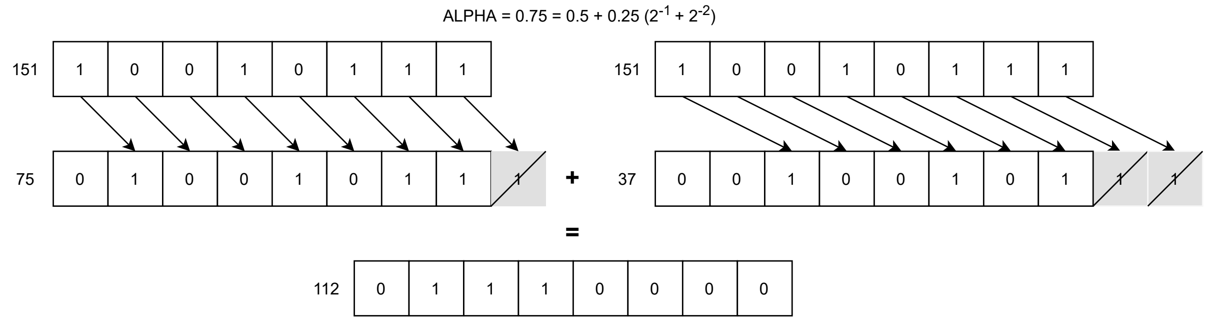


Figure 3.12: Bit-shift multiplication of 151 using $\alpha = 0.75$.

higher than 0.5, can be computed this way. In this example, we perform the multiplication via bit-shifts for $\alpha = 0.75$. Since $\alpha = 0.75$ cannot be computed directly with right bit-shifts, we decompose it into a combination of values which can be expressed as negative powers of two. In this case, $\alpha = 0.75$ is decomposed into $0.25 + 0.5$. We compute these values separately with right bit-shifts, and sum their results afterwards.

As it is well known, binary multiplication by **right** bit-shifts is not entirely accurate, since the remainder is lost. Because of this error, and since the observed values must be multiplied by alpha (see equation 3.6), the unitary update of the observed values ($S_o^{'}(t)$) becomes zero when using bit-shifts (for any α). We solve this problem by performing updates of 2^x, where x is the highest number of bit-shifts required to perform this operation. For example, when using $\alpha = 0.75$ we perform at most 2 bit-shifts, so the each update ($S_o^{'}(t)$) would be of 4 (2^2) instead of unitary.

3.5.5 The Role of the Control Plane

At the end of each epoch, K-MELEON needs to estimate the error E_k associated with each flow-key and to compute the alarm threshold T_A. These operations (steps 3 and 4 of the original k-ary algorithm, in Figure 3.2) are not trivial to be performed into the data plane of P4-programmable switch. Given the strict time budget available for the thesis, we leave this as future work.

The current version of K-MELEON thus offloads the error sketch $S_e(t)$ and the flow-keys stored during each epoch to a controller application responsible for performing change detection.

After offloading, the controller performs change detection by performing the following steps. First, it computes the alarm threshold T_A (see equations (3.8), (3.9), and (3.10)). Then it estimates the error associated with each of the flow-keys (see equations (3.11) and (3.12)). Finally, if the estimated error for a key is higher than the alarm threshold, that key is considered a change.

The controller does not implement any logic for epoch verification, so it cannot predict when an epoch changes. Consequently, K-MELEON requires checking the data plane epoch flag to verify if an epoch has changed by comparing it with the most recent epoch flag in the controller. If a different value is read, it means the epoch has changed and the error sketch and the flow-keys can be offloaded. The periodicity for reading the epoch flag from the data plane can be confirmed by the operator.

$$T_A = T \cdot [F_2^{est}]^{1/2} \tag{3.8}$$

$$F_2^{est} = median_{i \in [H]} F_2^{h_i} \tag{3.9}$$

$$F_2^{h_i} = \frac{K}{K-1} \cdot \sum (S[i][j])^2 - \frac{1}{K-1} \cdot (sketchSum(S))^2 \tag{3.10}$$

$$E_k^{est}(t) = median_{i \in [H]} v_a^{h_i} \tag{3.11}$$

$$v_a^{h_i} = \frac{S[i][h_i(a)] - sketchSum(S))^2}{1 - 1/K} \tag{3.12}$$

3.6 Summary

In this chapter, we presented the design of K-MELEON, an on-line solution for traffic change detection within the constraints of P4-programmable targets. First, we introduced the problem definition for on-line traffic change detection, presented in detail the state-of-the-art k-ary off-line solution, and highlighted the practical challenges of porting such solution into modern programmable switching hardware for on-line detection of traffic changes. Subsequently, we presented the design of K-MELEON. The reformulation proposed was driven by restrictions on the target programmable platform to allow K-MELEON to convert the batch-based k-ary into a new stream-based solution.

The next chapter dives into the implementation details of K-MELEON, presenting the main data plane program as well as other important auxiliary software developed in this work.

Chapter 4

Implementation

This chapter describes the steps undertaken to build and validate a P4 implementation of K-MELEON.

Figure 4.1 presents an overview of the main software components developed as part of this work. A trace analyser performs some basic measurements (packet rate, number of flow keys, etc.) on a given traffic trace, which can be optionally used to determine the several configuration parameters required by a change detection module. The change detection modules, k-ary and K-MELEON, take user-defined configuration parameters as input, and output the changes detected to a *.out file. Finally, a compare-tool can be used to compare the outputs from the k-ary and the K-MELEON for the same configuration parameters, computing certain statistics about detection and helping evaluate the proposed algorithm. Because P4 is a domain-specific language specifically designed for network data forwarding, a direct conversion from a general purpose language such as C or python cannot, most often, be trivially obtained. As such, this kind of translation is approximate and introduces errors. The above components have been developed with the main goal of testing the correctness of the P4 implementation of the change detection module, comparing the results achieved by such a module with the ones achieved with the original k-ary algorithm. Since no open-source version of the k-ary algorithm was available, a Python version of k-ary was developed by following the original description of the algorithm in [18], and made available as open-source software by this work.

The rest of this chapter is organized as follows. Section 4.1 describes the implementation of the k-ary sketch in python, and explains each of the various constraints and decisions made during this process. Section 4.2 presents the overall structure of the K-MELEON program, detailing the integration of operations and approximations from the design to the implementation in P4. To conclude, Section 4.3 describes all the additional software components developed to validate, test, and evaluate each of the change detection modules.

4.1 The k-ary in a high-level language

An implementation of k-ary has been developed as the first step of this work, by closely following the description of the k-ary algorithm provided in [18]. This implementation is written in Python 3.9.6

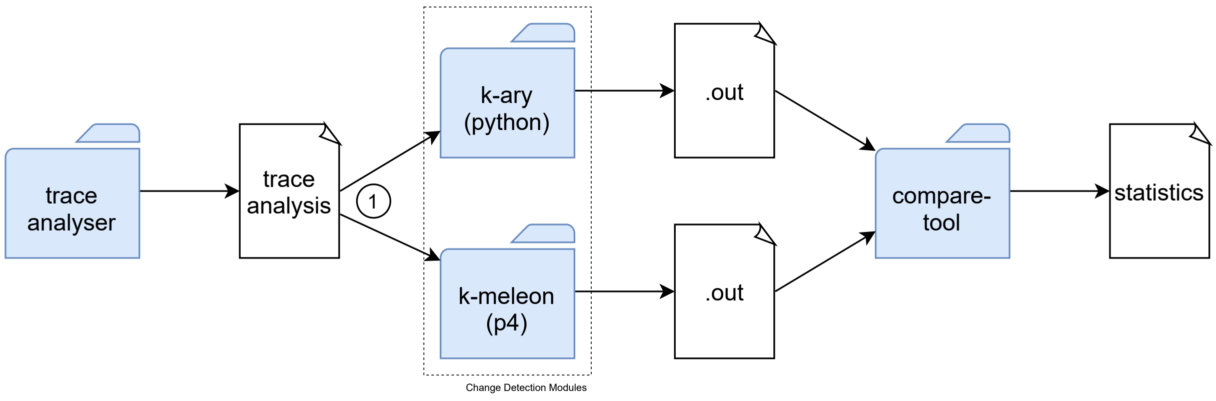


Figure 4.1: Overview of the main software components developed within this project on traffic change detection.

and consists of approximately 900 LoC. Its source-code files are illustrated in Figure 4.2. It requires the following python libraries to be compiled and executed: *scapy* (for packet processing), *mmh3* (for murmur3 hashing), and *statistics* (for statistical analysis).

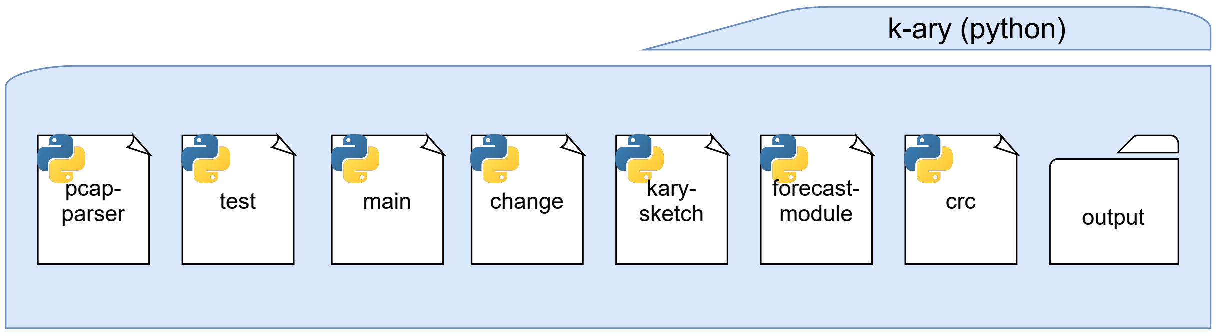


Figure 4.2: Overview of the k-ary source code.

The Sketch Module (kary-sketch.py) contains the sketch data structure, meant to store the sketch data and support the set of related operations. The k-ary sketch is implemented through a $list[H][K]$ (matrix) of values, where H represents the number of hash functions (the height of the sketch) and K represents the number of buckets per row (the width of the sketch). The main operations supported by this sketch are the update of selected buckets of the sketch data structure (UPDATE), the estimate of the error for a certain key (ESTIMATE), and the estimate of the sketch second moment (ESTIMATEF2). The code for these operations is reported in Listing 4.1. This implementation of the k-ary sketch currently supports two different hash functions: murmur3 and crc32. The choice of the hash function to apply is driven by a configuration parameter of the detection module. The UPDATE function receives as parameters a flow key k, an update value v, and an identifier of the hash function as a string. For each row of the sketch, it hashes k with the given hash function and obtains the index of the bucket to be updated with the value v.

The k-ary sketch is supposed to identify the keys responsible for certain changes at detection time.

This property of a sketch is known as reversibility. To be reversible, the k-ary sketch must somehow store the flow keys seen in every epoch. Therefore, the UPDATE function also stores all the keys seen in one epoch into a dedicated data structure (lines 9 and 10).

```python
1  def UPDATE(self,key,value,hash_func):
2      for i in range(0,self.depth):
3          bucket = None
4          if hash_func == "crc32":
5              bucket = binascii.crc32(str.encode(','.join(key)),self.seeds[i])%self.width
6          elif hash_func == "murmur3":
7              bucket = mmh3.hash64(','.join(key),self.seeds[i])[0]%self.width
8          self.sketch[i][bucket] = self.sketch[i][bucket] + value
9      if key not in self.keys:
10         self.keys.append(key)
11
12 def sum(self,row):
13     return sum(self.sketch[row])
14
15 def ESTIMATEF2(self):
16         result = []
17         for i in range(0,self.depth):
18             aux = 0
19             for j in range(0,self.width):
20                 aux = aux + (self.sketch[i][j]**2)
21             result.append(((self.width/(self.width-1))*aux) - ((1/(self.width-1))*(self.
       sum(i)**2)))
22         return median(result)
23
24 def ESTIMATE(self,key,hash_func):
25     result = []
26     for i in range(0,self.depth):
27         bucket = None
28         if hash_func == "crc32":
29             bucket = binascii.crc32(str.encode(','.join(key)),self.seeds[i])%self.width
30         elif hash_func == "murmur3":
31             bucket = mmh3.hash64(','.join(key),self.seeds[i])[0]%self.width
32         result.append( (self.sketch[i][bucket] - (self.sum(i)/self.width)) / (1 - (1/self
       .width)))
33     return median(result)
```

Listing 4.1: k-ary sketch operations in python.

The ESTIMATEF2 and ESTIMATE functions both require summing all of the values in the sketch, so the code features a common function $sum()$ for that purpose, returning the sum of all values for a given row of the sketch. Next, the respective equations are translated into functions which traverse the whole sketch, row by row.

The Forecasting Module (forecast-module.py) leverages the observed values in the current epoch and the forecast values from the past to build a forecast and an error sketch at the end of each epoch. In this implementation, those forecast models which presented characteristics more suitable to a data plane implementation with the P4 language have been selected. Hence, for example, smoothing models like EWMA or NSHW have been preferred over the ARIMA models (the reader can refer to Section 3.2 on Chapter 3 for a complete list of the models presented in [18]). Listing 4.2 reports the forecast models implemented in python.

```python
def EWMA(previous_forecast_sketch,previous_observed_sketch,alpha):
    depth = len(previous_observed_sketch.sketch)
    width = len(previous_observed_sketch.sketch[0])
    new_forecast_sketch = KAry_Sketch(depth,width)
    if previous_forecast_sketch != None:
        for i in range(0,depth):
            for j in range(0,width):
                new_forecast_sketch.sketch[i][j] = (alpha*previous_observed_sketch.sketch
    [i][j]) + ((1-alpha)*previous_forecast_sketch.sketch[i][j])
        return new_forecast_sketch
    else:
        return copy.deepcopy(previous_observed_sketch)

def NSHW(previous_forecast_sketch,previous_observed_sketch,observed_sketch,previous_trend
    ,previous_smoothing,alpha,beta):
    depth = len(previous_observed_sketch.sketch)
    width = len(previous_observed_sketch.sketch[0])
    smoothing_sketch = KAry_Sketch(depth,width)
    trend_sketch = KAry_Sketch(depth,width)

    #smoothing
    if previous_forecast_sketch != None:
        for i in range(0,depth):
            for j in range(0,width):
                smoothing_sketch.sketch[i][j] = (alpha*previous_observed_sketch.sketch[i
    ][j]) + ((1-alpha)*previous_forecast_sketch.sketch[i][j])
    else:
        smoothing_sketch = copy.deepcopy(previous_observed_sketch)

    #trend
    if previous_forecast_sketch != None:
        for i in range(0,depth):
            for j in range(0,width):
                trend_sketch.sketch[i][j] = (beta*(smoothing_sketch.sketch[i][j] -
    previous_smoothing.sketch[i][j])) + ((1-beta)*previous_trend.sketch[i][j])
    else:
        for i in range(0,depth):
            for j in range(0,width):
                trend_sketch.sketch[i][j] = observed_sketch.sketch[i][j] -
    previous_observed_sketch.sketch[i][j]

    #Forecasting sketch
    forecasting_sketch = KAry_Sketch(depth,width)
    for i in range(0,depth):
            for j in range(0,width):
                forecasting_sketch.sketch[i][j] = trend_sketch.sketch[i][j] + smoothing_sketch.sketch[i][j]
    return forecasting_sketch, smoothing_sketch, trend_sketch
```

Listing 4.2: Forecasting models support in the python implementation of k-ary.

As a caveat, the original description of the k-ary sketch also included a COMBINE operation to perform linear transformations on the sketch data structure. Since every forecast model performs those combinations in a slightly different way, this implementation, however, does not include a unique COMBINE function and rather includes the linear transformations required by each specific module's function definition.

The Change Detection Module - (change.py) source code is shown in Listing 4.3. After building the error sketch (lines $2-8$), the Change Detection module is responsible for computing the application threshold (line 9) (see section 3.2 on Chapter 3) and the error associated with each key in the list from the sketch Module (lines $14-20$), as described in the previous chapter. After detecting a change it sends an alarm (line 20) and adds its associated metadata to the corresponding log (line 19).

```
1  def change(forecast_sketch,observed_sketch,T):
2      depth = len(observed_sketch.sketch)
3      width = len(observed_sketch.sketch[0])
4  
5      new_error_sketch = KAry_Sketch(depth,width)
6      for i in range(0,depth):
7          for j in range(0,width):
8              new_error_sketch.sketch[i][j] = observed_sketch.sketch[i][j] - forecast_sketch.sketch[i][j]
9      TA = T * sqrt(new_error_sketch.ESTIMATEF2())
10     return new_error_sketch, TA/10
11 (...)
12     error_sketch, threshold = change(forecast_sketch,sketch_list[-1],T)
13 (...)
14     for key in keys:
15         if not any(v is None for v in key):
16             estimate = error_sketch.ESTIMATE(key,hash_func)/10
17             if estimate > threshold:
18                 complex_res.append(key + (str(estimate),))
19                 res.append(key)
20                 print("Change detected for:", key, "with estimate:", estimate)
```

Listing 4.3: Change detection module functionality in the python implementation of k-ary.

4.1.1 The Main Program

The main program used for running the k-ary algorithm in python consists of two main parts. The first part, in **main.py**, is responsible for parsing and storing the configuration parameters from the input and for creating the output file with the results from the execution. A list of all user-defined configuration

Parameter	Values
alpha and beta	[0.0:1.0]
sketch depth	[1:15[
epoch size (s)	[0:300[
forecasting model	{ewma,nshw}
hashing function	{murmur3,crc32}
flow-key	{srcIP, srcPort, dstIP, dstPort, proto}
path for the input trace	string
number of past sketches saved	[1:5]
threshold	[0.0:1.0]

Table 4.1: Configuration parameters available through the main program main.py.

parameters available through main.py is shown in Table 4.1. These configuration parameters allow the user to customize the size of the sketch data structure, to select a specific forecasting model and specify the associated parameters, and to define other configuration options such as the size of the epoch, the hash function to use, or the fields of the flow-key.

The output file produced by main.py is meant to store information about changes detected in the processed traffic trace. This file stores information per epoch. The format prints three distinct lines per epoch. More specifically, the first line contains a sequential identifier, the application threshold, and the number of packets processed. The second line presents a (possibly empty) list of flow-keys with significant changes and the corresponding error estimates. The third line shows the number of distinct flow-keys observed. An example of such output file format is presented in listing 4.4.

```
Epoch: 1        Threshold: 43.31       Num Packets: 2021
[('149.171.36.239','192.168.1.248','54.85'),('52.8.186.218','192.168.1.241',54.85)]
Num Keys: 33
Epoch: 2        Threshold: 55.84       Num Packets: 3180
[('192.168.1.106','35.224.42.73',65.52)]
Num Keys: 31
```

Listing 4.4: Example of an output file showing information about changes detected in two consecutive epochs.

The second part, in **(change.py)**, implements the main logic of the k-ary sketching algorithm. This part parses incoming packets, implements the epoch control, and leverages the different sketch modules to perform change detection at every epoch. Epoch control is performed through packets timestamp. When a packet timestamp exceeds the current epoch interval, the change detection module is triggered. First, the Forecasting Module is called to compute the forecast and error sketches. Secondly, the computed error sketch is sent to the Change Detection Module, which checks for relevant changes in the current epoch. After the change detection operations are completed, all the detected changes and associated metadata are sent to the main.py program for logging purposes. Finally, the sketch data structures are prepared for a new epoch to start.

4.1.2 Exploring the configuration space

One of the first steps in the evaluation of this solution required identifying the best configurations for different kinds of changes or traffic characteristics. This configuration-parameter search is implemented inside **test.py**. This program takes one traffic trace and the expected detection results as input, and outputs, out of a certain number of tested configurations, the parameters which provide the best results in terms of accuracy, precision, and recall. To achieve this, the program basically executes multiple instances of the k-ary using different configurations, and compares the obtained results. Performing such an extensive search of the configuration space for the k-ary algorithm is not a trivial problem. It consumes time (1), requires defining a ”reasonable” interval of values to test (2), and, in the case of continuous parameters, cannot fully explore all possible values (3).

The **multiprocessing** [67] python package has been used to minimize the time consumed (1) with this task. This package allows splitting the processes, which can be run in parallel, thus saving time. Tests with 2, 4, 6 and 12 processes confirmed the expected gain against the execution on a single process on a 6-core COTS machine, with hyper-threading enabled.

Issues (2) and (3) have been addressed by reducing the set of possible values for certain configuration parameters leveraging specific knowledge of the problem domain. For example, flow-keys can be of the format [dstIP], or [srcIP,dstIP], or the five-tuple [srcIP,srcPort,dstIP,dstPort,protocol]. Continuous values which take values between 0 and 1, on the other hand, are split into decimal steps of 0.1, and the best configurations were then further refined with increments of 0.01.

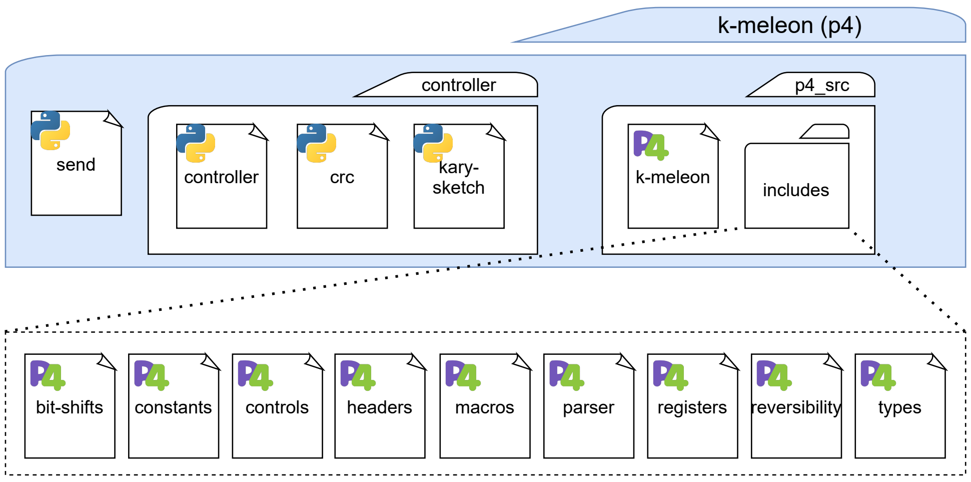


Figure 4.3: Overview of the K-MELEON implementation code.

4.2 K-MELEON

The implementation of K-MELEON can be described referring to the separation between control and data plane logic of this algorithm, which is illustrated in Figure 4.4. The data-plane processes build the sketch’s data structures at line rate. The implementation of K-MELEON developed by this work targeted, and has been tested on, the P4 software reference switch a.k.a. bmv2 [68]. The control plane logic is responsible for retrieving the keys and the error sketch from the data plane, and for performing change detection. The overview of the source code for this implementation is illustrated in Figure 4.3.

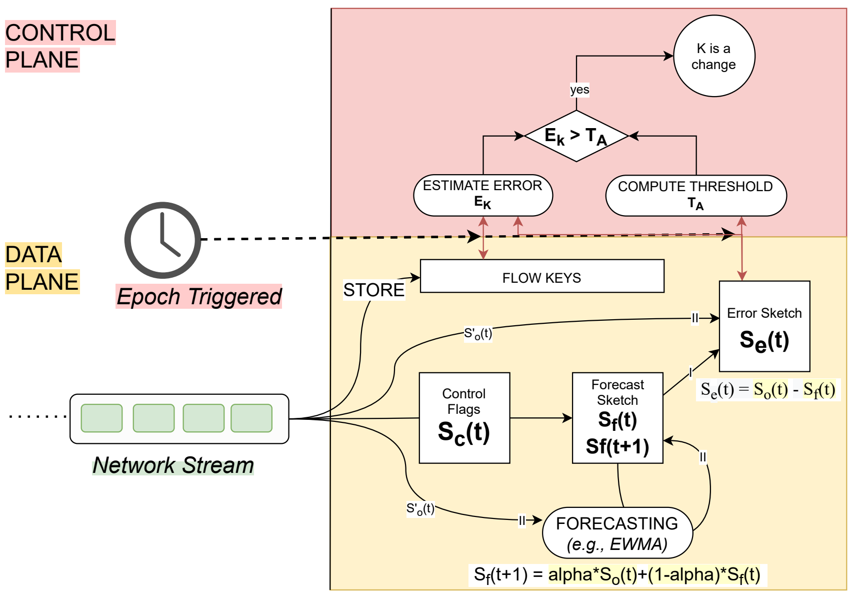


Figure 4.4: High-level block diagram of K-MELEON.

4.2.1 Data Plane

The K-MELEON P4 program for the *bmv2* switch target architecture consists of less than 500 LoC. The data structures required by the K-MELEON algorithm are implemented through registers stateful memory available on the target architecture. A snippet of code with the register definition in p4 can be seen in Listing 4.5.

```
register<int<32>>(SKETCH_WIDTH) reg_forecast_sketch_row0;    // forecast sketch Sf(t)
register<int<32>>(SKETCH_WIDTH) reg_forecast_sketch_row1;
register<int<32>>(SKETCH_WIDTH) reg_forecast_sketch_row2;

//       [ - - Se(t) (Read) - - | - - Se(t) (Write) - - ]
register<int<32>>(SKETCH_WIDTH*2) reg_error_sketch_row0;       // error sketch Se(t)
register<int<32>>(SKETCH_WIDTH*2) reg_error_sketch_row1;
register<int<32>>(SKETCH_WIDTH*2) reg_error_sketch_row2;

register<bit<1>>(SKETCH_WIDTH)  reg_controlFlag_sketch_row0;   // control-flag Sc(t)
register<bit<1>>(SKETCH_WIDTH)  reg_controlFlag_sketch_row1;
register<bit<1>>(SKETCH_WIDTH)  reg_controlFlag_sketch_row2;
```

Listing 4.5: K-MELEON's sketch data structures definition in P4.

Feed-forward pipelined switch architectures, such the one of Tofino-based switches [21], have specific constraints with respect to the number of actions we can perform per stage, of the match-action table, as well as constraints which impede us from reading or writing from the same register in more than one stage. At the same time, each sketch needs to be updated multiple times for each packet, depending on the number of sketch rows we need to update. Thus, performing the update operation while using only one register to represent a sketch, would require to access the same register in H consecutive stages (where H is the number of rows in one sketch), operation which would be infeasible. Hence, by using a different register to represent each of the sketch rows, the K-MELEON algorithm does not require to access multiple times the same registers when updating the different sketches' rows. Furthermore, to guarantee that atomic read and reset operations can be performed by the control plane between consecutive epochs (see section 3.5.3) the array of registers for $S_e(t)$ has been allocated double size. Using this register in a mirrored fashion allows using only half of the sketch for reading and the other half for writing. To determine which parts of the sketch is being read and which ones are being written, this solution uses a *epoch-bit* with an associated offset.

The epoch verification is the first operation performed in the P4 program. A code snippet of this operation is reported in Listing 4.6. This operation is meant to check whether or not an incoming packet belongs to the current epoch or rather triggers an epoch change. This can be calculated with the number of packets processed (lines 4-5), or as a time interval (lines 7-8). When using the number of packets to define the size of an epoch, a register stores the number of packets processed for the current epoch. When using a time interval instead, the timestamp of the first packet in each epoch is stored into a register and all the subsequent packets timestamps are checked against the $timestamp + epochsize$ value. A timestamp greater than the $timestamp + epochsize$ values triggers an epoch change (lines 15 and 19) and flips the one-bit sketch flag for the error sketch register.

```
1  reg_epoch_value.read(meta.epoch_value,0);
2  #ifdef EPOCH_PKT /* epoch calculated with the number of packets processed */
3  if ( meta.epoch_value >= EPOCH_SIZE) {
4    reg_epoch_value.write(0,1); //reset packet counter for the current epoch
5  #elif EPOCH_TS /* epoch calculated as time interval */
6  if ( (standard_metadata.ingress_global_timestamp - meta.epoch_value) >= EPOCH_SIZE) {
7    reg_epoch_value.write(0, standard_metadata.ingress_global_timestamp);
8  #endif /* EPOCH TS Or PKT */
9  
10   // start new epoch by flipping the epoch bit
11   reg_epoch_bit.read(meta.epoch_bit,0);
12 
13   if (meta.epoch_bit == 0) {
14     meta.epoch_bit = 1;
15     reg_epoch_bit.write(0,meta.epoch_bit);
16   } else {
17     meta.epoch_bit = 0;
18     reg_epoch_bit.write(0,meta.epoch_bit);
19   }
20   # flag to signal that epoch has changed
21   meta.epoch_changed_flag = 1;
22 }
```

Listing 4.6: Epoch verification mechanism for K-MELEON in P4.

Sketch Update - After the correct epoch is selected, the program proceeds with the update of the sketch data structures. Analogous operations are performed onto the h sketches rows, so for the sake of illustration, only operations related to a single row are presented here. Each of these h update is divided into two parts. Listing 4.7 shows the P4 code for the first part of the update function of the first row of each sketch. The function starts by reading and checking the control flag (line 1-2) to check whether or not the sketch values from the previous epoch have been overwritten. If so, it updates the forecast and error sketches with the observed value, without copying values from the previous epoch (lines 3-18). Otherwise, the function copies the values from the previous forecast sketch and resets the error sketch before updating (lines 20-32).

```
1  reg_controlFlag_sketch_row0.read(meta.ctrl,meta.hash0);
2  if (meta.ctrl != meta.epoch_bit) {  // If diff, copy forecast_sketch
3      reg_controlFlag_sketch_row0.write(meta.hash0,meta.epoch_bit);   // Flip control flag
4  
5      reg_forecast_sketch_row0.read(meta.forecast,meta.hash0);    // Sf(t)
6  
7      // update error
8      meta.new_err = SKETCH_UPDATE - meta.forecast;               // S'o(t) - Sf(t)
9      // Se(t) = S'o(t) - Sf(t)
10     reg_error_sketch_row0.write(meta.hash0+meta.err_offset,meta.new_err);
11 
12     // update forecast
13     meta.obs = SKETCH_UPDATE >> 1;                      // alpha*S'o(t)
14     meta.aux_forecast = meta.forecast >> 1;             // (1-alpha)*Sf(t)
15     meta.new_forecast = meta.obs + meta.aux_forecast;   // alpha*S'o(t) + (1-alpha)*Sf(t)
16 
17     // Sf(t+1) = alpha*S'o(t) + (1-alpha)*Sf(t)
18     reg_forecast_sketch_row0.write(meta.hash0,meta.new_forecast);
19 } else {    // else, only update with observed
20     // update error
21     reg_error_sketch_row0.read(meta.err,meta.hash0+meta.err_offset);    // Se(t)
22     meta.new_err = meta.err + SKETCH_UPDATE;                            // Se(t) + S'o(t)
23     // Se(t) = Se(t) + S'o(t)
24     reg_error_sketch_row0.write(meta.hash0+meta.err_offset,meta.new_err);
25 
26     // update forecast
27     reg_forecast_sketch_row0.read(meta.forecast,meta.hash0);    // Sf(t+1)
28     meta.obs = SKETCH_UPDATE >> 1;                              // alpha*S'o(t)
29     meta.new_forecast = meta.obs + meta.forecast;               // Sf(t+1) + alpha*S'o(t)
30 
31     // Sf(t+1) = Sf(t+1) + alpha*S'o(t)
32     reg_forecast_sketch_row0.write(meta.hash0,meta.new_forecast);
33 }
```

Listing 4.7: P4 logic to update sketch rows in K-MELEON.

The second part of the update function performs one extra operation per packet, called fake update, to entirely copy the current forecast sketch Sf(t) into the error sketch $S_e(t)$ for the correct computation of the forecast sketch $S_f(t+1)$ in the following epoch. This behavior is shown in the code snippet of Listing 4.8. This requires storing into a counter the number of columns of the sketch data structure which have already been updated by the fake update in the current epoch. The function starts by reading and checking (lines 1-2) if the counter is greater than the width of the sketch. Whenever such a condition holds, the sketches have been completely updated, therefore, no fake update is required within the current epoch. Otherwise, the function increments the counter (line 3) and checks if the control flag associated with that counter has already been updated (line 4-5). If none of the previous conditions apply, it proceeds to updating the sketch data structures (lines 6-18) with the forecast sketch term of the corresponding formulas.

```
1  reg_extraOp_counter.read(meta.counter,0);
2  if (meta.counter < SKETCH_WIDTH) {
3      reg_extraOp_counter.write(0,meta.counter+1);
4      reg_controlFlag_sketch_row0.read(meta.ctrl,meta.counter);   // Sc(t)
5      if (meta.ctrl != meta.epoch_bit) { // If diff, copy forecast_sketch
6          // Sc(t) = !Sc(t)
7          reg_controlFlag_sketch_row0.write(meta.counter,meta.epoch_bit);
8          reg_forecast_sketch_row0.read(meta.forecast,meta.counter);  // Sf(t)
9
10         // update error
11         meta.new_err_op = -meta.forecast;   // -Sf(t)
12         // Se(t) = -Sf(t)
13         reg_error_sketch_row0.write(meta.counter+meta.err_offset,meta.new_err_op);
14
15         // update forecast
16         meta.new_forecast = meta.forecast >> 1; // (1 - alpha)*Sf(t)
17         // Sf(t+1) = (1 - alpha)*Sf(t)
18         reg_forecast_sketch_row0.write(meta.counter,meta.new_forecast);
19     }
20 }
```

Listing 4.8: Fake Update algorithm for K-MELEON in P4.

Reverting the sketch - K-MELEON needs to store the keys which contributed to the values of the error sketch. Since storing the flow keys of all the flows which traversed the switch in one epoch could not be easily feasible on common target architectures for the K-MELEON program, the majority vote algorithm (MJRTY) proposed in [17] is leveraged by K-MELEON. The implementation of the majority vote algorithm in K-MELEON is reported in Listing 4.9. The snippet only presents the code applied to the first row of the sketch when the *epoch-bit* value is zero. Analogous code is executed across all the sketch's rows. Registers are required for storing the keys and the counters (lines 4-10). Similarly, as done for the other sketches, each sketch row for MJRTY is represented as, and stored into, a distinct register. These register structures are also replicated to allow consistent reading operations from the controller (see Section 3.5.3 in Chapter 3). Each sketch cell stores three values: the counter, the source IP address, and the destination IP address inside the same row. This MJRTY algorithm starts by setting the offset required to update the source and destination addresses in the sketch register (lines 14-15). It checks the *epoch-bit* (line 17) to decide which sketch to update, and then reads all values from the sketch, which are associated with the current key (lines 18-20). Then, the algorithm checks if the flow key is the same as the one stored inside the sketch (line 21). If so, it increments its counter (lines 30-32). Otherwise, it decrements the counter (line 27-28) and, if the counter reaches zero, it replaces the stored key with the new one (lines 22-25).

```
1  - - - - - - - - - - - | REGISTER DEFINITION | - - - - - - - - - - -
2  
3  //  [ - - Counters - - | - - SRC IPs - - | - - DST IPs - - ]
4  register<bit<32>>(SKETCH_WIDTH*3) reg_mjrty0_row0;      // mjrty (Read)
5  register<bit<32>>(SKETCH_WIDTH*3) reg_mjrty0_row1;
6  register<bit<32>>(SKETCH_WIDTH*3) reg_mjrty0_row2;
7  //  [ - - Counters - - | - - SRC IPs - - | - - DST IPs - - ]
8  register<bit<32>>(SKETCH_WIDTH*3) reg_mjrty1_row0;      // mjrty (Write)
9  register<bit<32>>(SKETCH_WIDTH*3) reg_mjrty1_row1;
10 register<bit<32>>(SKETCH_WIDTH*3) reg_mjrty1_row2;
11 
12 - - - - - - - - - - - - - | ALGORITHM | - - - - - - - - - - - - -
13 
14 meta.src_offset = SKETCH_WIDTH;
15 meta.dst_offset = 2*SKETCH_WIDTH;
16 //compare candidate flow key with current flow key
17 if (meta.epoch_bit == 0) {
18     reg_mjrty0_row0.read(meta.tempsrc, meta.hash0+meta.src_offset);
19     reg_mjrty0_row0.read(meta.tempdst, meta.hash0+meta.dst_offset);
20     reg_mjrty0_row0.read(meta.tempcount, meta.hash0);
21     if (meta.tempsrc!= hdr.ipv4.srcAddr || meta.tempdst != hdr.ipv4.dstAddr) {
22         if (meta.tempcount == 0){ //if counter is zero, add new key
23             reg_mjrty0_row0.write(meta.hash0+meta.src_offset, hdr.ipv4.srcAddr);
24             reg_mjrty0_row0.write(meta.hash0+meta.dst_offset, hdr.ipv4.dstAddr);
25             reg_mjrty0_row0.write(meta.hash0, 1);
26         } else if (meta.tempcount > 0) { //if counter is not zero decrement counter by 1
27             meta.tempcount = meta.tempcount - 1;
28             reg_mjrty0_row0.write(meta.hash0, meta.tempcount);
29         }
30     } else { // if keys are equal increment counter by 1
31         meta.tempcount = meta.tempcount + 1;
32         reg_mjrty0_row0.write(meta.hash0, meta.tempcount);
33     }
```

Listing 4.9: Majority Vote Algorithm for sketch reversibility in K-MELEON.

4.2.2 Control Plane

The control plane logic of K-MELEON is written in Python. The controller uses the SimpleSwitchAPI from the *p4utils* library to communicate with the data plane module. The controller program is divided into three different modules, each module corresponding to a different file. One module implements the main functionality of the controller, namely, initializing the data plane hash functions with pre-defined polynomials, retrieving sketches and flow keys from the data plane, and performing change detection. Another module mimics the behavior of the custom crc32 algorithm running inside the target switch architecture (bmv2). This is needed because this specific hash is the one used in the P4 switch, so we need to perform the hashing computations with this same function. The third module manages the sketch data structures and supports the required operations for those structures.

K-MELEON leverages the control plane to compute the alarm threshold T_A, the estimated error E_k for each key k, and to compare the estimated E_k against T_A. For that purpose, a controller reads the error sketch $S_e(t)$ including the heavy changer flow keys from the reversibility mechanism. Whenever an error estimate for a key E_k is greater than T_A, a change is detected for the flow key k. Besides, after retrieving the data necessary to the detection, the control plane cleans the corresponding data structures allocated for the error sketch, the forecast sketch and the flow keys stored in the data plane to ensure a clean status for the next epoch.

The controller periodically reads the *epoch-bit* flag to understand when epochs change. In Listing 4.10, we show how the data plane elements are extracted from the data plane by the controller. The controller first reads the *epoch-bit flag* (line 2) and checks that it equals the one currently stored (line 3). If the stored epoch bit corresponds to the one just read, the controller aborts the operation since the current epoch has not changed. Otherwise (line 4), the controller updates the stored epoch-bit for the current epoch, and proceeds with reading the flow keys and error sketches row by row (lines 6-11). Afterwards, the controller converts the per-row information read from the data plane into their respective representations using the sketch module (lines 13-23). Finally, the controller resets all flow keys present in the data plane (lines 29-31) before performing change detection as defined before in the python implementation of the k-ary algorithm.

```
1  self.registers = []
2  self.registers.append(self.controller.register_read("reg_epoch_bit"))   #0 flag
3  if self.registers[0] != self.flag:
4      self.flag = self.registers[0]
5      if (self.registers[0][0] == 0): #choose error and mjrty
6          mjrty_row0 = self.controller.register_read("reg_mjrty1_row0")
7          mjrty_row1 = self.controller.register_read("reg_mjrty1_row1")
8          mjrty_row2 = self.controller.register_read("reg_mjrty1_row2")
9          err_row0 = self.controller.register_read("reg_error_sketch_row0")
10         err_row1 = self.controller.register_read("reg_error_sketch_row1")
11         err_row2 = self.controller.register_read("reg_error_sketch_row2")
12
13         self.registers.append(mjrty_row0[self.width:2*self.width])         #1 src ips
14         self.registers[1] = self.registers[1] + mjrty_row1[self.width:2*self.width]
15         self.registers[1] = self.registers[1] + mjrty_row2[self.width:2*self.width]
16
17         self.registers.append(mjrty_row0[2*self.width:3*self.width])       #2 dst ips
18         self.registers[2] = self.registers[2] + mjrty_row1[2*self.width:3*self.width]
19         self.registers[2] = self.registers[2] + mjrty_row2[2*self.width:3*self.width]
20
21         self.registers.append(err_row0[self.width:2*self.width])        #3 error sketch
22         self.registers[3] = self.registers[3] + err_row1[self.width:2*self.width]
23         self.registers[3] = self.registers[3] + err_row2[self.width:2*self.width]
24
```

```
        #reset mjrty keys and counters
        self.controller.register_reset("reg_mjrty1_row0")
        self.controller.register_reset("reg_mjrty1_row1")
        self.controller.register_reset("reg_mjrty1_row2")
```

Listing 4.10: Read and Reset operations of K-MELEON data structures by the python controller.

4.3 Additional tools for testing and evaluation

The evaluation and testing phases of the traffic change detection algorithm K-MELEON required the development of several ancillary tools. Specifically, those were mostly necessary to parse and analyze traffic traces, and to analyze and compare detection results.

Trace Parsing tool. This tool is essential to running and testing the python implementations of the k-ary and K-MELEON. Running each of these detection modules requires loading an entire packet trace to memory, which depends on the size of the trace and on the resources available on the machine used for testing. Because only the flow keys, the timestamp, and the size of the packet (for some configurations) must be stored, a parser tool only stores these values in an array. This tool also helps with testing multiple different configurations in parallel without unnecessarily overloading the compute and storage resources on the testing environment.

Comparison Tool. Testing the correctness of the k-ary and K-MELEON implementations requires comparing the changes detected by each of these solutions, including the thresholds, the number of packets processed, and the estimates obtained for each of the changes detected in each epoch. To facilitate the comparison of detection results with different algorithms, a custom tool was developed to compute several measurements, such as the false positives and negatives, true positives and negatives, and relative differences. The different algorithms share a similar format for their output files to ease the comparison of their detection results.

Analysis Tool. Some general knowledge of the traffic used for testing was shown to be important for the correct testing and evaluation of the developed algorithms. For example, if the packet rate for a given trace was in the order of the hundreds of packets per second, choosing a small epoch size might not collect enough information to perform the change detection task. By contrast, choosing a large epoch size might incur in missing relatively quicker changes such as microbursts, for example. To specifically aid with the aforementioned tasks, we developed an analysis tool. The tool takes a given network trace as input and outputs relevant information about the trace such as the packet rate, and graphs describing the changes in packet-rate across different time scales. Furthermore, this tool allows retrieving all flows which exceed a given threshold for a given time scale, or the heavy flows, those that constitute a certain percentage of the whole traffic. This last feature turns to be very useful to test different definitions of microbursts for a given network trace.

4.4 Summary

In this chapter, we presented the implementation steps of K-MELEON. First, we introduced the implementation of the k-ary sketch in python, described each of its modules, the main program, and the optimizations required for testing. Afterwards, we described the implementation of K-MELEON. The main obstacle was the constraint programming model of the target programmable switches, requiring careful analysis to understand which actions could be performed in the data plane and which should be offloaded to the controller. Finally, we introduced the additional tools created to aid in the testing and evaluation of this work.

The next chapter dives into the evaluation of K-MELEON, describing the testing environment, the validation of our k-ary implementation, and the evaluation of K-MELEON.

Chapter 5

Evaluation

The evaluation of K-MELEON consists of two main phases with different objectives. First, the aim is to assess if the original k-ary algorithm can detect changes with high fidelity. Second, the goal is to measure the accuracy in detection of K-MELEON, compared to its ancestor k-ary.

This Chapter is organized as follows. In Section 5.1 we describe the testing environment. Then, in Section 5.2 we verify the accuracy of change detection achieved by the original k-ary. Finally, in Section 5.3 we report and discuss the evaluation of K-MELEON with respect to the detection of traffic changes.

5.1 Testing Environment

Datasets. The evaluation of K-MELEON uses several packet traces from two different datasets. The CSE-CIC-IDS2018 dataset [69] can be used for identification of traffic changes in the context of network intrusion detection – from now onward we refer to this dataset as `network_attacks`. This dataset contains seven different categories of attacks (brute force, DoS, DDoS, etc.), that present different patterns of network traffic change. We thus use an attack as a proxy for a traffic change event. The dataset is labeled, helping assess the accuracy of detection of our approach. The second dataset used is the data-center measurement trace (UN1) from [70] - from now onward we refer to this dataset as the `microbursts` dataset. As recent work [71] has detected the presence of microbursts in this trace, the evaluation with this dataset has the goal to understand whether K-MELEON can distinguish short-lived traffic spikes within certain flows, commonly referred to as microbursts.

Experimental setup. There is not an open-source implementation of the k-ary algorithm proposed in [18], so we have implemented a version in Python that is made available as open-source software with this work. The effectiveness in detection of this implementation of k-ary was validated through extensive testing against the labelled traces from the `network_attacks` dataset. The data-plane part of K-MELEON was implemented in P4, while the control-plane logic was implemented in Python. The P4 reference compiler *p4c* [72] and the P4-enabled software switch *bmv2* [68] were used to compile and run K-MELEON. The Scapy python library [73] was used to replay the packet traces from the selected datasets. A set of additional Python scripts were developed to compare detection results with k-ary and K-MELEON and produce specific evaluation metrics. The evaluation was performed on a COTS machine with an Intel i7-8750H CPU and 16B of RAM, through a virtualized environment running Ubuntu 16.04. The source code of K-MELEON and all the auxiliary software developed for testing has been made publicly available on GitHub [24].

5.2 K-ary

The implementation of the k-ary sketching algorithm in python followed the description of the algorithm in the original paper [18]. This section aims to validate the k-ary implementation by extensively testing it against the `network_attacks` dataset. The goal is both to understand how effective is k-ary in detecting changes, and also to analyse how the different configuration parameters affect its performance. First, we evaluate the impact of the size of the sketch data-structure on the detection results obtained against per-flow analysis[1]. Then, we evaluate the k-ary with respect to one use case: the ability to detect network attacks. Finally, a series of tests are performed to understand the impact of each individual configuration parameter on the detection results.

5.2.1 Determining the sketch size

The k-ary sketch implementation was compared against a similar monitoring solution performing per-flow analysis without memory compression, considered as ground truth. We perform change detection with the two monitoring solutions using the traces from the `network_attacks` dataset. The metric used is the relative difference as defined in [18] (see equation 5.1). The equation compares, in each epoch, the alarm threshold values v_a obtained with k-ary, against the ground truth values v_e.

$$\text{Relative Difference} = \frac{v_a - v_e}{v_e} \cdot 100 \tag{5.1}$$

Storing per-flow information in memory is the ideal solution. In practice, sketching algorithms trade off monitoring accuracy for memory efficiency and so incur in collisions, storing information about multiple flows in the same memory. As a consequence, per-flow information is estimated when using sketches and, thus, the resulting values can introduce errors.

In k-ary, increasing the width (K) of the sketch data-structure decreases the probability of collisions of different flows into the same buckets, while increasing the height (H) improves the estimates of the sketch values. Therefore, the size ($H \times K$) of the sketch data-structure influences the relative error obtained by the algorithm. To measure the impact of the sketch size with regard to the monitoring task targeted by this work, we run k-ary against the `network_attacks` traces using different sketch widths. These results are reported in Figure 5.1.

As expected, using higher sketch widths (K) improves the performance of change detection of k-ary. In this test, sketch widths of 32 buckets incurred in a higher number of collisions and so produced a

[1]We use per-flow analysis as the baseline. This represents the switch keeping all information necessary to the detection of every change for every flow.

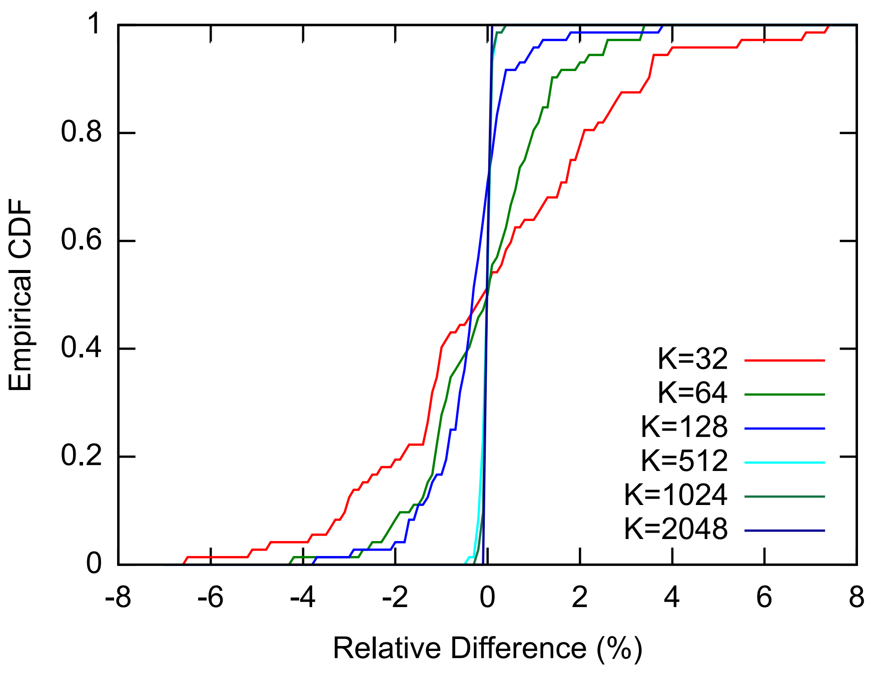


Figure 5.1: Empirical CDF values obtained for the relative difference between k-ary and per-flow analysis (ground truth) for different sketch widths (K), represented as the number of buckets per hash. The number of hashes (H) is equal to 3.

relative difference of 7% in the worst case. Instead, sketch widths of 512, 1024, and 2048 show relative differences lower than 0.5%, guaranteeing high fidelity in monitoring. Increasing the sketch width above 512 led to little to no improvement for this dataset, so this value represents a good memory/accuracy trade-off. It should be noted that the traffic packet rate in these testing traces is relatively low (around 100 packets/s), containing 50 to 100 different flows per epoch (we consider epoch size of 20s for these experiments). Higher packet rates and higher number of flows per epoch would require further memory.

A similar experiment was performed to measure the effect of changes in the height of the sketch data structure. Figure 5.2 reports the relative difference obtained for the k-ary sketching algorithm with different sketch heights (H), while using a fixed sketch width (K) of 2048 buckets. Five different height values in the range [1,10] were tested. As expected, values of H higher than 2 produce more accurate results, achieving relative differences under 0.1%. Increasing the height value (H) further has very little impact on the relative difference. As a result, the sketch height (H) was set to 3 for all the following experiments.

5.2.2 Use case: Attack Detection

One common use of a network change detection system is for the identification of network attacks. We resorted to the labelled attack traces from the `network_attacks` dataset to measure the effectiveness of the k-ary algorithm in detecting the attacks present in those traces.

This evaluation required tuning several configuration parameters for the k-ary algorithm, from the choice of the forecasting model (NSHW or EWMA), and the values of the smoothing constant in the

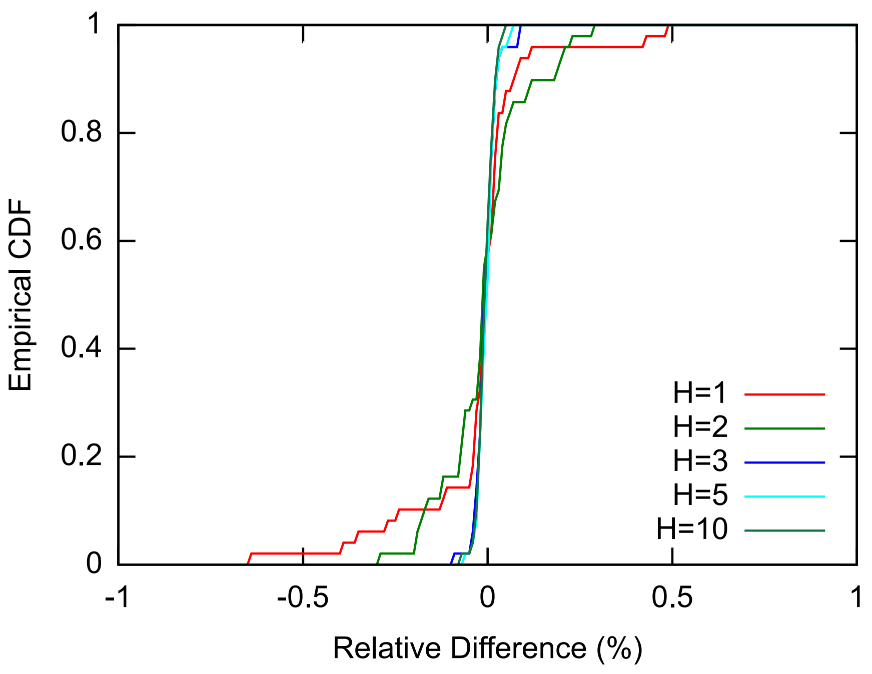


Figure 5.2: Empirical CDF values obtained for the relative difference between k-ary and per-flow analysis (ground truth) for different sketch heights (H). The sketch width is $K = 2048$.

chosen forecast model, to the values of the application-specific threshold (T) and the size of the epoch. We report about our exploration of the configuration space for k-ary next, in Section 5.2.3.

For this use-case, we tested epoch sizes between 5 and 20 seconds. We tested detection with k-ary across all of the 11 traces in the `network_attacks` dataset and report the accuracy obtained with the best configuration for each trace in Figure 5.3[2]. The k-ary algorithm produced high detection accuracy for all kinds of attacks featured in the traces. The algorithm has shown to achieve the best accuracy with epoch size of 20 seconds for this experiment given the low packet rate of the testing traces. Faster packet rates could most likely increase the accuracy even for shorter epoch sizes.

5.2.3 Exploring the configuration space for attack detection

Several parameters contribute to the detection accuracy achieved by the k-ary algorithm. Namely, the number (H) and width (K) of the hash tables (and hence the size $H \times K$ of the sketch data structure), the application-specific threshold T, the epoch size, the choice of the forecast model, and the model parameters. We have not explored this configuration space exhaustively in this work. Rather, we have first focused on finding configurations which on average worked well across the tested traces from the `network_attacks` dataset. Our main goal in this phase was to identify configurations which could fit into a more constrained environment, like the one targeted for the development of K-MELEON, which could work well on average.

[2]Accuracy is defined as the ratio between the number of correctly reported events (in other words, the sum of true positives and true negatives) and the total number of events. On these sections we only report accuracy values as we are interested in analysing the behaviour of the algorithm with different configurations. In the case of K-MELEON we will extend the analysis with precision and recall metrics.

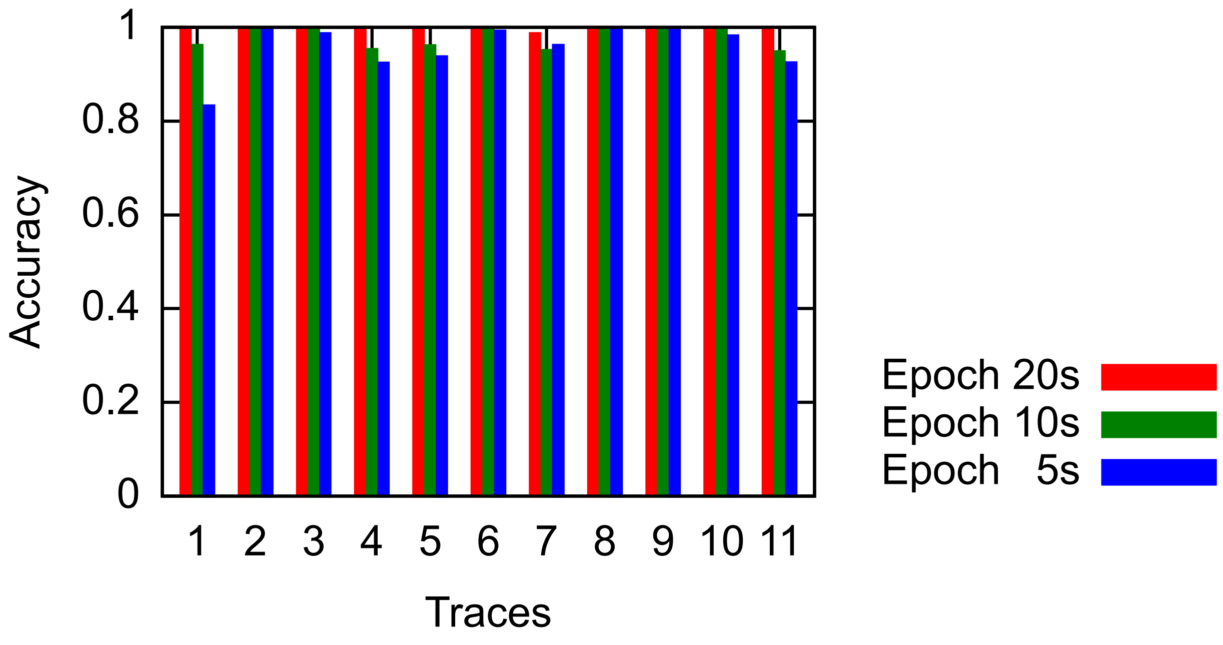


Figure 5.3: Empirical accuracy values obtained for each trace with the k-ary implementation in python.

Our experiments confirmed that the tuning of certain parameters may help improve detection accuracy.

Forecasting Model

We started testing two different forecasting models, namely, the Exponentially Weighted Moving Average (EWMA) and the Non-Seasonal Holt-Winters (NSHW). We aimed to understand if using NSHW, which requires more computation and memory, would give better results than using the simpler EWMA. The comparison between the two models was done by selecting the best accuracy results obtained with different parameter configurations, across all traces, for epoch sizes of 5, 10, and 20 seconds. Figure 5.4 reports the accuracy values obtained for those epoch sizes in a box plot, used here to show the variability of our results across the tested traces. Each box represents the interquartile range and the lines (whiskers) represent the variability of the values obtained outside the lower and upper quartiles. Dots represent outliers, which can be easily ignored for an overall interpretation of these results.

The results obtained suggest that the average accuracy obtained with both forecasting models is very similar. However, the accuracy with EWMA tends to decrease more rapidly than with the NSHW for smaller epoch sizes. Overall, although both forecasting models present slightly worse detection accuracy for smaller epoch sizes, these results show that on average they still detect changes with high accuracy. Based on these results, we decided to use EWMA as a forecasting model for K-MELEON, since this model can achieve comparable results with NSHW with lower computation and memory requirements.

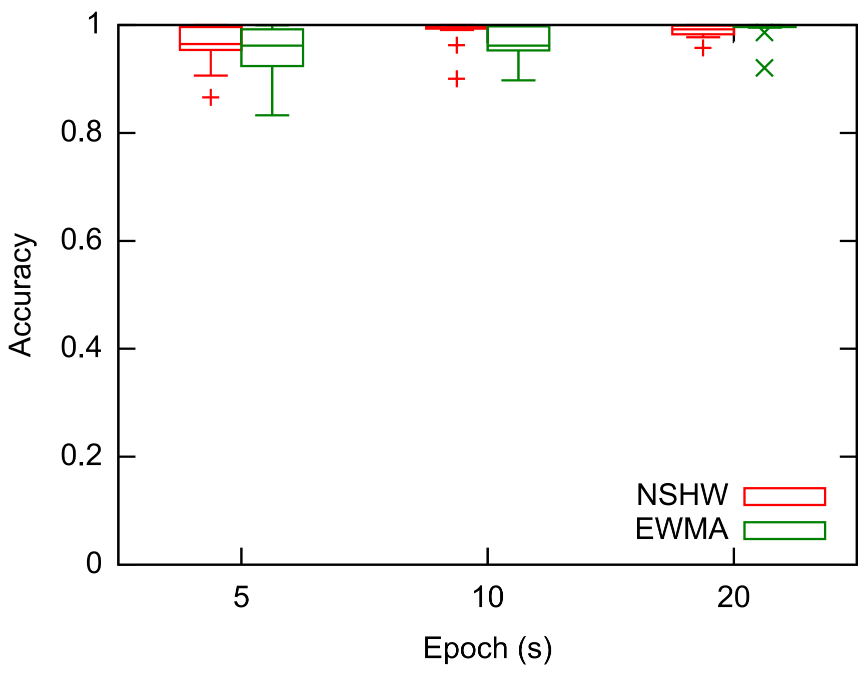


Figure 5.4: Empirical accuracy values obtained for each forecasting model using different epoch sizes.

Threshold

We are also interested in understanding how the application-specific threshold T can affect the accuracy of detection for different traffic changes. Since it is very unlikely that multiple instances of the algorithm can run at the same time, we tried to identify a single threshold value which could achieve detection with sufficient accuracy with all the tested traces. We leveraged the results from the previous experiments to identify the threshold values which provided the highest accuracy for the two forecasting models evaluated. Figure 5.5 illustrates the cumulative distribution function of the application-specific threshold for those experiments.

One configuration which matched all kinds of changes would require the use of a threshold value as low as 0.4, for EWMA, the lowest value obtained across all traces. The NSHW model, on the other hand, shows that it can use a higher threshold 0.5 for the same traces. However, the best configuration is different for different traces/attacks. For instance, threshold of 0.8 could be used for traces 7, 8, and 9, but a threshold of 0.5 would be better for the remaining traces. The aggregation of attacks by their type could prove useful in achieving higher accuracy across similar attacks using the same configurations. This is something to explore in the future.

Flow-keys

In our evaluation three kinds of flow-keys were tested: (1) destination IP and protocol, (2) source IP, destination IP, and protocol, and (3) the five-tuple key (source IP, source port, destination IP, destination port, and protocol). We are interested in understanding if performing ”coarser” monitoring, using smaller flow-keys, can still provide good results. Smaller flow-keys bring the benefit of lower memory requirements for storing flow information.

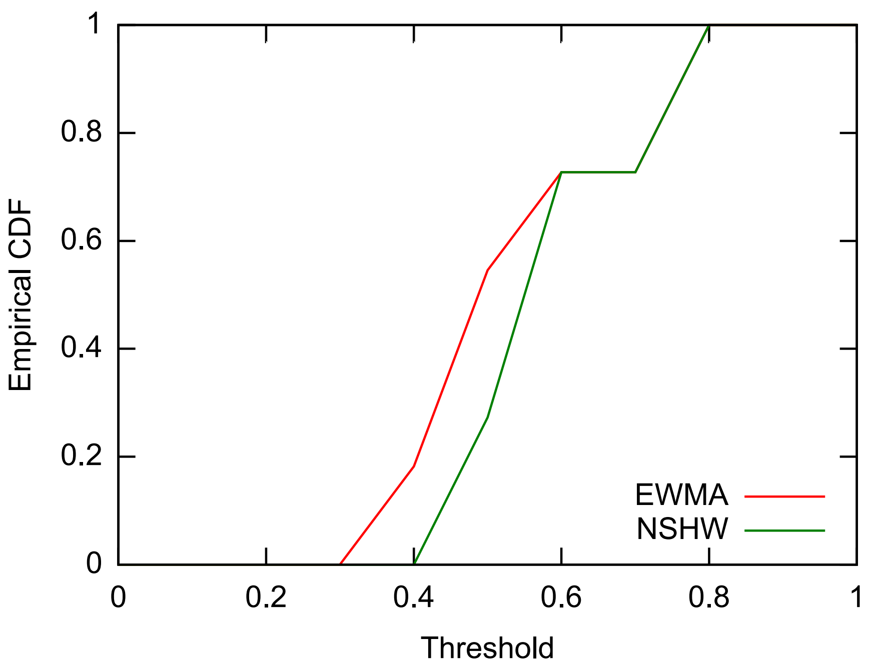


Figure 5.5: Empirical optimal threshold values obtained for each trace.

Trace	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
EWMA	5T	5T	5T	5T	5T	5T	5T	2T, 5T	2T, 5T	5T	5T
NSHW	5T	5T	5T	5T	5T	5T	2T, 5T	2T, 5T	2T, 5T	5T	5T

Table 5.1: Best flow-key type for each trace in the `network_attacks` dataset, where 5T means the five-tuple key and 2T means destination IP and protocol.

Table 5.1 reports the flow-key types which allowed the algorithm to achieve the best accuracy for all tested traces (values in each column corresponds to a different trace). As expected, the five-tuple key proved to be the most effective for all kinds of changes in the tested traces, independently of the forecasting model used. However, there were cases (namely, the TCP SYN Flood attacks $(7, 8, 9)$), where a coarser flow-key (IP and protocol) allowed the algorithm to achieve the same level of accuracy as with the five-tuple key.

5.3 K-MELEON

The goal of this work was to build a change detection system capable of running within the constraints of the P4 programming model. Towards that goal, the main logic of the k-ary algorithm has been preserved in K-MELEON through careful restructuring of its operations. This restructuring has introduced approximations in some of the operations of the original algorithm, like, for example, in the computation of the forecast sketch, as explained in Section 3.5.1. The main question we ask in the evaluation is therefore if K-MELEON achieves the same level of detection accuracy and resource usage compared to k-ary.

5.3.1 Preliminary note

Our first experiments evaluating K-MELEON required running multiple instances of our solution with different configuration parameters across many different network traces. These experiments consist of *emulations*, so real packets are generated and traverse the network stacks of the source host, bmv2 switch, and destination host. However, running experiments in the behavioral model software switch *bmv2* is very time consuming. The software switch is also not optimized for performance. As a consequence of the slow packet processing and I/O rate, each evaluation run could take hours, and was a bottleneck for evaluation of K-MELEON. For this reason, we decided to develop, for testing purposes, an additional version of k-ary in python, mimicking *exactly* the approximations of K-MELEON. This means that we perform *simulations*, instead of emulations. Importantly, the simulations run orders of magnitude faster, from hours to less than 1 minute in most cases. Several tests were performed across different traces, using the two versions, and the results were identical in all runs.

Hence, all the results reported in the rest of this Section were performed using the K-MELEON implementation in Python. However, our tests with the two versions give us high confidence that the results produced with the P4 version would be identical.

5.3.2 K-MELEON vs k-ary

The validation of the proposed solution requires, as a first step, computing the *relative difference*, defined in [18] as the difference between the total energy (square root of the sum of second moments for each epoch), between K-MELEON and k-ary. In this experiment, several sizes of the sketch data structure are considered, varying the sketch height from 1 to 5. In Figure 5.6, which illustrates the results for this experiment, it can be seen that most of the mass is concentrated around the 0% point on the x axis, with very few points that differ more than 0.2% from the corresponding k-ary ones, for any H. Moreover, the curves corresponding to each distinct value of H show a very similar trend. In conclusion, these results show that the change detection capabilities of K-MELEON are very similar to those of kary, and that the height of sketch (H) has little to no effect on the measured relative difference.

Since some approximation is introduced by K-MELEON in the computation of the forecast sketch in EWMA, we measured any possible impact of these approximations, specifically varying the smoothing constant α values, which affect the level of approximation introduced in the model. To this end, K-MELEON and k-ary were run on eleven traces from the `network_attacks` dataset, fixing the algorithm configuration except for the smoothing constant a values. Finally, each value of the application threshold T_A obtained with the K-MELEON was compared with the value of T_A obtained with k-ary, using a relative difference metric, as done previously.

The results of this experiment are illustrated in Figure 5.7. The figure shows the average empirical CDF for the relative difference obtained for different values of the smoothing constant alpha. These results show that K-MELEON incurred in less than 0.3% relative difference for any tested α value. Interestingly, the relative difference varies slightly according to the value of α used. This variation illustrates the error introduced by the approximations required to run k-ary in the data plane. As multiplication

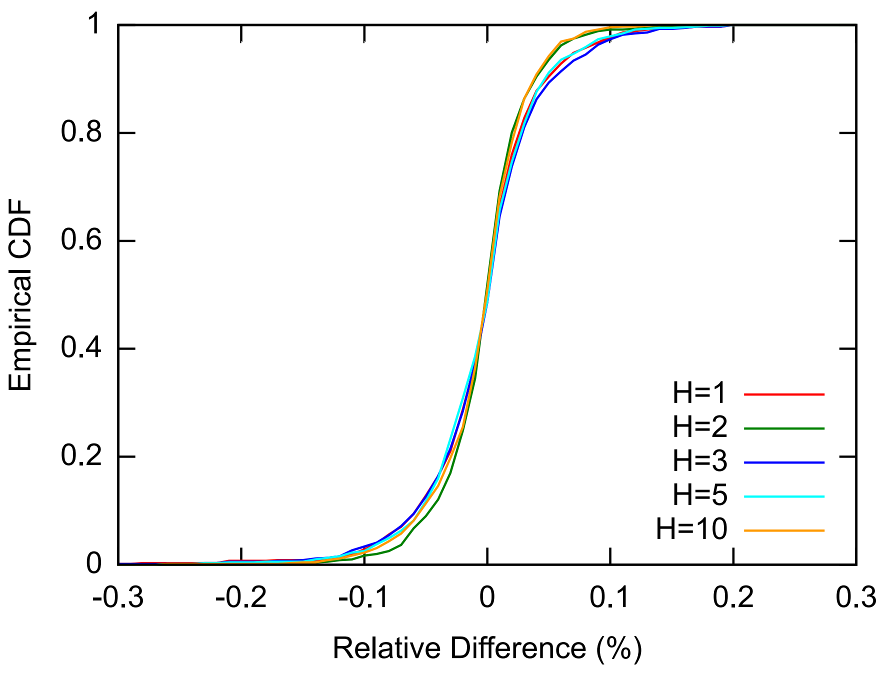


Figure 5.6: Empirical CDF values obtained for the relative difference between k-ary and K-MELEON for different sketch heights (H).

is performed using bit-shifts, the smoothing constant α will introduce an increasing error rate with the number of bit shifts. The smoothing constant $\alpha = 0.5$ only requires one bit shift, whereas $\alpha = 0.75$ and $\alpha = 0.875$ require two and three bit-shifts (and additions), respectively, and the error rate slightly increases accordingly.

The two experiments above show that K-MELEON achieves comparable detection performance as k-ary (requirement **R1**), despite the approximations introduced. Besides, it is worth mentioning that K-MELEON does not achieve this result at the cost of additional memory. It should be clear from our design that the memory consumption for both schemes is very similar. We will return to this point in Section 5.3.5.

5.3.3 Detecting network attacks

To assess the ability of K-MELEON to detect the attacks present in the `network_attacks` traces, we considered the following metrics:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.2}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.3}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.4}$$

The experiments require executing K-MELEON with different configurations across different epoch

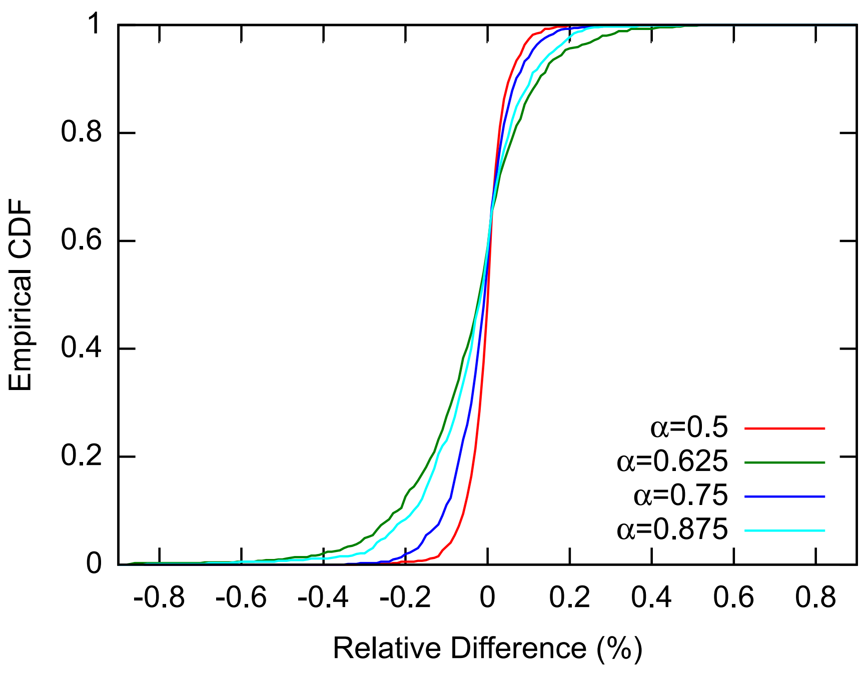


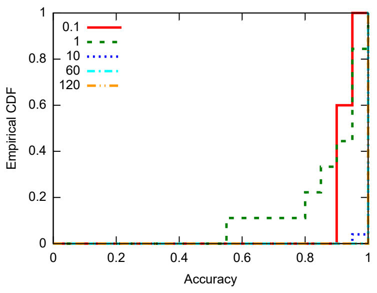
Figure 5.7: Empirical CDF values obtained for the relative difference between k-ary and K-MELEON for different alpha values (α).

sizes, while counting: the number of attacks detected (true positives), the false alarms (false positives), the attacks not detected (false negatives), and the benign traffic undetected (correctly) as an attack (true negatives).

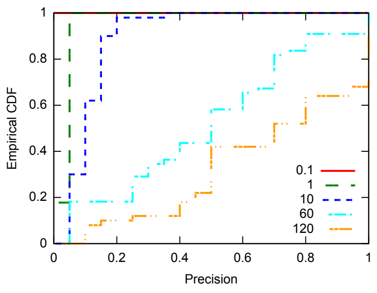
Figure 5.8 reports the detection performance with varying epoch sizes (in seconds) of K-MELEON, using traces including ping of death, SNMP reflection, smurf, TCP syn flood, and UDP flood attacks. All attacks last around 10 seconds – an important detail to interpret the results that follow. The metrics include detection accuracy, precision, and recall. For these experiments, we present average values among several configurations of the sketch and of the forecast model per tested trace.

The main take away is that K-MELEON is an effective change detector. It is able to achieve close to 100% accuracy for epoch sizes greater or equal to 10 seconds, without generating any false negatives (Recall is always 1 – i.e., we are detecting *all* attack-related changes). The reader will note the precision results do not make K-MELEON particularly interesting to be used *alone* as an intrusion detector. Several false positives, or false alarms, are raised, which is undesirable. Indeed, the traces include other traffic changes that K-MELEON correctly detects but that are unrelated to the network attacks. In practice, anomaly-based detectors typically analyse changes in traffic patterns with statistical or ML-based techniques [1].

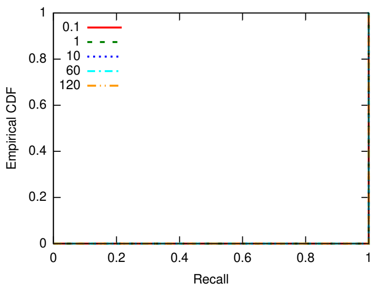
In all experiments our system detects a change either in the same epoch it occurs or in the subsequent. We are thus confident in fulfilling **R3** and **R4** when we perform evaluations with real hardware, which is left as future work.



(a) Accuracy.

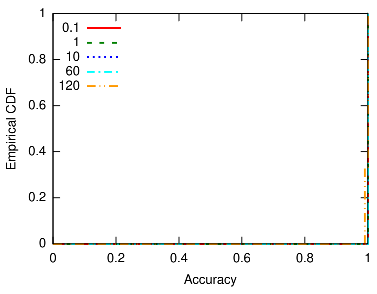


(b) Precision.

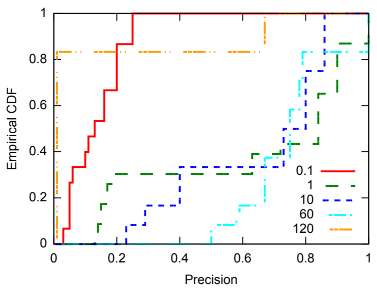


(c) Recall.

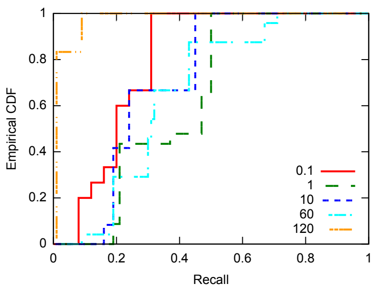
Figure 5.8: Empirical CDF for network attacks.



(a) Accuracy.



(b) Precision.



(c) Recall.

Figure 5.9: Empirical CDF for microbursts.

5.3.4 Detecting microbursts

Towards a more comprehensive evaluation of K-MELEON as a generic change detector, we also tested K-MELEON with regard to the detection of microburst events, another crucial kind of network traffic change. For this purpose, three different traces from the `microbursts` dataset were used. In these experiments, we considered as a microburst any flow, within a 1 second time window, whose count was 20 times higher than the average flow, and accounted for at least 20% of all traffic in that time window. This definition is aligned with the literature on the topic [3]. The evaluation methodology is very similar to the one for the detection of network attacks. Each trace was tested against multiple configurations of the K-MELEON for each epoch size. The metrics considered in this evaluation were the accuracy, precision, and recall.

Figure 5.9(a) shows K-MELEON is very accurate for every epoch size. This is a good result but should be taken with a grain of salt, as will be clarified in the other plots. In this dataset the number of true negatives (normal events not labelled — correctly — as microbursts) is significantly higher than the other metrics, and accuracy is therefore always close to 1.

Figure 5.9(b) illustrates the precision obtained for each epoch size and its results are more interesting. They show that with an epoch size of 1 (the maximum duration of a microburst) K-MELEON detects microbursts with a median precision close to 85%. For epoch size values of 10 and 60 seconds the results slightly decrease but they are anyway more or less acceptable. For the largest and smallest epoch sizes the precision results are very poor, as they are way over or way under the defined duration for a microburst.

H	**1**				**3**			
K	**32**		**64**		**32**		**64**	
B	**4**	**8**	**4**	**8**	**4**	**8**	**4**	**8**
Baseline	1600	3200	1600	3200	1600	3200	1600	3200
K-ary	384 (-76%)	768 (-76%)	768 (-52%)	1536 (-52%)	1152 (-28%)	2304 (-28%)	2304 (+44%)	4608 (+44%)
K-MELEON	388 (-75.75%)	772 (-75.875%)	776 (-51.5%)	1544 (-51.75%)	1164 (-27.25%)	2316 (-27.625%)	2328 (+45.5%)	4632 (+44.75%)

Table 5.2: Memory consumption (in bytes). Baseline values were calculated with 100 flows per epoch.

The results for recall are shown in Figure 5.9(c). Again, using epoch sizes of 120 seconds hinders the detection performance, as the recall is close to 0 for most configurations. The epoch size of 1 second is again the one with best median, of around 0.5. This means that around half of all microburst events are detected. We conjecture that increasing slightly the epoch size, for example to 2, may improve the results as the microbursts events will appear in its entirety in a single epoch, but we leave testing this hypothesis as future work. Similarly to the precision results, the recall decreased as the epoch size increases.

Overall, the evaluation of microburst detection showed that K-MELEON is potentially capable of performing generic change detection, not only of heavy changers. These last experiments further confirmed that one single configuration is not enough to accurately perform detection of all kinds of changes. Different applications on attacks require different configurations. Again, as the events are detected right after the epoch where they happen, K-MELEON can potentially detect several microbursts fast, immediately as they occur.

5.3.5 System performance and resource usage.

We implemented our program to run on our Intel Tofino. The code was compiled successfully with the Tofino SDE. This demonstrates it achieves line rate performance (6.4 Tbps, **R5**).

The memory consumption for K-MELEON and the original k-ary is similar, as they both use 3 main sketches of equal size. A difference is that K-MELEON requires additional space to store a second error sketch (to be used when changing epochs), while k-ary needs to keep the observed sketch in memory, something not required in our streaming-based design. But they are of the same size, cancelling out. However, our system further requires a control sketch that uses 1 bit per bucket. As a result, the relative increase in total memory of K-MELEON over k-ary is equal to $1 - \frac{1}{3\times B}$, where B is the bucket size. Table 5.2 presents the memory consumption for several sketch values, and compares them against the baseline values using per-flow analysis.

There are two main takeaways. First, the memory consumption of K-MELEON is roughly the same as that of k-ary, for these realistic sketch values **(R2)**. Second, the sketch versions represent important memory savings when compared to the baseline, up to a point. For the larger sketch sizes (H=3 and K=64), the sketch uses more memory than the baseline alternative of storing all flows. This is a result of the small number of active flows in the trace used as reference in this experiment (only roughly 100 flows per epoch, on average). This is a result of the small scale of the tesbed used to generate this network_attacks testbed. In any realistic mid- to large-scale network the number of active flows is expected to be in the order of the several thousands [15], so the memory efficiency of the sketch-based versions would become evident. This can be demonstrated by the results with the smaller sketches ($H = 1$ and $K = 32$). In this case the memory savings are considerable, and this is achieved with an insignificant relative difference error of less than 0.2% compared to the baseline.

5.4 Summary

This chapter presented the evaluation of k-ary and K-MELEON, two solutions for the traffic change detection problem. First, we described the testing environment and the datasets used for evaluation in Section 5.1. Afterwards, we validated the k-ary sketch implementation by comparing it against the per-flow analysis baseline. This section also included results on our exploration of the configuration space for k-ary. In section 5.3, we report the evaluation of K-MELEON against the k-ary algorithm on its ability to detect changes, with small resource consumption and with the guarantee to run at line rate in a programmable switch.

Chapter 6

Conclusions

This thesis presented K-MELEON, a system that detects significant traffic changes for any type of flow. K-MELEON revisits traditional change detection algorithms and proposes a new streaming-based design that fits the computation model and constraints of programmable switches. By performing the required computations in-network, as the packets traverse the switch, K-MELEON can potentially respond to sub-second changes and scale to Tbps speeds.

We studied the state of the art in the fields of programmable networks and change detection techniques and selected the existing solution which best fit our purpose, the k-ary sketching algorithm. More specifically, we leverage the k-ary sketch to build a solution capable of performing change detection almost entirely in the data plane of programmable switches using the P4 language.

One of the main challenges consisted in fitting the k-ary operations into such a constrained programming model. For instance, most operations performed by the forecasting model of the k-ary sketching algorithm required floating-point arithmetic, which was not feasible to implement in P4. Instead, these operations had to be approximated using bit-shifts.

Another main obstacle was the batch-based nature of the original k-ary. More specifically, the k-ary algorithm performs all forecasting and change detection operations at the end of one epoch, as a batch. This solution does not fit the pipelined model of hardware packet processors, and thus required careful modifications to the k-ary functions to be implemented in a stream-based fashion on each packet, within the constraints of the target P4-capable platform.

6.1 Achievements

In summary, the contributions of this dissertation are:

- The design of K-MELEON, an online change detection system that speeds and scales up detection by leveraging programmable switches.

- The implementation of a prototype in P4 [22, 23], available open-source.

- An evaluation that demonstrates K-MELEON achieves the same level of accuracy as the offline k-ary solution, and detects changes from *any* type of flow (not only heavy-hitter traffic as existing work [16, 17]).

6.2 Discussion and future work

Our initial exploration of in-network change detection with K-MELEON shows its promise as an online change detector. To foster discussion on an area still on its early days of leveraging network programmability, in this section we discuss some limitations of K-MELEON and avenues we think are worth exploring in the general context of change detection.

Detection entirely in the data plane. The current design of K-MELEON is still bottlenecked at the switch CPU. At the end of each epoch, the control plane must read the entire error sketch and the keys stored in the current epoch to compute and compare the alarm threshold T_A with the error estimates E_k. This limits scalability and introduces latency that may be crucial for timely detection of certain events. We are currently offloading this task to the control plane due to the relative complexity of computing $F2$ and E_k. While challenging, the complexity may not be insurmountable, however. We are exploring target-specific parallelizations to avoid this offload, both "horizontal" and "vertical". The first is to perform various sub-parts of a complex operation across multiple stages (similar to pipelining), transporting partial results using packet metadata between stages. The second is to perform multiple operations in parallel in the same stage, aggregating the results in subsequent stages. The challenge is being able to perform the required calculations within the limited number of stages of the target platform.

Configuration space. Several parameters contribute to the detection accuracy of K-MELEON. Namely, the number (H) and width (K) of the hash tables (and hence the size $H \times K$ of the sketch data structure), the application-specific threshold T, the epoch size, the choice of the forecast model, and the model parameters. We have not explored this configuration space exhaustively. Rather, we have so far focused on finding configurations which on average worked well across the tested traces. Our experiments confirmed that the tuning of certain parameters may considerably help improve detection accuracy. For instance, we clearly observed the influence of changing the smoothing constant α of the *Exponentially Weighted Moving Average* (EWMA) forecast model to achieve better detection results. Because the search space is large, we consider as an interesting avenue of research the development of automatic profilers (e.g., using Bayesian optimization techniques) to find the most adequate parameters for a given application.

Target-specific optimizations. The design of K-MELEON considers a high-speed switching chip like Tofino [21] as target platform. The prototype was tested on the *bmv2* software switch but it also compiled on our Intel Tofino switch. The P4 program mostly deals with partitioning K-MELEON data structures and sketch operations accordingly along the switch's pipeline, since the number and type of those operations already generally abide by the target's specific constraints. Besides, we believe that architecture specific support for mathematical operations, provided through *extern* objects, could help us further optimize the operations of K-MELEON.

Bibliography

[1] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018*.

[2] P. Wendell and M. J. Freedman. Going viral: Flash crowds in an open cdn. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, 2011.

[3] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, 2017.

[4] V. Paxson. Bro: A system for detecting network intruders in real-time. *Comput. Netw.*, Dec. 1999.

[5] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, 1999.

[6] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita. Network anomaly detection: Methods, systems and tools. *IEEE Communications Surveys Tutorials*, 16(1), 2014.

[7] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[8] G. Cormode and S. Muthukrishnan. What's new: Finding significant differences in network data streams. *IEEE/ACM Transactions on Networking*, 13(6):1219–1232, 2005.

[9] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 29–42, 2013.

[10] R. Harrison, Q. Cai, A. Gupta, and J. Rexford. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[11] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126, 2017.

[12] Q. Huang, P. P. Lee, and Y. Bao. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590, 2018.

[13] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. 2019.

[14] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[15] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, pages 164–176, 2017.

[16] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[17] L. Tang, Q. Huang, and P. P. Lee. Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2026–2034. IEEE, 2019.

[18] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247, 2003.

[19] R. Miao, R. Potharaju, M. Yu, and N. Jain. The dark menace: Characterizing network-based attacks in the cloud. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, 2015.

[20] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[21] Intel programmable ethernet switch products. URL `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html`.

[22] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[23] The p4 language consortium. URL `https://p4.org/`.

[24] K-meleon github repository. URL `https://github.com/netx-ulx/dataplaneChangeDetection`.

[25] N. Feamster, J. Rexford, and E. Zegura. The road to sdn: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, 44(2):87–98, 2014.

[26] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[28] O. S. S. Version. 1.5. 0 (protocol version 0x06). *Open Networking Foundation*, 2015.

[29] J. Proença, T. Cruz, E. Monteiro, and P. Simões. How to use software-defined networking to improve security-a survey. In *European Conference on Cyber Warfare and Security*, page 220. Academic Conferences International Limited, 2015.

[30] A. Lara, A. Kolasani, and B. Ramamurthy. Network innovation using openflow: A survey. *IEEE communications surveys & tutorials*, 16(1):493–512, 2013.

[31] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[32] H.-P. Development Company. Hp switch software openflow v1.3 administrator guide k/ka/wb15.18. Technical report, Hewlett-Packard Development Company, L.P., April, 2016.

[33] R. Bifulco and G. Rétvári. A survey on the programmable data plane: Abstractions, architectures, and open problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–7. IEEE, 2018.

[34] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 693–703. Springer, 2002.

[35] P4-16 language specification. URL `https://opennetworking.org/wp-content/uploads/2020/10/P416-Language-Specification.html`.

[36] J. Kučera, D. A. Popescu, H. Wang, A. Moore, J. Kořenek, and G. Antichi. Enabling event-triggered data plane monitoring. In *Proceedings of the Symposium on SDN Research*, pages 14–26, 2020.

[37] C. I. NetFlow. Introduction to cisco ios netflow a technical overview. *White Paper, Last updated: February*, 2006.

[38] M. Wang, B. Li, and Z. Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 628–635. IEEE, 2004.

[39] B. Claise and S. Bryant. Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information. Technical report, RFC 5101, January, 2008.

[40] E. Teramoto, M. Baba, H. Mori, Y. Asano, and H. Morita. Netstream: traffic simulator for evaluating traffic information systems. In *Proceedings of Conference on Intelligent Transportation Systems*, pages 484–489. IEEE, 1997.

[41] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 311–324, 2016.

[42] P. Laffranchini, L. Rodrigues, M. Canini, and B. Krishnamurthy. Measurements as first-class artifacts. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 415–423. IEEE, 2019.

[43] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

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

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

[46] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 153–166, 2003.

[47] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.

[48] V. Braverman and R. Ostrovsky. Zero-one frequency laws. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 281–290, 2010.

[49] V. Braverman and R. Ostrovsky. Generalizing the layering method of indyk and woodruff: Recursive sketches for frequency-based vectors on streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 58–70. Springer, 2013.

[50] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[51] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31 (23-24):2435–2463, 1999.

[52] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[53] C. S. Hood and C. Ji. Proactive network-fault detection [telecommunications]. *IEEE Transactions on reliability*, 46(3):333–341, 1997.

[54] I. Katzela and M. Schwartz. Schemes for fault identification in communication networks. *IEEE/ACM Transactions on networking*, 3(6):753–764, 1995.

[55] A. Ward, P. Glynn, and K. Richardson. Internet service performance failure detection. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):38–43, 1998.

[56] J. Tölle, O. Niggemann, et al. Supporting intrusion detection by graph clustering and graph drawing. In *Proceedings of Third International Workshop on Recent Advances in Intrusion Detection RAID 2000*, 2000.

[57] N. Ye et al. A markov chain model of temporal behavior for anomaly detection. In *Proceedings of the 2000 IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop*, volume 166, page 169. West Point, NY, 2000.

[58] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: Methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 234–247, 2003.

[59] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 207–212, 2004.

[60] L. Tang, Q. Huang, and P. P. Lee. Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 2026–2034. IEEE, 2019.

[61] R. S. Boyer and J. S. Moore. Mjrty—a fast majority vote algorithm. In *Automated Reasoning*, pages 105–117. Springer, 1991.

[62] Barefoot tofino. `https://www.barefootnetworks.com/products/brief-tofino/`, 2020. Accessed: 2021-06-28.

[63] C. Kim. Programming the network data plane: What, how, and why? URL `https://conferences.sigcomm.org/events/apnet2017/slides/chang.pdf`.

[64] S. Muthukrishnan. Data streams: Algorithms and applications. In *Manuscript based on invited talk from 14th SODA*, 2003.

[65] S. Ibanez, G. Antichi, G. Brebner, and N. McKeown. Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 133–140, 2019.

[66] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 67–82, 2017.

[67] Multiprocessing python package. `https://docs.python.org/3/library/multiprocessing.html`. Accessed: 2020-05-29.

[68] The behavioral model version 2 (bmv2), the reference p4 software switch. URL `https://github.com/p4lang/behavioral-model`.

[69] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSp*, pages 108–116, 2018.

[70] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[71] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich. Catching the microburst culprits with snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, pages 22–28, 2018.

[72] p4c, the reference compiler for the p4 programming language. URL `https://github.com/p4lang/p4c`.

[73] Scapy, the packet library for packet manipulation. URL `https://scapy.net/`.