TÉCNICO LISBOA

Identification of Skype Covert Channels using Sketches in SDNs

André Filipe Antunes Madeira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Rodrigues
Prof. Nuno Santos

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. Luís Rodrigues
Member of the Committee: Prof. Fernando Manuel Valente Ramos

December 2019

Acknowledgments

I would like to thank my parents for the support and courage they have provided me over the years and made this project possible. I would like to thank my friends who are always by my side and gave me the courage to do my best. I would like to thank my teachers and coordinators Luís Rodrigues and Nuno Santos for guiding me during the duration of this project. I would like to thank the PhD student, Diogo Barradas, for the time and help he provided to clarify doubts and discuss new ideas. I would also like to thank Professor Fernando Ramos for his comments on a previous version of this work.

Thank you all.

Abstract

The characterization of network flows is relevant for multiple applications, in particular for security applications, such as the detection of covert channels in real time. Typically, the characterization of network flows is performed by capturing information of all the packets of the relevant flows and analyzing their characteristics, for example, performing machine learning classification based on the distribution of their sizes. However, this solution consumes many resources, affecting network performance, and typically can only perform the classification at a later time. In this work, we evaluate the possibility of exploring the recent advances in SDN networks, programmable switches, the P4 programming language and probabilistic data structures (also called sketches) to characterize the flows in the switch itself thus reducing the amount of network data that need to be stored and analyzed to identify covert channels. We present a software architecture for programmable switches that allows us to characterize flows using two layers of filtering, each using a sketch, to first reduce the high amount of flows being processed, limiting the number of misclassified flows that do not contain covert channels. Our solution allows us to monitor 5K flows while keeping an accuracy of 0.95 in the detection of covert flows, representing an increase in analysis capacity of 20 times for the same amount of memory on the switch in the absence of sketches.

Keywords

Covert Channels; Software Defined Networks; Programmable Switches; Sketches.

Resumo

A caracterização de fluxos de rede é relevante para múltiplas aplicações, em particular para aplicações de segurança, tal como a deteção de canais encobertos em tempo-real. Tipicamente, a caracterização de fluxos de rede é realizada por capturar informação de todos os pacotes dos fluxos relevantes e analisando as suas características, por exemplo, realizando classificação com aprendizagem de máquina baseada na distribuição dos seus tamanhos. No entanto, esta solução consome muitos recursos, afetando o desempenho da rede, e tipicamente apenas consegue realizar a classificação posteriormente. Neste trabalho aferimos a possibilidade de explorar os avanços recentes nas redes SDN, nos comutadores programáveis, na linguagem de programação P4 e nas estruturas de dados probabilísticas (também designadas por *esboços*, do Inglês, *sketches*) para caracterizar os fluxos no próprio comutador reduzindo assim a quantidade de dados de rede que têm de ser armazenados e analisados para identificar canais encobertos. Apresentamos uma arquitetura de software para comutadores programáveis que permite caracterizar os fluxos utilizando duas camadas de filtragem, cada uma recorrendo a um esboço, para primeiro reduzir a quantidade elevada de fluxos a processar, limitando o número de fluxos que não contêm canais encobertos classificados incorretamente. A nossa solução permite monitorizar 5K fluxos mantendo uma precisão de 0,95 na deteção de fluxos encobertos, representando uma capacidade de análise 20 vezes maior para a mesma quantidade de memória no comutador na ausência de esboços.

Palavras Chave

Canais Encobertos; Redes Definidas por Software; Comutadores Programáveis; Esboços.

Contents

List of Figures

List of Tables

Listings

Acronyms

SDN Software-Defined Networks

P4 Programming Protocol-Independent Packet Processors

CM Sketch Count-Min Sketch

No-CM No Count-Min Sketch

CM-S Count-Min Sketch Single

CM-PF Count-Min Sketch Per Flow

CM-PB Count-Min Sketch Per Bucket

PISA Protocol Independent Switch Architecture

ROC AUC Receiver Operating Characteristics Area Under Curve

1

Introduction

Contents

In this work we assess the feasibility of exploring recent advances in computer networks to detect covert channels with high resource efficiency and low latency. In particular, we intend to take advantage of programmable switches, which can perform simple operations at line speed, to extract an approximate distribution of the packet sizes of the various flows in the switch itself without needing to duplicate packets. Since the amount of memory available in switches is relatively limited, we use probabilistic data structures (also called sketches) [3, 4]. These structures take up the available memory in a very efficient way, but, on the other hand, they do not allow for characterization of packet size distributions in a completely precise way. One of the challenges of this work is to understand how it is possible to increase the number of multimedia connections, Skype in our case, monitored simultaneously, with the limited memory that exists in the switch, obtaining satisfactory accuracy in the classification of the flows.

1.1 Motivation

The characterization of packet flows is required for various security applications, such as detection of covert channels [5], or creation of access profiles [6]. In this work, we address the problem of detecting, in real time, covert channels in multimedia flows, namely in Skype calls. We consider the scenario where it is possible to have access to packets exchanged for Skype connections and we intend to identify which of these are legitimate calls and which are calls that carry a covert channel. Covert channel information is typically encoded in audio and/or video exchanged by Skype through censorship-resistant communication tools such as Facet [7] or DeltaShaper [8]. As Skype encrypts multimedia information, the presence of the covert channel can only be detected by analyzing some features of the packets, such as their size or frequency. In this thesis we focus on the detection of covert channels using machine learning classification algorithms based on the packet size distribution of Skype calls. Previous work [5] showed that most techniques used to create covert channels on Skype are vulnerable to this type of analysis. However, that same work did not showcase the ability to employ such techniques towards real-time classification.

To analyze the packet size distributions of a given Skype call, copies of the corresponding flow packets can be gathered on a dedicated server that analyzes their characteristics. Since the number of Skype calls can be quite high, this process can entail a high cost, namely the memory of the switch needed to collect the necessary metrics, limiting the amount of concurrent flows that can be monitored. Tools like WireShark [9] and sometimes even on the SNMP protocol [10] use this deferred mode. Other technologies like NetFlow [11] are usually hardware-dependent and employ sampling, which may result in the complete lack of classification of certain flows. Additionally, tools such as ping and traceroute [12] are limited in what measurements they can perform making them poor fits for our purpose. Due to modern advances in networking, these have been complemented or even replaced by newer tools and

technologies, such as software-defined networking (SDN) [1], which allows for complete interoperability within the network, and programmable switches [13], that allow for a greater range of measurements to be performed within the devices themselves. The latter are especially needed because, given the high speed of current networks and the amount of information that may need to be collected and processed. In particular, the packet size distributions we require can be too large for the often-limited amount of memory possessed by switches.

For our purposes, we aim at combining techniques that have been made viable by recent advances in networking technologies. Technologies such as Software-Defined Networks (SDN) that provide the tools to change the behavior of switches at runtime, and therefore to dynamically select the flows for which statistical data is collected. Programmable switches and the Programming Protocol-Independent Packet Processors (P4) language [14] allow us to implement programs capable of performing simple computations at the switches, which opens the door for extracting fine-grained information with almost no added latency. Finally, space saving data structures such as bloom filters [15] and counting sketches [3, 4] can be implemented within programmable switches at the packet processing operation to provide ways for the switch to store larger amounts of information than would be possible otherwise, as estimates of per-flow traffic characteristics. And space saving data structures which allow for the careful management between the space consumed for storing the estimates and the loss of accuracy between the estimates and the real values.

1.2 Contributions

This thesis proposes and compares different strategies for collecting the quantized packet length distribution of network flows on a P4 programmable network switch. This is then used to identify covert channels in Skype multimedia connections at a seperate machine. The main contribution of the thesis is the following:

- Two novel approaches to collect packet length distributions that use the Count-Min sketch [4] as a building block. These approaches focus on obtaining precise enough approximations to maintain a high level of accuracy when performing covert channel identification. One employs the use of a single sketch to gather the distributions. The other combines the first approach with an additional filtering operation (also employing a single sketch and classification operation) before obtaining the desired results.

1.3 Results

This thesis produced a detailed experimental comparison of different systems and measurement techniques.

This experimental evaluation shows that our algorithms can significantly increase the number of Skype flows that can be monitored. In some cases, our solution allows for the analysis of 20 times more flows for the same amount of available memory.

1.4 Context

This work was performed in the context of a broader project, being carried out in the Distributed Systems Group of INESC-ID, which aims at understanding if it is possible to implement covert channels that are indistinguishable from legitimate traffic and how the combination of several of the previously mentioned technologies allow for the efficient classification of certain types traffic when in the presence of other types as well. In particular, this work complements the work by Diogo Barradas, who has developed a tool to encode cover channels, named DeltaShaper [8], and has conducted a study on the use of machine learning to detect convert channels [5].

This work was supported by the Fundação para a Ciência e a Tecnologia (FCT), through projects with ref. UID/CEC/50021/2019 and COSMOS (financed by the OE with ref. PTDC/EEI-COM/29271/2017 and by the Programa Operacional Regional de Lisboa in its FEDER component with ref. Lisboa-01-0145-FEDER-029271).

1.5 Organization of the Document

This thesis is is organized as follows. In Chapter 2 we present all the background related with our work and similar applications and systems. In Chapter 3 we describe the proposed architecture to be implemented. Chapter 4 we showcase the results obtained from our experimental evaluation. Chapter 5 concludes this thesis by outlining the discoveries and proposes some directions for future work.

2

Background and Related Work

Contents

This chapter starts by introducing the background concepts discussed in this work and some systems that could enable us to perform the necessary measurements to identify Skype covert channels. In Section 2.1, we discuss methods to obtain the necessary measurements and their limitations. In Section 2.2, we introduce the concept of software-defined networks. In Section 2.3, we discuss programmable switches and their uses. In Section 2.4, we present the Programming Protocol-Independent Packet Processors programming language or P4. In Section 2.5, we discuss the identification of Skype covert channels in multimedia calls. In Section 2.6, we discuss how space-saving algorithms function in order to perform memory-efficient measurements. Finally, in section 2.7, we compare different systems that share similar goals to ours.

2.1 Network Traffic Measurement

Network traffic measurement is the process of extracting quantitative information regarding the operation of a network, including data regarding the operation of the devices and links, but also information about the traffic that the network carries. Typically, network measurement is performed with the help of the network devices themselves, that keep some statistics regarding their own operation. The values captured by the network devices can be read remotely, using protocols such as SNMP (Simple Network Management Protocol [10]) or HTTP, such that they can be collected and analyzed in a central location. Application-level tools that run on the network endpoints, such as *ping* and *traceroute* [12], that exploit native features of the IP protocol can also be used to extract useful information, such as reachability and packet routes.

Unfortunately, the type, amount, and level of detail of information collected by tools such as NetFlow [11], which has to resort to sampling to manage the passing traffic, can be limited and not enough to satisfy the requirements of all modern network management tasks such as: detecting which flows consume more resources in a network (also known as “heavy hitters” [16]), detecting denial of service attacks [17], intrusion detection [18], among others. For a comprehensive survey of measurements that are relevant in today’s network, the reader is referred to the works of Callado et al. [19] and Mohan et al. [20].

One way of obtaining detailed information about network flows, a sequence of packets from a source to a destination, that pass through the network consists in collecting the packets and logging them at some central location for deferred processing. Packets can be collected using specialized hardware and middleboxes to collect traffic traces, sniffers like WireShark [9] (that store representations of packets as they cross the network) or by having switches duplicate the packets and send copies to a central location [21]. Unfortunately, this approach is not without limitations. Current small and medium sized networks can route packets at speeds in the range of billions of packets per second. Logging packets at

this speed may be unfeasible and consume too many resources. This can be mitigated using strategies such as sampling [22], where information regarding only a random subset of the packets collected, with some loss of accuracy. However, sampling is also difficult to apply when one seeks to collect information regarding short flows, as sampling may collect few, or even no samples, of the target flows or yield sub-optimal results when performing certain traffic analysis tasks.

2.1.1 Network-Level Measurements

At the network-level, network performance metrics like throughput, packet loss and packet delay are commonplace for works in this area. This is mostly due to this area of network traffic measurement being more focused on overall network debugging and optimization. While our own measurements are not of this type it may still prove useful to analyze how systems working at this level perform similar tasks to our own Skype covert channel identification.

2.1.2 Flow-Level Measurements

At the flow-level, we can see some more diversity in the metrics taken. Some works, such as [23, 24], focused on the UDP/TCP ratios and TCP/IP options in order to provide a better understanding of usage patterns regarding these protocols. Murray et al. [23] sought out to investigate which was the most used transport layer network protocol (TCP or UDP) in terms of flows, packets and bytes. This allowed for the better understanding of network usage patterns employed by streaming applications. Zhang et al. [24] study intended to promote further development on the IP and TCP protocols by reflecting on the most common characteristics of Internet traffic. They also point out misbehaviors and errors when employing these protocols.

A study by John et al. [25] aims at achieving accurate Internet statistics by discovering the mean and distribution of packet sizes, as well as a range of network and transport layer statistics. This is similar to our choice of metrics.

If we were to try and collect packet length distributions with only these techniques, the optimal way to do so would be through the addition of a middlebox to the network between the switches/routers and the users. Despite the heavy resource consumption, it is our best alternative to collect detailed metrics, since the use of sampling could lead to a misrepresentation of the traffic. However, the middlebox would add latency to the network which we argue is unneeded due to the application of the technology analyzed next.

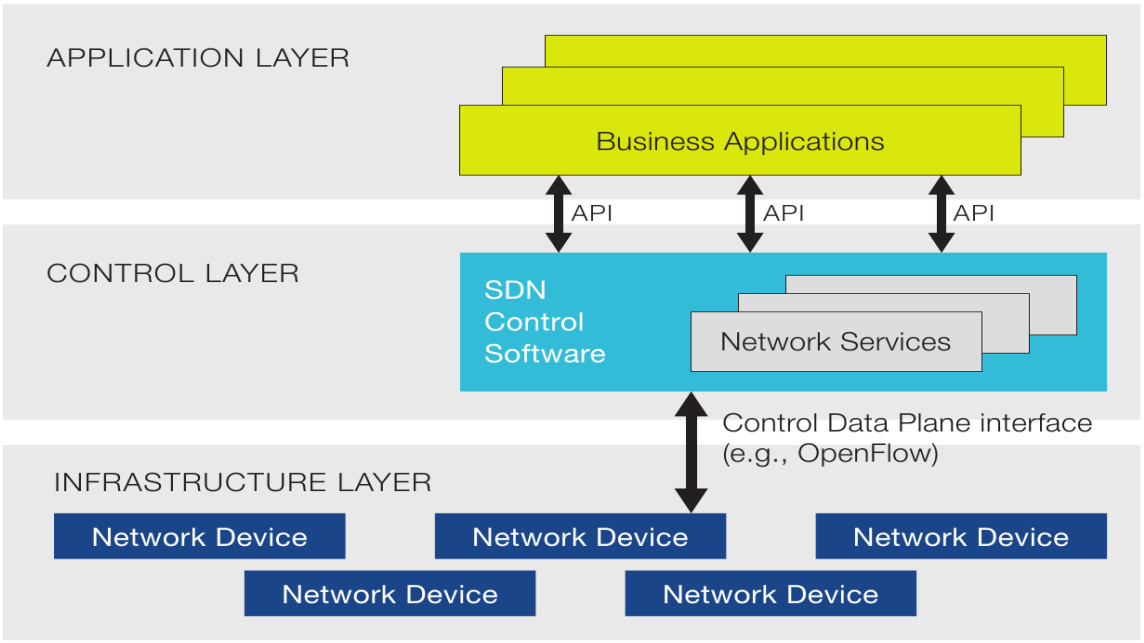


Figure 2.1: SDN networking stack. From the SDN white paper [1]

2.2 SDN

Software-defined networking (SDN) [1] is an architecture that allows for the underlying infrastructure of networks to be abstracted for applications and network services. It decouples the network control from the data forwarding functions of network devices. It enables the network control, which fill the flow tables at the switch (that decide if and where packets are forwarded), to become directly programmable, facilitates network management, and enables efficient network configuration.

The SDN architecture is known for separating what is called the data plane and the control plane. The control plane usually refers to the part of the networking stack that makes decisions about where traffic is sent and the data plane is the part through which user packets are transmitted/forwarded. The control plane is managed by what is known as a SDN controller. An SDN controller manages several devices of the SDN network, interacting with the devices to control their behavior. Overall, the SDN architecture, often represented as a stack as seen in Figure 2.1, contains a total of three layers: the application, control, and infrastructure layers. The infrastructure layer consists of the networking devices, such as switches and routers. The control layer holds the SDN controllers, which usually operate on separate machines and servers. The application layer holds the applications, that use the controller to request network resources and to define the network behavior. An SDN controller communicates with the network devices through an API called the *southbound API* (the name derives from the location of the API in the networking stack) and with the services and applications through an API called the

northbound API. The application can use this interface to modify the configuration and operation of the network. This allows SDNs to be dynamic, manageable, cost-effective, and adaptable.

SDNs introduce several benefits to networking including lower operating costs, easy automation of network devices, improved security across the network, and improved quality of service for multimedia transmissions. In addition to these we mainly focus on the next two. First, SDNs provide a centralized view of the entire network, making it easier to centralize measurement gathering from multiple devices. Second, they allow for the repurposing of existing hardware with the use of the SDN controller, also allowing for less expensive hardware to be deployed to greater effect.

SDNs typically operate as follows. First, the controller may pre-configure the switch with a set of permanent rules. Otherwise, if new rules are to be added, then when a switch receives a packet that does not match any already existing rule, it may contact the controller in order to handle the new packet. The data plane will clone the packet towards the controller and drop the original. The controller will evaluate the packet and add a new rule to the switch matching prominent header fields of the packet, including the source and destination MAC address, source and destination IP, source and destination port, protocol, to name a few. Afterwards, the packet will be resent towards the switch where it will be matched to the rule just added and forwarded toward the proper destination. It should be noted that in the case where the controller receives multiple packets, all of the same flow, then only one instance of the rule should be added and all of the resent towards the switch. In this case, rules will usually have a timeout associated with them to flush them after a set period of inactivity. Figure 2.2 illustrates this process.

2.2.1 OpenFlow

OpenFlow defines a communication protocol in SDN environments, enabling the SDN Controller to directly interact with the data plane of network devices such as switches and routers. It can query any of the switches it is connected to for a wide range of information it may require, including information about its ports, interfaces, flow structures, statistics and forwarding rules. It benefits from easy programmability, allowing for the addition of new and innovative features at an accelerated rate, optimized network performance, easy implementation of network-wide forwarding policies and vendor independence.

OpenFlow explicitly specifies protocol headers on which it operates. As such, despite originally only supporting four protocols: Ethernet, VLANs, IPv4 and ACLs, nowadays about fifty protocols are supported. And despite growing quite large already, the community worries if OpenFlow will be able to keep supporting a growing number of protocols given that it would increase the complexity of the specification while still not providing the flexibility to add new headers. Additionally, while it is possible to extend OpenFlow with new features the switches themselves are still limited in what they can accomplish. This is often due to them possessing fixed-function chips designed by the original vendors.

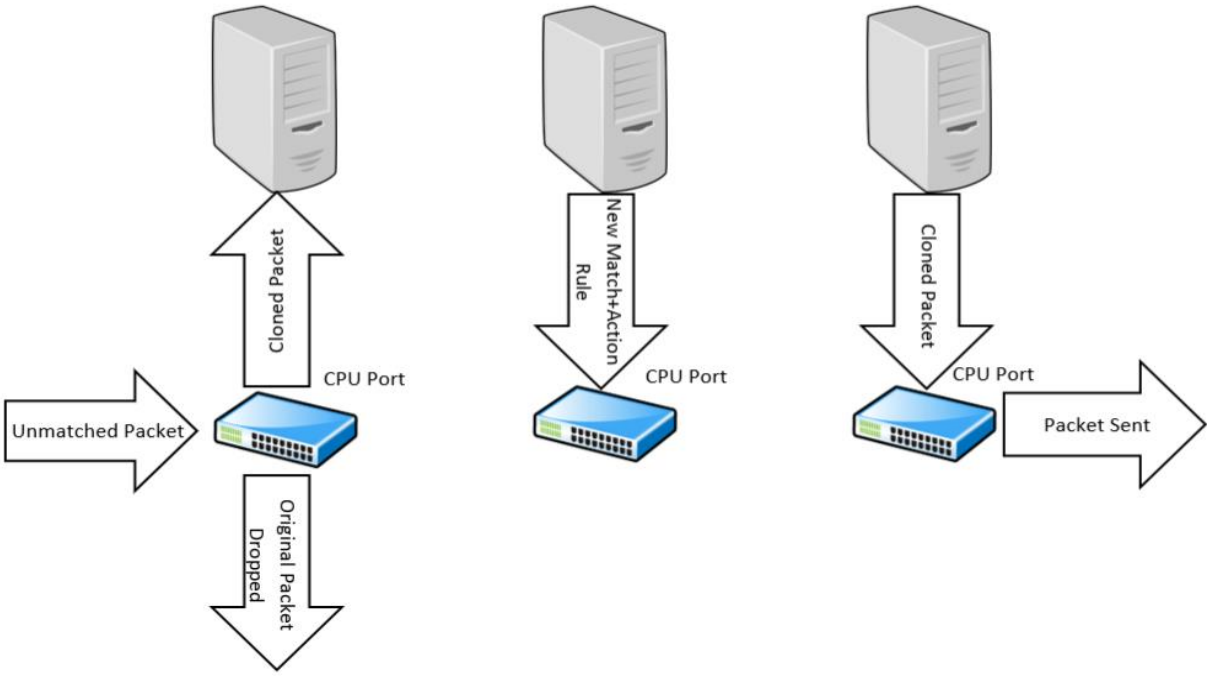


Figure 2.2: Process for the addition of new rules.

Unlike what we saw in the previous section, if we tried to collect packet length distributions while making use of SDN technology and OpenFlow, we could immediately make some improvements. First, we could dispose of the middlebox entirely since SDN technology can be easily integrated into preexisting networks. While the middlebox is able to collect the measurements, it might have to handle the traffic of multiple switches by itself. If the middlebox lacks the processing power to handle influx of packets this could quickly lead to packets being delayed and even getting timed out. Instead, by performing the measurements at each individual switch the workload can be split up throughout all the switches. By utilizing switches that support our measurements and extending OpenFlow to do the same, we could simply have the controller use OpenFlow to directly query the switch for the metrics. Additionally, as controllers frequently provide centralized control over the multiple switches, it is possible to perform this on multiple devices with a single controller. However, resource consumption is still a problem, specifically memory usage can quickly escalate to take up most of the available space in the switch. It is also limited when trying to support our measurement operations, since both the switch and OpenFlow might not support network protocols the user requires, whether it be custom-made or otherwise. Additionally, their use might not allow for the implementation of the space-saving data structures we present in future sections due to lack of programmability. Not only would the switch be unable to process these protocols, but OpenFlow would be unable to install any new rules concerning them.

2.3 Programmable Switches

Programmable switches [13] are switches whose behavior can be re-programmed by network engineers, in opposition to traditional switches, that have a set of pre-defined behaviours built-in. When using fixed function switches, the network engineer is limited by the constraints imposed by the fixed behavior of equipment and by the limitations of the (fixed) protocol used to communicate with the switches (for instance, OpenFlow). Early programmable switches could only process packets at a rate of 1/10th or 1/100th compared to fixed-function ones. Today however, there are programmable switches on the market that process packets just as fast as the fastest fixed-function switches. This type of programmable chip is typically called a Protocol Independent Switch Architecture (PISA) chip.

With programmable PISA chips it became possible to define how packets are processed in switches. The packet processing capabilities programmable switches enable in the data plane allow for the integration of techniques that allow for the efficient storing of statistics, using sketches for example. With the intent of creating a common programming language for both ASIC and PISA chips, P4 was created. P4 not only provides target independence, but also provides a way for switches to be reconfigured after deployment, all the while not being tied to any number of protocols like OpenFlow.

Once more, programmable switches can aid in performing our measurements. First, due to their packet processing programmability, the switch would be able to support any network protocol the user would require. This would allow us to lower the memory used at the switch. By directly altering the packet processing operation we can circumvent this issue, either by using space efficient storage techniques, which we will discuss in Section 2.6, to store the metrics until they are requested by the controller or by setting up a flush time loop for the saved metrics with the controller, to periodically dispose outdated metrics.

2.4 P4

The Programming Protocol-Independent Packet Processors language (P4) [14] is a high-level language for protocol-independent packet processing and data plane programming. P4 relies on the concept of match+action pipelines. Forwarding network packets can be broken down into a series of table lookups, until a suitable “match” is found, and modifications to protocol headers, which are known as “actions”.

P4 has three goals: *reconfigurability*, programmers should be able to change the way switches process packets once they are deployed, *protocol independence*, it should not be tied to any predefined set of network protocols dependent on the switch hardware but any protocol of interest can be easily supported by the data plane programming, and *target independence*, programmers should be able to describe packet processing functionality independently of the specifics of the underlying hardware.

P4 addresses only the data plane of a packet forwarding device. As such, it does not specify the

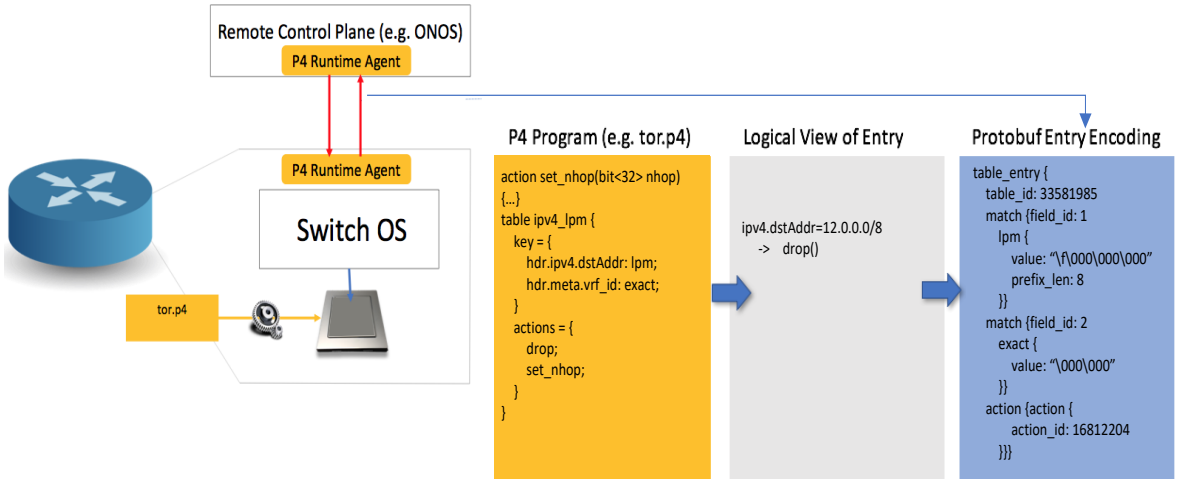


Figure 2.3: Communication between a controller running a gRPC client and the data plane running the gRPC server. A table and a logical representation of an entry for the tor.p4 program. P4Runtime encoding of a table entry for the tor.p4 program.[1]

control plane nor any exact protocol for communicating state between the control and data planes. Instead, the P4Runtime API which we will discuss next, can be used.

Since P4 dealt exclusively with the programmable data plane, there was no one way for the control plane to communicate with it. So, p4.org launched the API Working Group, to create silicon-independent APIs for controlling the forwarding plane of switches. P4Runtime's architecture makes it independent of protocols as well as underlying forwarding switch. It lets users control any forwarding plane, regardless of whether it is built from a fixed-function or programmable switch ASIC. Some of its main functionalities are managing match+action tables, which allows for the modification of entries in the match+action tables, and updating the forwarding plane logic. Due to their protocol independence, P4 and P4Runtime can scale to new protocols as it was designed keeping in mind fixed-function switches.

P4Runtime messages are sent between a gRPC client, in the controller, and a server, in the switch. gRPC is a remote call system procedure that allows for subroutines in the server to be executed directly by the client as if it were performing it locally. Figure 2.3 first shows an overview of the communication between the gRPC client and server and a representation of a table for a P4 program, a logical representation of a table entry for that very same table and a encoding for a table entry in P4Runtime. Each table and table entry as a set of key parameters for matching and a set of actions of which each entry can choose a single one from. In this case, the table has an entry in which the ipv4 destination address is 12.0.0.0 and a metadata field "vrf_id" of 0. The first field is matched by longest prefix first, in this case 8, and the second must be exactly the same. If a packet matches this rule, it will be dropped per the specified action. P4Runtime uses Protobuf (Protocol buffers) as the serialization method for data. The use of gRPC allows for easy extensibility, since if new operations are necessary users can simply

[1] Available at: https://p4.org/api/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane.html

write the new procedure on the controller, and provides a low-latency, highly scalable alternative over standard communication protocols, since it transports data as binary allowing for quick deployment of the sent procedure.

P4 and P4Runtime allows our solution to become independent of any specific protocols or hardware. Due to how easy they are to extend, by using P4 and P4Runtime it would be simple to add support for whatever protocol the user may require. They also improve upon the data plane programmability of PISA chips by allowing a controller to alter the way packets are handled dynamically. Through the use of the P4Runtime API the collected packet length distributions can efficiently be retrieved by the controller. Distributions that can are then classified following the methods discussed in Section 2.5

2.5 Covert Channel Detection in Multimedia Calls

A recent study [5] showed that the different techniques that are now known to establish covert channels in multimedia flows alter the packet size distribution in relation to the distribution observed in a legitimate call. These differences can be detected automatically using supervised classification techniques based on decision trees, such as Random Forests or the *XGBoost* algorithm. These techniques make it possible to perform the detection of flows carrying covert channels by collecting their packet size distributions. If it is possible to capture this distribution at line speed, for example using the capabilities offered by programmable switches, it becomes possible to detect covert channels in real time.

2.6 Space-Saving Algorithms

It is common that in order to properly monitor a network, it is necessary to either use middleboxes explicitly for that purpose or utilizing techniques that minimize the amount of data a switch has to hold (even if temporarily). Sampling is one of these techniques. Sampling is often used in order to reduce the amount of data a switch would have to store to manageable levels, while still maintaining adequate levels accuracy to perform the necessary measurements, as most switches' storage space is severely limited, even when compared to the storage of average household computers. However, sampling is not without faults, as it can lead to a misrepresentation of the data [26].

A possible alternative or complementing technique to sampling would be to use algorithms that allow for the data to be stored efficiently as estimates instead of exact metrics. This tradeoff between estimate accuracy and how much space they take up must be carefully considered when using this type of solution. Next, we describe some of these algorithms, ranging from counter based techniques, to the use of more advanced data structures like bloom filters and counting sketches:

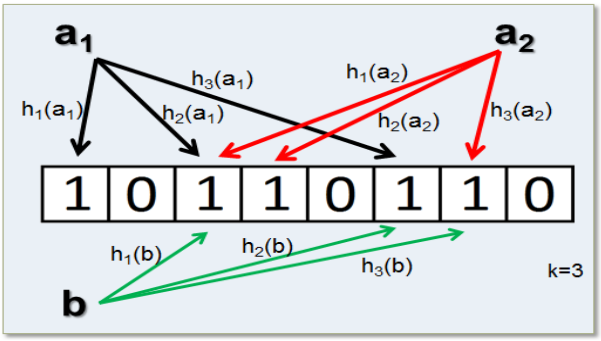


Figure 2.4: Bloom filter operation example. [2]

2.6.1 Counter Based Techniques

These methods usually use individual counters to count the frequencies of different items and hash tables to store the explicit keys of those items allowing it to report the most frequent elements. It is possible to adapt these techniques towards counting operations and making them keep track of all packet flows passing through in order to identify as many as possible. However, the estimated storage space occupied would still surpass the space required by the other techniques presented below. Counter based techniques are usually focused only on the number of items they store, ignoring implementation overheads like space usage and as such can take up a high amount of memory. Assuming we would need to keep track of 500000 individual counters of 32-bits each, that would require approximately 16 MB of storage, which already surpasses the available storage space of some switches.

2.6.2 Bloom Filters

A bloom filter is represented as an array of single-bit entries used to test whether an element is a member of a set while balancing a tradeoff between measurement accuracy and computational and storage complexity. False positive matches are possible, but false negatives are not. Elements can be added to the set, but not removed. The more elements that are added to the set, the larger the probability of false positives.

Figure 2.4 shows the operation of a bloom filter in which three items (a1, a2, b), each represented by three bits, are being inserted into the set by three separate hash functions (h1, h2 and h3). The multiple bits being used are meant to reduce the number of false positives, as all three bits must be set for an element to be present, which in this case means that b is a false positive. Usually bloom filters are used exclusively to test the membership of a set. However, counting filters provide a way to implement a way to delete elements from the set by turning each array position from a single bit to a n-bit counter. The addition operation now increments the appropriate counters and the lookup operation now checks

[2] Available at: https://redislabs.com/blog/rebloom-bloom-filter-datatype-redis/

whether each of the required entries are non-zero. Counting filters can produce similar results to the technique we will present next with the difference that the next technique uses a sublinear number of cells while a counting filters match the number of elements in the set. Usually, counting arrays operate in ranges of less than 5 MB while maintaining accuracy values above 90%.

2.6.3 Counting Sketches

Counting sketches keep track of estimates of frequencies of items. The keys for these items (identifiers) do not need to be maintained. Each item uses several counters that are incremented with each observation.

The sketch is made up of an array of counters and a set of hash functions that map items into the array. The array is treated as a sequence of rows, and each item is mapped by each of the hashes into their designated row. An item is processed by mapping it to each row in turn via the corresponding hash function and incrementing the counters to which it is mapped. The number of columns in the array is based on a user-defined variable ϵ, that defines the deviation factor between the estimate and the real value. The number of columns is usually much smaller than the number of items that need to be mapped which is why sketches only require a sublinear amount of memory to work. The number of rows is based on a user-defined variable δ, that defines the probability of the error in answering a query be within ϵ. Due to the error introduced with these two variables, when querying sketches, statistical operations like the median and minimum are used to deduce the closest possible estimate to the true value from the different values of the entries for each item. Ideally, these will produce summaries of the data they hold that are mergeable, meaning that one can process many different portions of traffic independently, and then the summaries computed from each can be quickly combined, for example through union or intersection of them, to obtain an accurate summary of various combinations. This drastically reduces memory usage, compute time, and latency compared to exact methods.

Figure 2.5 shows the operation of a counting sketch in which two items are being inserted into three different rows of the array by hash functions. All hash functions (h1, h2 and h3) are different from one another and upon selecting the correct entry the counter therein is incremented.

Two examples of counting sketches are the Count Sketch [3] and the Count-Min Sketch (CM Sketch) [4]. Both are very similar data structures, each being represented as 2-dimensional array of counters that utilize hash functions to map items to each row. However, CM Sketch's design, which we will discuss in Section 4, makes it a better alternative to Count Sketch as it can achieve similar results in smaller amounts of space.

These methods are typically optimized for hardware implementation as they use statically allocated memory and do not store explicit flow identifiers. Therefore, these methods can answer queries, that when given a flow identifier, generate an estimation for that flow. Usually, counting arrays operate in

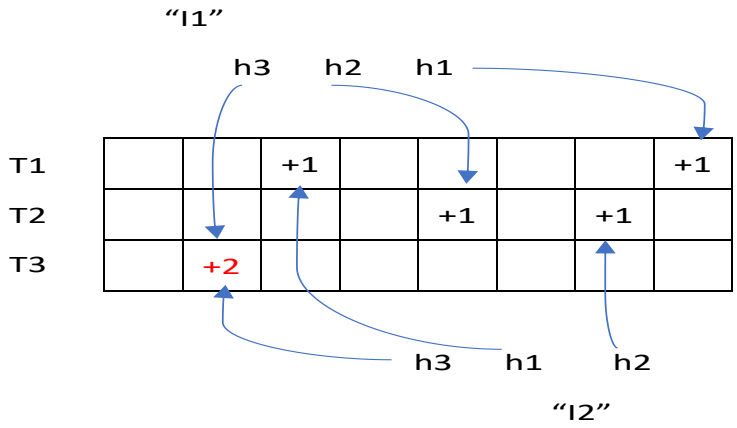


Figure 2.5: Counting Sketch operation example.

ranges of several KB while maintaining accuracy values above 90%.

Although all these techniques perform their objective (of saving storage space) they either do so to different degrees or they are optimized for different objectives. More specifically, counters, even with the use of hash tables, would still take up too much space for commodity use, and bloom filters, despite sometimes being able to conserve more storage space than sketches, are optimized for detecting the presence of elements in a set rather than counting and in order to perform the same tasks as sketches would require the use of multiple bloom filters at a time. As such, counting sketches emerge as a promising alternative for achieving the goal of keeping track of fine-grained traffic metrics.

2.7 Similar Systems

In this section, we examine and discuss previous work in the area of network traffic measurement. We start by surveying works that perform network traffic measurement at the network-level, to study their differences when compared with the type of measurements we are seeking to perform. Next, we look at some works that perform network traffic measurement at the flow-level, as to provide an overview of the sort of issues common with the solutions we are focusing on. Afterwards, we dwell on works that leverage network traffic measurements to perform more complex tasks, such as intrusion detection. We compare the above works across six different properties. Finally, we focus on works related to space-saving algorithms, to compare how different algorithms and data-structures perform and how they could improve the measurement of traffic flows.

2.7.1 Network-Level Traffic Measurement

In this section we discuss works that perform measurements of the network load across multiple network devices, like packet delay or packet loss between two switches.

OpenSample [27] presents a platform for network measurement in commodity SDNs. It focuses on ensuring low latency by leveraging packet sampling to provide near real-time measurements of both network load and individual flows. OpenSample can be integrated into already existing networks.

sFlow [28] is used to sample packets from an OpenFlow switch. These samples are used to capture two types of data. First, it captures TCP packet header samples with low overhead and extracts TCP sequence numbers from the captured headers to reconstruct nearly-exact flow statistics. Second, packet samples are used to estimate port utilization at sub-second time scales. A single, centralized collector combines samples from all switches in the network to construct a global view of traffic in the network.

In OpenSample, by sampling the header of TCP packets it is possible to obtain an estimation of the average packet lengths for the passing packets. While it uses SDN technology, which gives OpenSample a centralized view and control over all switches in the network, it is severely limited in other areas. While it would be possible to estimate the packet length distributions using OpenSample, depending on the sampling rate, the results could be inaccurate. This is due to the use of sampling as we run the risk of misrepresenting the traffic flows. Additionally, we would be limiting ourselves to only being able to measure TCP packets.

We argue that the sampling of entire packet and counters could be avoided in order to avoid misrepresenting traffic flows. Instead the space used to store the wanted metrics could be reduced and queried from the controller directly, especially as several applications may not require sub-second time scales. With this reduction, sampling would no longer be necessary and the additional computation from performing the estimation from the TCP sequence numbers would no longer be required.

OpenNetMon [29] is a network monitoring tool made for use in software-defined networks in OpenFlow. OpenNetMon seeks to provide accurate measurements while reducing network and switch CPU overhead. It does so by increasing the rate at which switches are polled when flows arrive or change frequently and decreases when flows stabilize to minimize the number of queries.

OpenNetMon focuses only on a few select measurements while OpenSample reconstructed near-exact statistics to perform measurements of both the network load and individual flows. However, OpenNetMon's measurements are less focused on counting operations and one of them requires additional packets to be inserted into the network. OpenNetMon is also not limited to TCP traffic.

OpenNetMon is able to provide accurate measurements of throughput, packet loss and packet delay across all flows. OpenNetMon continuously monitors all flows between predefined link and destination pairs. To determine throughput for each flow, OpenNetMon regularly queries switches to retrieve the amount of bytes sent and the duration of each flow, enabling it to calculate the effective throughput for

each flow. Per-flow packet loss is calculated by polling flow statistics from the first and last switch of each path and subtracting the increase of the source switch packet counter with the increase of the packet counter of the destination switch. Path delay is calculated by using OpenFlow's capabilities to inject packets into the network. At every monitored path, packets are regularly injected at the first switch and have the last switch of the path send it back to the controller to estimate the path delay.

Unfortunately, since OpenNetMon is limited by OpenFlow's regular features, it is not possible to perform our specific measurements with it. In order to do so we would have to modify the OpenFlow implementation in use and alter the way the switches function, so that they would both support the counting our packet length distributions. Additionally, the use of sampling could result in the misrepresentation of traffic flows.

While OpenNetMon is effective at calculating some complex statistics like packet transmission delay, since it cannot be measured at any one switch, the storage space occupied by the required information of each flow may prove to be too high. Specifically, every time the controller attempted to query the switch it might have to send several MBs of data across the network. Even if the switches were queried frequently to avoid this accumulation of data, it might still flood the network. The use of space-saving algorithm would address this issue. Nevertheless, OpenNetMon is an improvement when compared to OpenSample, as it performs per-flow, fine-grained measurements and does not capture packets.

Narayana et al. [30] study states that instead of designing parcial solutions to work around existing switch mechanisms, the right approach is to co-design language abstractions and switch hardware primitives for network performance measurement. They present a declarative query language that for diverse set of network performance operations that can also be efficiently implemented in switch hardware using a programmable key-value store primitive. Unlike OpenSample and OpenNetMon, and most of the other works discussed in these sections, Narayana et al. own switch hardware primitives are general enough to perform a variety of measurements while using key-value stores to aggregate information across sets of packets.

Their query language enables network operators to specify diverse performance questions, independent of their implementation on the network. Operators are able to request per-packet performance information, request traffic experiencing uncommon performance, like high queuing delays, aggregate information over packets sharing headers, find simultaneous occurrences of performance conditions and compose queries over results of other queries. Additionally, by implementing a programmable key-value store it is possible to aggregate information across different sets of packets. The value stores and programmatically updates aggregated information across packets belonging to the same key. The key-value stores are kept in cache using a least recently used (LRU) cache-eviction policy.

Additionally, Narayana et al. [30] used programmable switches to access available metadata to better support certain performance-related queries and realize several constructs necessary to perform them.

Seeing as the query constructs are implemented directly within the programmable switch, it would not be too difficult to perform our own measurements with this language, by adding our own if necessary. However, to obtain packet length distributions several key-value stores would be required for each flow. Additionally, the filtering operation is also easily supported by the query language. However, this would lead to either a heavy use of space to maintain several stores for each flow or we would quickly run the risk of losing metrics due to the use of the LRU policy for the key-value store cache. Additionally, since it does not use SDN technology, it may be difficult to perform the measurement over different switches.

While this study provides a general language to perform highly diverse measurement tasks, it does not take into account the heavy storage space usage that could be built up with time, especially when the often small amount of storage space some commodity switches are limited by.

When compared to OpenSample and OpenNetMon, we can see that it does not require sampling and is not bound to any specific protocol or hardware, however it is not easily integrable as each network device would have to be manually reconfigured every time a change in the network's operation occurred.

2.7.2 Flow-Level Traffic Measurement

In this section we discuss works that perform measurements of individual flows in network devices, like packet sizes and protocol usage ratios. As a whole, these works have more to do with the traffic passing through the devices and the performance of the devices themselves as opposed to the entire network.

Duffield et al. [31] discusses several methods on the use of flow statistics formed from sampled packets to infer additional information on the missing unsampled flows. They focus mainly on the accuracy of the estimations of the unsampled flows while minimizing the consumption of resources by the measurement operations. Duffield et al. do not require the capturing of packets.

This work is very similar to OpenSample [27]. They both estimate statistics of unsampled flows through the use of TCP header fields and statistical inference techniques. However, Duffield et al. make even finer use of TCP details and general-flow binomial distributions to perform their estimations, all without capturing packets themselves.

Through the presented methods, they are able to deduce both the number of missing flows that evaded sampling and the distributions of their lengths. This is achieved through statistical inference, and by exploiting protocol-level detail reported in the flow records, more specifically by using the number of flows containing TCP packets with a set SYN flag to estimate the total number of TCP flows. They most commonly obtain flow statistics from Cisco's NetFlow [11] and packet samples from Inmon's sFlow [28], which are then exported from routers to a collector.

It is not possible to perform our measurements using the techniques here as we would run into the same problems we saw in OpenSample. Similarly to OpenSample, using the samples provided would allow us to obtain the packet length distributions. At the same time, we would once again be limiting

ourselves to only being able to measure TCP packets, and having to rely on sampling once again.

We argue that if the reduction of consumed resources is one of the driving focus of this thesis then sampling is unnecessary to accomplish that. Data sketches only require sublinear space (a matter of KB at most) and it is feasible that sketches could accomplish a similar quality of metrics without additional storage required (perhaps even requiring less storage based on the sampling ratio). This is in addition to avoiding the added complexity incurred from trying to estimate the length of the unsampled flows.

FlowRadar [32] is an attempt at improving NetFlow [11], a feature on Cisco routers that provides the ability to collect and monitor IP network traffic, while removing the need for sampling. FlowRadar is the only work we discuss which is a direct improvement of an already existing technology, NetFlow. Like OpenSample [27] and Narayama et al. [30], FlowRadar can perform a variety of measurements. However, unlike them, it does not need to perform sampling through the use of a space-saving algorithm, based on bloom filters [15], to reduce the size of the data obtained from the measurements.

As it is an improvement over NetFlow, FlowRadar is able to provide the same type of records as the original NetFlow including, but not limited to, packet and byte counts for flows, flow duration, layer 3 headers and source and destination ports (if applicable). FlowRadar can scale up to a large number of flows with small memory and bandwidth overheads by encoding flows and their counters into a small fixed memory size that can be implemented in merchant silicon with constant flow insertion time at switches. The encoded flowset data structure made up of two parts. First a flow filter which is just a normal bloom filter with an array of 0s and 1s, which is used for testing if a packet belongs to a new flow or not. Second is a counting table which is used to store flow counters. Each entry of the counting table includes an XOR of all the flows in a bin, a counter for the number of flows mapped in the bin and a counter for the number of packets mapped in the bin. This allows the remote collector to perform network-wide decoding and analysis of the flow counters. Thus, all the flows can be captured without sampling and periodically exported to the remote collector in short time scales. Afterwards, given the encoded flows and counters exported from the different switches, the remote collector can perform network-wide decoding of the flows, and temporal and spatial analysis of the flows for numerous different monitoring applications.

FlowRadar is a big improvement over Duffield et al. [31] study, as besides not requiring packet capturing, it performs per-flow fine-grained measurements, does not use sampling and uses a space-saving algorithm.

FlowRadar can be extended to obtain our packet length distributions, uses a form of space-saving algorithms and was even implemented using P4. However, due to the 3 fields the counting table needs to decode the information gathered by the switch adding the additional complexity of tracking the frequency of different packet sizes per flow would greatly decrease FlowRadar's efficacy. However, since it does not utilize SDN technology, it would not be able to support our operations since it would not be possible

to change switch's behavior at run time.

UnivMon [33] presents a framework for general-purpose flow monitoring for use in SDN environments which provides high accuracy measurements. UnivMon is the only work we discuss that performs only general-purpose measurements. By collecting small amounts of specific data during the measurement we can, afterwards, extract different statistics from them. Even if the original measurement was performed only for a single statistic. To reduce the amount of memory utilized, it leverages sketching algorithms. UnivMon is also the only work we discuss here that is implemented over P4.

Univmon's data plane uses what are called universal sketches, which have their foundation on the concept of universal streaming. Universal sketches hold multiple sketch instances each performing the same measurement operation, like "heavy hitter" detection. Each subsequent instance may only count a packet if all previous instances have counted it as well, at which point it may sample the packet with some probability. Universal sketches can only perform measurement operations over metrics that satisfy a series of statistical properties. By utilizing universal sketches additional information may be extracted from these measurement operations that could not be extracted otherwise. For example, extracting information on heavy hitters and DDOS victim detection simultaneously.

The UnivMon control plane queries the data plane and runs simple estimation algorithms for every management application of interest and allows applications to run estimation queries on the collected counters. The control plane generates sketching manifests that specify the monitoring responsibility of each switch. These manifests specify the set of universal sketch instances for different dimensions of interest (IPs, 5-tuples, among others) that each switch needs to maintain for different origin-destination pair paths that it lies on. Periodically, the UnivMon controller gives each switch a sketching manifest which specifies the dimensions for which it needs to maintain a sketch. When a packet arrives at a node, the node uses the manifest to determine the set of sketching actions to apply. When the controller needs to compute a network-wide estimate, sketches, pulled from all nodes and for each dimension, are combined across the network for that dimension. This method minimizes communication to the control plane while still making use of the controller's ability to optimize resource use.

An example of a Network Traffic Measurement task that can be accomplished with UnivMon is the detection of change, which is the process of identifying flows that contribute the most to traffic change over two consecutive time intervals. This computation takes place in the control plane, so the output of the universal sketches from multiple intervals can be stored there without impacting online performance. By comparing the volume of flows in two adjacent time frames. If the difference in volume of a flow between the first sketch and the second one is higher than a predefined percentage of total change in all flows, then that flow is said to be a heavy change flow.

UnivMon not only showcases most of the same properties as FlowRadar but also is easily integrable into network and is not dependent of any specific protocol and hardware. However, unlike FlowRadar,

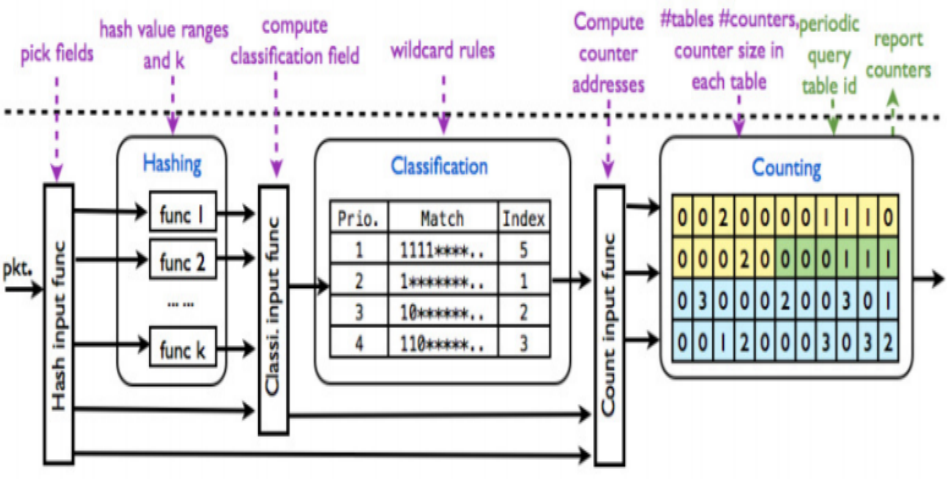


Figure 2.6: OpenSketch Data Plane. From the OpenSketch [2] paper.

its measurements are not fine-grained and requires multiple instances of the same sketch for some of the measurement tasks.

Despite UnivMon utilizing space-saving algorithms, SDN technology and being both hardware and protocol independent, it is not possible to perform our measurements efficiently over it. Since employing multiple sketch instances of a measurement for obtaining packet size distributions at different sampling rates would not provide us with any additional information than using a single instance without sampling. This is especially unfortunate since UnivMon already supports its own filtering operation. Being application agnostic can be useful, however not many applications require focus on all available sub-regions of interest. Especially considering that multiple dimensions require multiple sketches which require more memory.

OpenSketch [2] proposes a traffic measurement architecture with a focus on maintaining a careful balance between measurement generality and efficiency. OpenSketch can support a wide variety of measurement operations by implementing different space-saving algorithms in the switch and simply altering the amount of memory they require and how they interact with each other. Like some other works discussed here, it utilizes SDN technology, allowing the controller to automatically configure the switch in order to perform the desired measurement operations.

OpenSketch maintains high link speed and low cost by utilizing one of its many sketches optimized towards the required operation or, when some traffic characteristics are not directly supported by sketches, using a simpler sketch and leaving the complex data analysis for the controller.

The data plane has three stages as seen in Figure 2.6: First, a hashing stage used to reduce the measurement data, which picks the packet source field and calculates a single hash function. Next, a classification stage used to select flows, which picks the packet destination field and filters all the packets matching a rule. Each rule has a index used to calculate the counter location in the final stage. And

finally, a counting stage to accumulate traffic statistics, which uses a counting input function to calculate the index of the bit to be updated using the hash value of the source field.

Its data plane architecture allows OpenSketch to be generic and support various measurement tasks but in order to also be efficient and save switch memory only a few simple hash functions should be used. The OpenSketch controller provides a sketch manager that automatically configures the sketches with the best memory-accuracy tradeoff and a resource allocator that divides switch memory resources across measurement tasks. An example of such a task is the detection of heavy hitters. Employing the Count-Min Sketch allows OpenSketch to measure the volume of flows. And employing a reversible sketch allows it to construct a reversible array to get the keys for each count. Using both it is possible to identify the flows in the Count-Min Sketch with heavy counts.

OpenSketch, unfortunately, has an initially high memory consumption due to the extra steps of hashing and classification in the data plane, therefore for some tasks the additional memory cost (and complexity if the necessary operation requires the combination of different hashing, classification and counting techniques) is unnecessary in the presence of simpler, more efficient solutions. If the hashing and classification steps could have been supported with P4, specific attributes of packets could have been targeted by the rules added/modified by the controller.

OpenSketch can support our measurements. As it already supports the count-min sketch we can simply design our own measurement task. This task would have to modify three phases of OpenSketch's data plane. First, the hashing stage would need enough compatible hash functions to meet our accuracy requirements. The classification phase in conjunction with the hashing phase would need to separate the packets not only by flow but by size as well. Finally, the counting phase would have to include enough count-min sketch counters to reduce possible collisions within the sketch. However, there are two main problems with using OpenSketch for our measurements. First, it is not hardware independent, meaning that it would have to be adapted to different hardware configurations. Second, it is unable to support our filtering operation to the degree of detail we seek as it is limited to a certain set of network protocols.

2.7.3 Space-Saving Algorithms

Space-Code Bloom Filters (SCBF) [34] are a variation of bloom filters for per-flow traffic measurement. They focus on achieving reasonable measurement accuracy with very low storage and computational complexity.

SCBF represents a multiset, extending the capabilities of traditional bloom filters. Like bloom filters, SCBFs can check if an element is in the multiset but it can also count the number of occurrences of that element. A SCBF uses multiple groups of independent hash functions, each of which can be viewed as a traditional bloom filter. When inserting an element into the SCBF, one of the groups is selected at random and the bits obtained from hashing the element with the different hash functions are set to

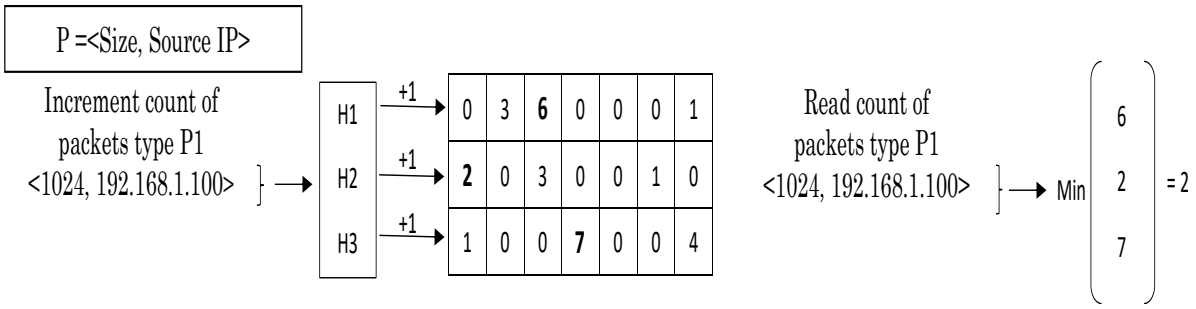


Figure 2.7: Representation of the Count-Min Sketch.

1. While querying the SCBF, we count the number of groups where all bits for an element are set to 1. Based on the number of groups obtained, called observation, we can estimate the multiplicity of the element and return it as the result. However, the existence of heavy-tailed distributions, which are distributions that have heavier tails than the exponential distribution, implies a high multiplicity of elements. Since the group is chosen at random during the insertion operation it is possible that all groups will have the element inserted into them. This unfortunately means that if more elements were to be added to the SCBF, there would be no difference if all groups were already filled.

The solution to this problem is the multiresolution SCBF (MRSCBF), in which the authors employ multiple SCBFs to cover the entire range of multiplicities. For MRSCBFs, when a packet arrives it will be inserted into each SCBF with certain sampling probabilities. SCBFs with high sampling probabilities will keep track of smaller flows and vice-versa. The querying operation is similar to regular SCBFs, however the final estimate is obtained from the estimates of the most relevant SCBFs. One method for estimating the result is the maximum-likelihood estimation (MLE). MLE can be thought of as finding the most likely multiplicity of an element that would have caused the observation of that element. The other method used is mean value estimation (MVE), which computes the multiplicity which is expected to have caused the observation on average.

SCBFs require an increased usage of storage space which is not required for sketches as instead of counting each instance of an element in a multiset (represented by the SCBF) simply employs counters in the structure itself.

The Count-Min Sketch [4] (CM Sketch) is a sublinear space data structure that allows the summarizing of data streams, or sets of packets. CM Sketch allows fundamental queries in data stream summarization such as point, which returns individual items, range, which returns the sum of a sequence of items, and inner product queries, which returns the sum of the product of sequences of items of different sets, to be approximately answered very quickly.

Figure 2.7 illustrates the operation of a CM sketch with three hash functions (H1, H2 and H3) while processing a packet identified by the tuple P = $<$*Size*, *Source IP*$>$. Upon receiving the packet P1 = $<$*1024*, *192.168.1.100*$>$, the sketch groups the values of P1 and uses each hash function to select

the corresponding increments. Intuitively, knowing the space and amount of identifiers of the items to be recorded, it is possible to configure the number of rows and columns of the matrix so that the probability of collisions in all rows is small (this value is a sketch configuration parameter). Finally, it is possible to read the P1 packet count recorded by the sketch by grouping the values belonging to the tuple and gathering the values of the registers corresponding to each hash function. The sketch returns an approximation of the actual count of P1 packets by choosing the lowest value obtained from each line. The reasoning for this decision focuses on the observation that the value of each line never returns a count lower than the actual value of the item and that higher values result from the occurrence of collisions.

A CM sketch can be applied to solve several important problems in data streams such as finding quantiles and frequent items for example. The time and space bounds used for the CM sketch to solve these problems significantly improves those previously known, typically from $1/\epsilon^2$ to $1/\epsilon$ in factor. A CM Sketch with parameters is represented by a two-dimensional array counters with width w and depth d, which given parameters (ϵ,δ), makes w=[e/ϵ] and d=[$ln(1/\delta)$]. Each entry of the array is initially zero. Additionally, d hash functions are chosen uniformly at random from a pairwise-independent family. This sketch shows itself to be exactly what we require, because not only does it utilize sublinear storage space but also as a sketch it is optimized towards frequency counting, and therefore it is difficult to develop a more efficient data structure at the present. For our purposes, we will perform mostly point queries, which are performed by identifying the smaller entry for the specific item.

2.7.4 Applied Network Traffic Measurement

In this section we discuss works that utilize network traffic measurements to perform more complex tasks like DDoS detection or providing quality of service.

Sonata [35] is a system for performing network monitoring line-rate queries at switches for tasks like anomaly detection, real-time application performance analysis and distributed port scan detection. It does so through streaming analytics, which is a method for performing real-time computations on data streaming from applications. The addition of scalable stream processors, devices which exploit parallel processing to perform a great number of operations in a short amount of time, to the network allow Sonata to fuse data streams to answer more sophisticated queries about the network in real-time.

Sonata, like Narayana et al. [30] study, allows users to specify queries to perform measurements. However, they do not define their own hardware primitives for their measurements and instead generates a set of rules from the queries to install. Additionally, while Narayana et al. was capable of aggregating packets, Sonata is able to better refine their measurements.

Sonata was implemented with switches running OpenFlow. It uses a runtime system that compiles queries to generate a set of rules to install in the switch data plane and processing pipelines at the stream processor. Additionally, the data-plane operations ensure that traffic filtering is based on relative

sampling rates for different flows and that the rate of the filtered data is always less than the system-defined constraints. By abstracting the packet headers and payload as tuples, Sonata facilitates the writing of queries since it allows users to address each packet tuple as a collection of fields, like source and destination address, bytes and ports. Sonata allows users to specify whether a specific operation should be performed at the switch or at the stream processor therefore reducing the workload on them. Finally, Sonata allows users to refine queries based on dynamic conditions. For example, if two different queries are being performed concurrently, then one could sample the entire traffic at a lower sampling rate while the other could sample only the traffic selected by the first query at a higher rate. Query partitioning and refinement allows Sonata to immensely reduce the number of counters and data rate required to efficiently perform measurements.

Although Sonata makes use of SDN technologies and the query language is expressive enough to support our filtering operation, it suffers from the same limitations as OpenNetMon. Due to OpenFlow's limited features it is not possible to perform our specific measurements with Sonata. In order to do so we would have to modify both the OpenFlow implementation in use and the switches function, so that we would be able to query the switch for our metrics. Additionally, the use of sampling could result in the misrepresentation of certain traffic flows. While this is an elegant and scalable solution to reducing the traffic to be analyzed to a small subset of what passes through the switch it relies on sampling to reduce the amount of traffic it has to inspect. Therefore, making it unfit to what we require, which would require the measuring all traffic.

Huang et al. [36] study on the identification of applications aims at improving Quality of Service (QoS) in SDN environments using Machine Learning and DNS Responses. They use a hybrid approach of DNS response identification and ML in order to maximize their accuracy.

Similar to OpenSample [27], Huang et al. capture packets in order to obtain flow attributes. They do so to generate their ML model. However, unlike OpenSample, Huang et al. do not collect only TCP traffic. Additionally, while OpenSample only captures a portion of traffic depending on the sampling rate, Huang et al. do not use sampling.

For the DNS-based method IP addresses and domain names are collected from DNS responses during the training phase. Afterwards, in the classification phase, they check whether source or destination IP address of a flow matches any IP linked to an application name. For ML however, a port mirrors traffic to a traffic classifier in order to obtain flow attributes, like number of packets, packet sizes, throughput and others. After the attributes of a flow are calculated, it obtains a label for the flow a mapping of attributes and a label, which are then used to identify the corresponding applications. Overall, their system works in two phases. First, a training phase, in which the system generates data required for the classification, by either training the ML model or collecting IP address mappings for the DNS-based method. Second, a classification phase, during which either the DNS-based method or supervised ML

will be used based on whether or not some requirement is met.

It is possible to perform our measurements by mirroring all traffic towards the classifier and collecting the metrics there and it is even possible to perform filtering through the addition of rules to the SDN switch. However, this requires a heavy use of bandwidth to send all the required packets to classifier and a high amount of memory to keep all the metrics as it does not employ nay space-saving algorithms. Additionally, although it is possible to perform filtering, it can only be done for the protocols supported by OpenFlow and the switch.

The problem with this, however, is how both methods obtain the packets necessary for their operation. Whether it be the DNS-based method or the ML-based method, it is necessary to capture packets for proper classification. This incurs heavy bandwidth usage as entire portions of flows may need to be cloned somewhere other than their original destination and a requires a high amount storage space for storing the necessary packets.

Athena [37] is a SDN-based framework for scalable anomaly detection. It intends to support sophisticated anomaly detection services across the entire network data plane while paying special focus to usability, scalability and overhead.

Athena offers API's to users so that they can select the features Athena should collect similar to a query language. Like Sonata [35], the features collected are turned into SDN rules that are installed into the switch and are limited by OpenFlow's capabilities. Additionally, Athena measurements do not require sampling, although sampling rates can be specified. Athena queries can also aggregate information similar to Narayana et al. [30] study but do not possess the same level of refinement as Sonata.

The Athena system is composed of three major elements: the Athena Southbound (SB) Element instances, which isolates control messages, extracts features to drive the analysis algorithms and mitigates detected problems, run above multiple parallel controllers for scalability and allows for monitoring and issuing of control messages. The Athena Northbound (NB) Element, which exports Northbound APIs for application developers to utilize Athena's functionalities for anomaly detection for scalability and transparency. The Athena Off-The-Shelf Strategies, which include preset anomaly detection strategies. The features required for these strategies belong to one of these three categories. Protocol-centric features, which are directly derived from control messages and include packet and byte counts. Combination features, which are derived from pre-defined formulas using protocol-centric features and include flow packet counts and duration. And Stateful features, which are the states of the network operations and include the pair flow ration. In real-time, users can use Northbound API's to specify to Athena what features to collect from traffic and what kind of anomaly detection ML model to train using the collected features. These ML models can then be used to detect anomalies from future collected features.

It is not possible to perform our measurements on Athena without modifying OpenFlow's features and the switch's functioning. This is due to OpenFlow's features not being modifiable to collect packet

Table 2.1: Comparison of properties of works discussed to our solution. (PFQPLD:Per-Flow Quantized Packet Length Distributions, PC:Packet Capture, S:Sampling, SSA:Space-Saving Algorithms, EN:Existing Networks, DSPH:Dependent on Specific Protocols or Hardware)

	PFQPLD	Does Not Use PC	Does Not Use S	Uses SSA	Integrable into EN (via SDN)	Not DSPH (via Prog. Switches/P4)
Duffield et al. [31]	✗	✓	✗	✗	✗	✗
OpenSample [27]	✗	✗	✗	✗	✓	✗
OpenNetMon [29]	✓	✓	✗	✗	✓	✗
Sonata [35]	✓	✓	✗	✗	✓	✗
Huang et al. [36]	✓	✗	✓	✗	✓	✗
Narayana et al. [30]	✓	✓	✓	✗	✗	✓
Athena [37]	✓	✓	✓	✗	✓	✗
FlowRadar [32]	✓	✓	✓	✓	✗	✗
UnivMon [33]	✗	✓	✓	✓	✓	✓
OpenSketch [2]	✓	✓	✓	✓	✓	✗
Our Solution	✓	✓	✓	✓	✓	✓

length distributions like we require. While Athena does use SDN, allowing it to manage multiple network devices simultaneously, it does not use space-saving algorithms to reduce the amount of space used. Additionally, due to OpenFlow's fixed-function, it cannot operate over all protocols and hardware.

Unfortunately, the features gathered by the switches, which are then sent to the Athena instances through control messages, are stored without the use of any space-saving algorithms the storage space used may raise to worrying levels. Similar to the previous two works, Athena provides per-flow fine-grained measurements but lacks the necessity for sampling or packet capture.

2.7.5 Discussion

Among all the works in the previous sections, only a select few utilized space-saving algorithms and even fewer chose to use counting sketches. As we suggested for many of those that did not use them, sketches could provide a more efficient way to store metrics. In the case of the ones that utilized sampling, the use of sketches could have made the choice to do just that unnecessary. Counting sketches require only a sublinear amount of storage space and are optimized towards counting operations, and, in some cases, making so that what could potentially be terabytes of data be represented in just kilobytes instead. In some cases, it could even be more efficient than using sampling at all (depending on the sampling ratio) and, as is the case of Duffield et al. study [31], could bypass having to perform some operations that would incur additional complexity.

Additionally, in FlowRadar [32] we saw that besides the flow counters, they also encoded the flows themselves into their data structures so that network wide decoding could be employed. However, we argue that by employing a system similar to FlowRadar in a SDN environment, there would be no need

for doing such a thing, as the dynamic control it provides us over the data plane, it would be trivial to match each flow to their counters. Either by simply keeping track of some identifier for each specific flow in the controller, or if P4 is being used, tagging each packet of each different flow with a unique identifier and simply matching them across network devices. Table 2.1 summarizes and compares the works we discussed in the previous sections to our solution according to the six properties mentioned earlier.

Summary

This chapter introduced key concepts and advancements of network traffic measurements, SDNs, programmable switches, the P4 language, covert channel detection and space-saving algorithms, all of which we make use in our own architecture. Additionally, we analyzed several systems that despite incorporating some of these concepts into their architecture, employed certain architectural decisions that made them unfit to perform the same measurements we aim to. In the next chapter, we present our proposed architecture that allows us to not only perform our measurements over a high amount of flows, but can do so while minimizing the impact to the networks regular operation and working within the limited memory within certain network devices. This is in addition to being able to obtain measurements able of performing accurate covert channel identification minimally affected by the memory restrictions.

3

Architecture

Contents

This chapter introduces our architecture, a flow measurement system that allows for the monitoring of a high number of network flows while maintaining high accuracy for covert channel identification. The chapter is organized as follows. Section 3.1 expresses the goals that need to be fulfilled by our system. Section 3.2 presents the basic functioning of our system. Section 3.3 details the main operations of the switch including the various Count-Min sketch [4] variations we employed in our system for testing. Section 3.4 explains the functioning of the SDN controller and classification algorithms we employed in our system. Section 3.5 describes the added 2-phase architecture that expands on our initial design. Finally, Section 3.6 describes the first implementation of our system on a P4 simulator allowing us to program the switch.

3.1 Goals and Requirements

Our architecture aims at enabling network operators or system administrators to collect flow metrics about packet length distributions so as to identify covert channels in Skype multimedia connections with high-precision.

To accomplish this goal measurements must be as unobtrusive as possible to reduce any detriment to the network's performance or exceeding the memory capacity of whatever network device that may be used to obtain them. Sketches can aid us here. However, this should not come at the cost of a significant loss in metrics precision. To save memory, sketches have the ability to compress packet size frequencies. However, these frequencies should remain sufficiently accurate as to correctly identify covert channels with high precision. Therefore, our objective should be to provide a sketch configuration that can provide both high fidelity in the obtained packet length distributions while still maximizing the number of simultaneous flows that can be measured.

After obtaining the measurements at a network node (like a P4 switch), a separate machine should be responsible for identifying covert channels by performing the machine learning-based classification proposed in prior work [5]. These classifiers should previously be trained to correctly identify each type of network flow (with or without covert channels) before receiving any metrics from the network device.

3.2 Architecture Design

Our system, illustrated in Figure 3.1, makes use of programmable switches and SDN technology to create an execution pipeline capable of intercepting flows, counting packets, retrieving flow distributions and classifying. The outcome of the classification consists of a score associated with each flow which reflects the likelihood that a covert channel is present in that particular flow. The logic of our system is divided between two major components: the SDN controller and the P4 switch. Our system focuses

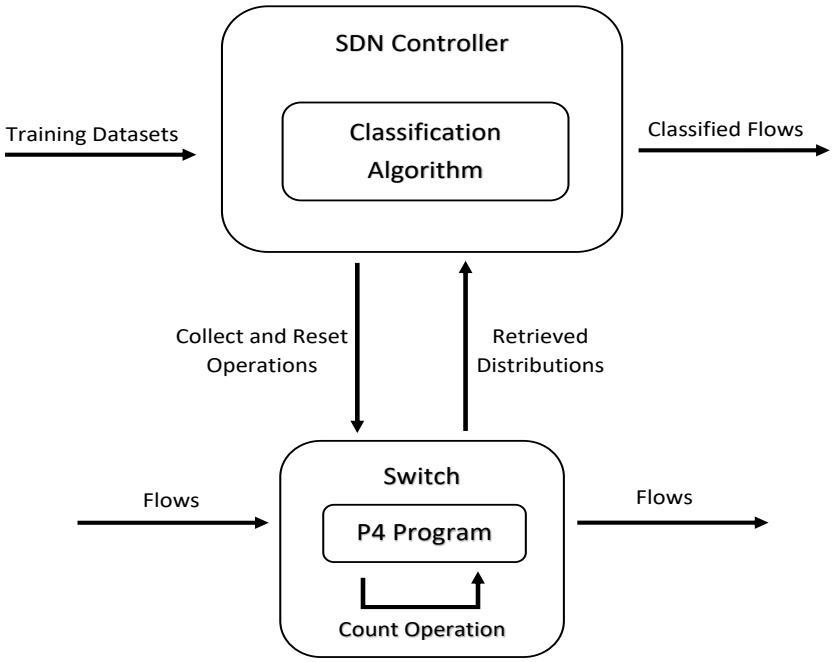


Figure 3.1: System architecture.

on obtaining packet length distributions to populate the traffic differentiation mechanisms capable of detecting the existence of covert channels in multimedia flows as analyzed in Section 4.

The P4 switch is mainly responsible for the counting operation. It uses two different data structures: a Sketch whose counters measure the packet length distributions for the different flows. And a Flow Table to keep track of the identifiers of flows whose packets are being counted on the sketch counters. Each packet is identified through certain header field combinations and range of values their size fall into called buckets (for example, the packets of size between 10 and 20 bytes). Additionally, at the request of the controller the P4 switch is responsible for gathering the distributions stored in the counters and supplying them to the SDN controller as well as resetting both structures to their initial empty state. Section 3.3 goes into more detail on the behaviour of the switch.

The SDN controller is responsible for periodically retrieving the metrics from the switch and resetting its data structures. The controller is programmed to execute a classification algorithm capable of identifying covert channels within Skype flows. In order to perform this operation, the algorithm must first be trained with datasets anticipating the conditions of the expected traffic, taking into consideration factors like sketch configurations, number of concurrent flows, bucket size and more. Finally, the execution of this algorithm leads to the classification of the analyzed flows with with varying accuracy according to the previously mentioned factors. Section 3.4 goes into more detail on the behaviour of the controller.

3.3 P4 Switch Behaviour

The P4 switch is responsible for collecting the metrics necessary for the identification of covert channels and providing it to the SDN controller. As such, the P4 program installed into the switch possesses two types of data structure, a flow table for storing the identifier of flows being measured and sketches which maintain the approximate distributions of the flows, and must be able to perform three operations:

- *Count*: Increment the sketch counters corresponding to each flow (updating the appropriate entry in the flow table if necessary) upon the arrival of a packet at the switch.
- *Collect*: Provides the flow representations and distributions to the SDN controller upon request.
- *Reset*: Resets the flow tables and sketches upon request from the SDN controller.

3.3.1 Flow Table

The flow table consists of an array of entries meant to store the flow identifiers of all the flows currently being measured. This identifier can be made up of different fields of the flow's packet headers. When dealing with flows, the most commonly used identifier is the TCP/IP 5-tuple comprising of the source and destination IP addresses, source and destination ports and protocol used.

The purpose of the flow table is to both maintain a list of all the flows being measured and to limit the number of concurrent flows being measured. The accuracy of the distributions is reduced the more flows are measured by the sketch. This can be due to having the same counters being shared by more flows. Another reason for it can be due to reducing the amount of memory allocated to each flow sub-sketch as will see in the next section with one of our sketch variations. Since the list only maintains a set of identifiers its memory requirements are negligible when compared with the sketch's. As such we will not consider it in our calculations further on in the experimental evaluation.

Upon the arrival of a packet the flow table must first be checked to see if the corresponding flow is already being counted in the sketch. If so, then the sketch will increment the corresponding counters. If it is not present, a new entry in the flow table needs to be allocated. If there are empty entries, the flow is added to the flow table and the sketch will increment the corresponding counters. Otherwise, the packet is ignored and forwarded as necessary. However, as performance is of great importance within the switch it is important to minimize the time spent read and writing to the flow table. This is achieved by hashing the flow identifier to locate a single entry in the flow table. Without having to check each entry of the flow table one by one, the action of checking if the flow is already present in the table (and consequently, in the sketch) can be performed much faster. However, this will lead to collisions between flow identifiers which result in flows being ignored despite some entries still being available at the table.

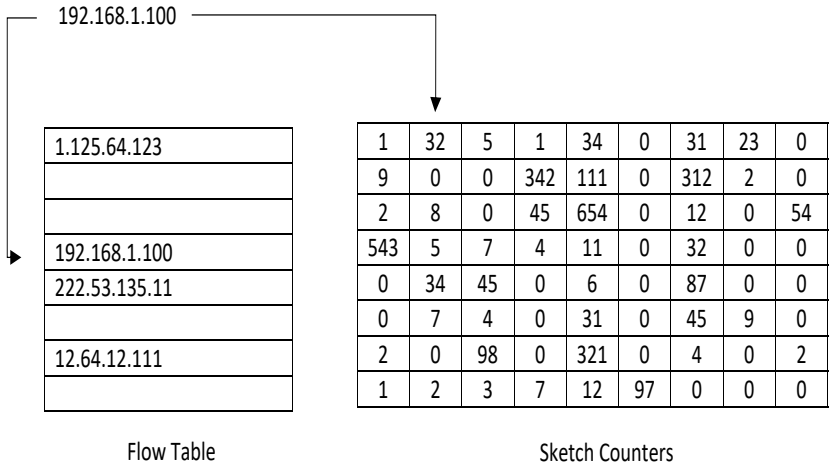


Figure 3.2: Flow Table Operation.

To illustrate this process, Figure 3.2 shows the behaviour of our system when counting a packet identified by the IP 192.168.1.100 checking its corresponding entry in the flow table and finding that it is already present. In this case, the program will identify the respective counters and increment them. It should be mentioned that the above representation of the sketch counters is fully abstract and does not fully represent the real implementation of our sketches. In fact, in our architecture we adopt several CM sketch variations that will be explained in the following section.

3.3.2 CM Sketch Variations

In this section, we begin by describing an approach for the collection of flow metrics that does not take as a basis the use of a sketch and that therefore is expected to consume a large amount of switch memory. Then we introduced several variations of the CM sketch, describing the advantages and limitations of each variation for the accurate collection of metrics. The success achieved by each sketch in characterizing and differentiating different classes of flows is analyzed through the experimental study presented in Section 4.

3.3.2.A Absence of Sketch

In the absence of a sketch or the No Count-Min Sketch (No-CM) variation, the switch keeps a record for each packet size bucket belonging to a given flow. Each bucket corresponds to a range of values of sequential packet sizes. To accommodate the collection of a large number of flows, it is possible to represent the packet size distribution of each flow in a summary form by compressing this distribution by increasing the range of each bucket. However, it is possible that the use of this method may result in

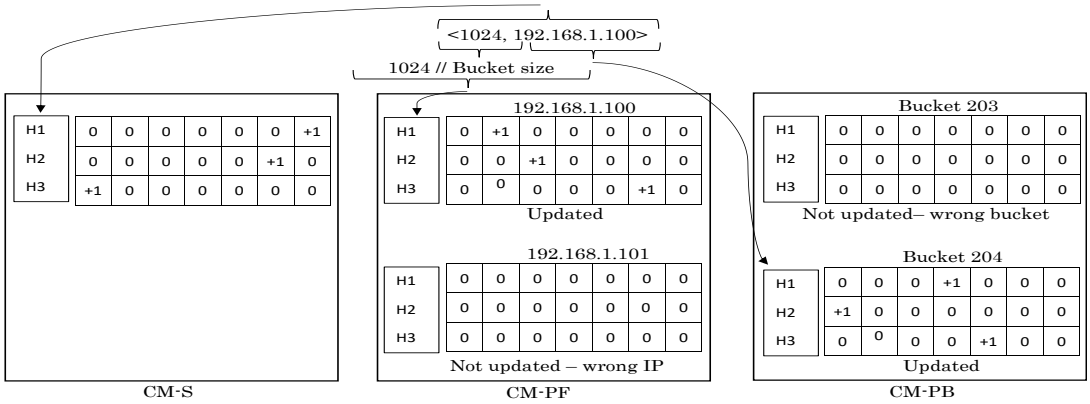


Figure 3.3: Example of the update of each sketch when receiving a packet.

the inaccurate representation of the packet size distribution of each flow.

3.3.2.B CM Sketch Simple

The Count-Min Sketch Single (CM-S) variation consists of a single CM sketch, the operation of which was previously described in Section 2.7.3. When a packet is received by the switch, a register in the sketch is incremented corresponding to the application of the hash function on a tuple that includes: (a) the bucket value to which the packet size corresponds and (b) the flow identifier. We remind the reader that the identifier of a flow is composed of a set of fields common to all packets belonging to the flow, such as the source/destination IP and the ports used. In the CM-S sketch, the counting of packets of a given bucket belonging to a flow may be greatly influenced by packets (in this or another bucket) belonging to any other flow. Since each flow has a different identifier, the resulting tuple that is used as input to the hash function is always different, which can reduce collisions to values that still allow to differentiate the flows from the estimation of the distribution that results from the use of the sketch. It should be noted that altering the range of values for the buckets will result in potentially drastic changes in the accuracy of the distributions and consequently the precision of the covert channel identification. By increasing the value range the accuracy of the distributions, we decrease the total number of buckets in return for a less detailed distribution of the packet lengths. Furthermore, by reducing the range we increase the number of buckets in exchange for a more detailed representation of the distributions. Figure 3.3 illustrates the counting of a packet identified by tuple P1=$<$*1024*, *192.168.1.100*$>$ in the CM-S sketch (on the left).

3.3.2.C CM Sketch per Flow

The Count-Min Sketch Per Flow (CM-PF) variation is composed of an CM sketch for each flow, each of which can be considered a sub-sketch of the CM-PF. When a packet is received by the switch, the switch checks the corresponding flow by attaching it to a specific entry in the sketch. This entry is determined

by another hash function. If the entry obtained is already associated with another flow, the new flow is not registered. The register to be incremented is selected based on the application of the hash function to the bucket identifier corresponding to the packet size. Since this behaviour is similar to that of the flow table the same hash operation may be used to identify both the correct table entry and sub-sketch. The CM-PF sketch is based on the observation that collisions in the registers of each bucket will occur equally for each existing flow (since all CM sketches are configured with the same hash functions). Thus, by compressing a packet distribution into a limited set of buckets, it is expected that different classes of flows will cause a signature to emerge with sufficient information to differentiate the various classes. Changing the value range of buckets will impact the accuracy of the distributions similar to the CM-S variation. Figure 3.3 illustrates the counting of a packet in the CM-PF sketch (at the centre). Only the first entry of the sketch is updated since it corresponds to the flow to which the packet received belongs.

3.3.2.D CM Sketch per Bucket

The Count-Min Sketch Per Bucket (CM-PB) variation is composed of an CM sketch for each packet size bucket, each of which can be considered a sub-sketch of the CM-PB. Each sketch occupies the same memory. When a packet is received by the switch, the CM-PB sketch increments the records corresponding to the application of the hash function on the flow identifier in the CM sketch for the size bucket of the packet concerned. In the CM-PB sketch, all the flows that collide in one of the sketches will collide in all sketches. However, if the colliding flows have similar distributions, these collisions do not affect the resulting distribution. On the contrary, if one of the colliding flows presents a distinct distribution, it may still be correctly identified, which is desirable in the context of this work. Similarly to the previous variation, alterations to the value range of the buckets will impact the final accuracy. By reducing the number of different buckets, we reduce the number of sub-sketches thus increasing the size of each of them. Similarly, increasing the number of buckets and sub-sketches will reduce the size of each of them. Figure 3.3 illustrates the counting of a packet received in the CM-PB sketch (on the right). Only the second entry of the sketch is updated since the bucket to which this sketch belongs corresponds to the size of the packet.

This concludes the discussion of the behaviour of the P4 switch and in the next section, we will begin discussing the behaviour of the SDN controller.

3.4 SDN Controller Behaviour

The SDN controller interacts with the switch using two operations supported by the P4Runtime API:

- *Collect*: Periodically retrieves the flow representations and distributions from the switch in order to classify them. Both the values from the flow table and the sketch employed are retrieved.

- *Reset*: Resets the flow tables and sketches after collecting the most recent data.

Afterwards, the metrics are classified by a classification algorithm, which is trained with a profile appropriate for the gathered flows. Next, we provide more details on how to generate training datasets and how the classification algorithm works.

3.4.1 Configuration Space and Dataset Generation

In order to train the classifier to correctly identify the we must generate training datasets based on the configuration employed. This configuration is meant to emulate the certain properties of the execution environment our system is to be employed in. This configuration is based on several parameters:

- *Sketch Variation*: Different sketch variations cause a different impact to the accuracy of flow identification since they allow the different flows to collide in different ways. Our datasets are retrieved from the results of applying the sketch variation compression to a non-compressed set of flows.

- *Available Memory*: The memory available to the sketch is dictates the total number of counters available to the sketch, therefore directly impacting the final accuracy.

- *Sketch Size*: Altering the width and length of the sketch (or sub-sketches) will alter the error margins according to the CM sketch [4] specifications.

- *Maximum Flow Number*: The number of concurrent flows can severely impact the accuracy of the classification since a too great number of different flows colliding on the same counters can lead to misleading factors towards the classification. For the CM-PF this will lead to a greater number of sub-sketches thus reducing their size and possibly compromising the distributions of their flows. The maximum number of flows can be enforced by the number of entries in the flow table.

- *Regular-Covert Flow Ratio*: Although this is of no significance to the CM-PF variation, both the CM-S and CM-PB variations can be severely impacted by this ratio since both covert flows and regular flows may collide with each other in these variations. In the cases where multiple flows of a single type collide with very few of another type, the latter may be incorrectly identified as the former. This is due to the more common type of flow being a greater influence on the final mixed result leading to it resembling those flows.

- *Packets per Flow*: This can also be interpreted as the amount of time between each retrieval of metrics. The more packets per flow we take in the more descriptive the packet length distribution becomes. However, similar to the previous item, in cases of multi-type collisions it may lead to misclassifications.

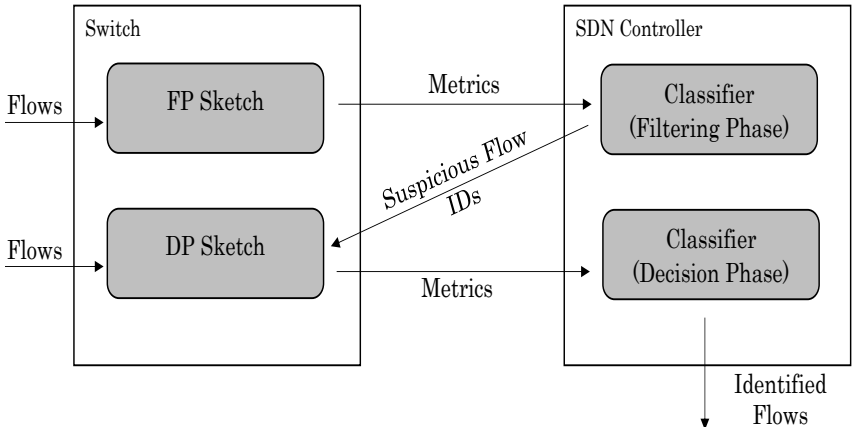


Figure 3.4: System architecture.

- *Bucket Range*: The range of values of each bucket impacts all sketch variations. Higher bucket ranges will lead to less overall buckets to maintain while providing less detailed distributions and vice-versa. For the CM-PB, a greater number of buckets will lead to a greater number of sub-sketches thus reducing their size and possibly compromising the distributions obtained.

Datasets with the the packet size distributions that take into account these parameters should be generated and used to train the classifier before the system begins its execution. Additionally, generating testing datasets using the same parameter configuration can be used to test the accuracy of the classifier until acceptable results can be achieved. This way the classification algorithm may begin accurately identifying covert channels as soon as they are retrieved from the switch.

3.4.2 Classification Algorithm

The classification algorithm allow us to sort the received flows into the two classes: Regular skype flows and Covert channel flows. It does so by taking into consideration the different statistics of each type of flow across several instances present in the training datasets. our case these statistics consist of the several buckets that make up the packet length distributions. Of course, this means that altering the number of buckets will provide us with with varying numbers of datapoints. By reducing the number of datapoints we can greatly reduce the error introduced by the sketches since the number of colliding items is reduced as well.

3.5 2-Phase Architecture

In addition to the previous architectural design, we propose an extension to it in the shape of a two-phase architecture, both working in parallel, as seen in Figure 3.4. It has a greater focus on limiting the number

of misclassified flows that do not contain covert channels. It uses two flow tables, two sketch variations and two classifier configurations in sequence in order to identify flows carrying covert channels. The first phase is called the *Filter Phase* and the sketch belonging to it is called *FP Sketch*. The second phase is called the *Decision Phase* and the sketch belonging to it is called the *DP Sketch*. Later, in the evaluation section, we will assess the effectiveness of this architecture against that of the single-phase architecture and what type of sketch is best suited for each phase.

In the two-phase architecture, certain suspicious flows can be processed by both phases. New flows are first inserted into the flow table associated with the FP Sketch and start incrementing counters in the sketch itself depending on the functioning of the variation. The purpose of this phase is to obtain a preliminary (and possibly more inaccurate than when using the single-phase architecture) estimate of the packet length distribution of each flow. Additionally, similarly to the regular architecture, the content of this sketch will be periodically retrieved by the controller in order to be classified. However, after the classification algorithm identifies the class of each flow, the regular Skype flows are discarded while the ones identified as containing a covert channel are kept. These flows are considered suspect and will be inserted into the flow table of the DP Sketch. This is performed during what was previously the "reset" operation and while the flow table of the FP Sketch will be completely reset, the one from the DP Sketch simply has its contents overwritten. Both sketch variations are still reset.

For the Decision Phase, the flow table associated with its sketch does not function the same way as its original incarnation. Instead, upon the reception of a packet belonging to a flow present in the flow table, the DP will increment its corresponding counters in its sketch. If the flow is not present in this flow table it may still be inserted in or matched to the flow table corresponding to the FP Sketch and increment the counters in the FP Sketch according to its original behaviour.

The purpose of the DP Sketch is to obtain a second estimation (usually more accurate than the previous phases' since it only has to account for the previously classified suspicious flows) of the reintroduced flows, thus providing the system with two layers of filtering out flows that the classifiers does not consider suspicious.

Seeing how regular Skype flows are more common than ones containing covert channels, the Filtering Phase contributes to softening the initially biased ratio between the two before performing a precise classification with the Decision Phase. Hence since the number of flows measured in the Decision Phase is much smaller than in the Filtering phase we can obtain much more precise metrics for the suspicious flows which can be used to better identify covert channels.

Finally, since both phases will have to handle significantly different number of flows the memory available will not be equally divided between the two sketches. Instead, a larger part of the available memory is allocated towards the FP Sketch and the remainder to the DP Sketch. Additionally, due to this separation of the available memory it is also likely that the classifiers for each phase use different

```
1       parser MyParser(packet_in packet, out headers hdr, inout metadata meta,
            inout standard_metadata_t standard_metadata) {
2
3       state start {
4           transition parse_ethernet;
5       }
6       state parse_ethernet {
7           packet.extract(hdr.ethernet);
8           transition select(hdr.ethernet.etherType) {
9               TYPE_IPV4: parse_ipv4;
10              default: accept;
11          }
12      }
13      state parse_ipv4 {
14          packet.extract(hdr.ipv4);
15          transition accept;
16      }
17  }
```

Listing 3.1: P4 code for the parsing phase

configurations when creating the training datasets. Primarily, they differ in term of maximum flow number and available memory, although some parameters such as sketch variation used may also change.

3.6 P4 Implementation

The first implementation is one we performed on a P4 Simulator [38] available through a virtual machine made by the P4 community and complete with several tutorials. It makes use of P4C (P4 compiler) to compile the P4 programs created and BMv2 (Behavioral Model v2) which provides a simulated environment to run a P4 software switch, standalone or with mininet, in order to simulate the transfer of packets through different networks devices. The purpose of this implementation is to provide us with realistic implementation of how one of our CM sketch variations, as seen in the previous section could be programmed into a switch. Specifically, we emulate the behaviour of the CM-PF sketch given that, we will see in the next in section 4, it consistently provided the best results.

To start we take a look at the different phases that make up the switch's behaviour, taking care to highlight important details and ending with a discussion of the implementation's limitations.

3.6.1 Parsing

The purpose of this first phase is to parse each of the headers that make up the incoming packets. Since, for simplicity's sake, we will use only the source IP address of the packet we also only parse the packets up to the IPv4 header, as seen in the code excerpt 3.1. Omitted, however is the definition of each of these headers as data structures, each defining every field of the header and their sizes in bits. If we were to use the TCP/IP 5-tuple, we would need to define and parse the extra headers.

```
1       register<bit<32>>(flowNum*flowCounters) sketchRow1;
2       register<bit<32>>(flowNum*flowCounters) sketchRow2;
3       register<bit<32>>(flowNum*flowCounters) sketchRow3;
4       register<bit<32>>(flowNum) hashmap;
```

Listing 3.2: P4 code for the data structures

After parsing the necessary information from the packets, we have to extract and transform the necessary fields in order to successfully increment the correct sketch counters.

3.6.2 Sketch

In Listing 3.2, we show the code fragment responsible for the implementation of the data structures that comprise the sketch itself. Each counter is actually what is called a “register” in P4. We use three arrays, called ´´sketchRow” 1 through 3 (since in this case we are emulating a sketch with three hash functions), of registers to represent the CM-PF counters (and of other variations). To avoid constructing multiple sets of arrays for each flow, each array has enough entries to account for the total number of flows times the number of counters per sub-sketch row. The “hashtable” array performs two functions. It both performs the duties of the flow table (explained in section 3) and specifies the sub-sketch identifier the flow is assigned to. As such, in order to increment the correct counters we must:

1. Obtain the number of the sub-sketch the flow is assigned to.
2. Multiply it by the total number of counters per sub-sketch row.
3. Add the position of the counter.
4. Repeat for all rows. This is cannot be done with a loop in P4 and should be done explicitly for each row.

3.6.3 Processing

The operation of the program can be visualized in Listing 3.3 and progresses as follows: If the IPv4 header is valid, we perform an hashing operation over the source IP address in order to obtain the the entry in the flow table the flow is assigned. This is performed using the “hash” method call that takes as parameters:

1. The variable that stores the result of the hashing operation.
2. The algorithm used.
3. The minimum value (always 0 in our case).

```
if (hdr.ipv4.isValid()) {
    hash(pos, HashAlgorithm.crc32, base, { hdr.ipv4.srcAddr }, flowNum);
    hashmap.read(meta.hashValue, pos);

    if (meta.hashValue == 0) {
        meta.hashValue = hdr.ipv4.srcAddr;
        hashmap.write(pos, meta.hashValue);
    }

    if (meta.hashValue - hdr.ipv4.srcAddr == 0) {
        hash(position, HashAlgorithm.crc32, base, { standard_metadata.
            packet_length, seed1}, flowCounters);
        position = position + (flowCounters * pos);
        sketchRow1.read(value, position);
        sketchRow1.write(position, value+1);

        hash(position, HashAlgorithm.crc32, base, { standard_metadata.
            packet_length, seed2 }, flowCounters);
        position = position + (flowCounters * pos);
        sketchRow2.read(value, position);
        sketchRow2.write(position, value+1);

        hash(position, HashAlgorithm.crc32, base, { standard_metadata.
            packet_length, seed3 }, flowCounters);
        position = position + (flowCounters * pos);
        sketchRow3.read(value, position);
        sketchRow3.write(position, value+1);
    }
```

Listing 3.3: P4 code for the counting operations of our architecture

4. The values to hash.

5. The maximum value. This value is optional and effectively results in a modulo operation being used with the original value and provided maximum.

Next, we read the “hashmap” at the position obtained. If the value is empty, we write the address to it and proceed. Otherwise, we must check if it is equal to the current address. If not, no counters are incremented.

Finally, we can increment the counters. The process is the same for all rows. We start by performing a hashing operation over the packet size and a seed (different for each row). We take this value and add it to the value obtained by multiplying the number of counters per row in each sub-sketch and the number of the corresponding sub-sketch. With this, we have the position of the counter to increment. Once the processing is complete, the packet headers are re-emitted to the packet and it can be forwarded.

3.6.4 Limitations

While operating the simulator we came across a few limitations that are not present in the specification of the real P4 language. The first of these is that certain operations like division and modulo could not be performed with the P4 program and, as such, we were unable to perform the division into buckets

previously mentioned. This could be overcome using other operations and loops, however, loops can not be performed with the P4 language itself by design. Instead, we could add our own hash algorithm at the back-end C++ code of P4. Since the simulator did not possess the source code (only the compiled executionables), we couldn't add our own operation to perform division for us. The second biggest limitation relates to the controller portion of the simulator. Although, this implementation concerns mainly the switch's implementation, a simple python program using the P4Runtime API to communicate with the switch in order for us to read the sketch counters was also created. However, we soon found out that it was impossible for us to read the entirety of the counter arrays all at once. Instead we would have to request a single counter at a time. Thankfully however, these limitations are restricted only to the simulator and not the actual P4 language.

Summary

This chapter addressed the design of our architecture and showcased a software implementation of it in a P4 simulator. The use of SDN technologies allow for communication between the switch, which is responsible for performing our measurements, and the controller, which makes use of a classification algorithm to identify the covert channels. Sketches allow the metrics to be stored in a memory-efficient manner, capable of measuring a high amount of flows within the limited memory of switches. And finally, the P4 language allows us to program the functionality of our architecture into the switch in order to process the packets of interest and collect the metrics for our measurements. The next chapter presents an experimental evaluation of our proposed architecture in a different manner of configurations.

4

Evaluation

Contents

This chapter describes the evaluation of our system. It highlights our architectures ability to perform flow distinction between those that contain covert channels and those that do not. In addition it highlights the ability of an architecture that contains two filtering phases to further reduce the number of misclassified flows that do not contain covert channels. Our evaluation is mainly concerned with two dimensions: the number of flows that can be characterized and the accuracy of the differentiation associated with these flows. Section 4.1 describes the implementation that was used for the experimental evaluation. Section 4.2 describes the configuration of the experiments, including the metrics and data sets used. Section 4.3 presents the results obtained for each sketch variation described in the previous chapter. Section 4.4 evaluates how increasing the number of hash functions employed by our sketches affects the precision of our system. In section 4.5 we evaluate the success of the covert channel identification by using a combination of sketches for this purpose. Section 4.6 discusses how the variation of available memory affects the precision of our system.

4.1 Implemented Simulator for the Experimental Evaluation

The implementation described in the previous chapter, was performed for us to validate whether or not it was possible to implement our architecture on a P4 environment. For the purpose of performing the evaluation of our proposed data structures, we have designed a simulator that simplifies the process of obtaining the metrics and classifying the flows after being processed by our sketch variations. The purpose of this variant is to simulate efficiently the execution of the system, using packet traces as input.

The first step in our simulator is to obtain a easily-readable representation of the packets of a given flow. We began with a few "*.pcap*" files obtained from previous work [5]. The "*.pcap*" files are parsed and an intermediate output file is produced where, for each packet, a tuple containing only the information relevant for our simulations in included. We represented each packet by its flow identifier and "bucket" corresponding to its original size. Since multiple files recorded packets that used the same IPs, a flow is uniquely identified using the name of the file the packet came from and the respective IP.

Once the list of all packets is obtained, we split the included flows into regular Skype flows and covert ones and then generate a new list containing packets from flows of both types following certain ratios. The most common of these ratios are 0.95 regular Skype flows and 0.05 covert flows. To account for our two-phase architecture, we divided the packets of this new list into two output files, each one to be used in each of the different phases.

Next, these lists of packets are used to feed a program written in C++ that enabled us to simulate the behaviour of each of our different sketch variations and generate the compressed quantized packet length distributions. This program can simulate the behaviour of each of the sketch variations at the same time, all with different configurations regarding the number of maximum flows, the dimension of

the sketch and number of hash functions.

From here, the resulting packet size distributions are processed by our classification algorithm (using the code provided in the previously mentioned study [5]). The algorithm outputs two types of information. First, the accuracy of the classification using the metrics used in prior work [5]. And second, a set of the flows the classifier identified as containing a covert channel. Using this set and the second list containing the second half of the packets after enforcing the regular-to-covert ratio we can simulate the behaviour of the second phase of the two-phase architecture. By restricting the program that simulated the sketch variation operation to this subset of flows, we can obtain new compressed quantized packet length distributions using configurations more appropriate to the Decision Phase, distributions which can once again be classified by the classification algorithm to obtain the final results of the two-phase architecture.

4.2 Configuration of the Experiments

4.2.1 Sketch Configuration

We simulate the use of a programmable switch using sketches for the collection of a representation of packet size distribution for a limited number of flows. Each sketch uses three hash functions that correspond to the application of a single hash function initialized with a distinct seed value.

The useful memory of the switch will be set to the total size of 0.3 MB, which was chosen chosen taking into account the theoretical basis underlying the design of the CM sketch. Measuring 1000 flows of approximately 3000 packets each while using 0.3 MB of memory provide us with certain guarantees. With this amount of memory it is guaranteed with 95% probability that the value of the count read from each register, in a given bucket of the CM sketch, incurs an error of less than 10% of the total package of the true distribution of a flow.

To match the configuration in previous work [5], the packet sizes of each flow are quantified in buckets at the granularity $k = 5$ bytes. Considering the possible existence of packets with a size ranging from 0 to 1500 bytes, we use a total of 300 buckets. Additionally, we use the source IP (randomly assigned) to assign an identifier to each flow.

4.2.2 Dataset

Our experiences contemplate the transmission of multimedia flows in a proportion of 95% of legitimate Skype flows and, respectively, 5% of Skype flows carrying a covert channel. This ratio captures the simulation of a real workload where it is expected that the vast majority of the observed flows in the network do not act as vehicle of a covert channel. The flows that carry a covert channel were produced

by the Facet [7] and DeltaShaper [8] tools, which replace a region of the video frames produced in legitimate video calls with some content to be transmitted in the covert channel, e.g., a Youtube video or a stream of IP packets. Each flow has a total duration of 60 seconds. All flows were obtained through contact with the authors of the study mentioned above [5] and number at 1000 Facet flows and 300 flows for two configurations of DeltaShaper. Since all of the available flows have an approximate duration of one minute we split them into segments of 30 seconds, one for each phase of the 2-phase architecture. Using the same 30 second segment during the single phase architecture allow us to compare the two architectures.

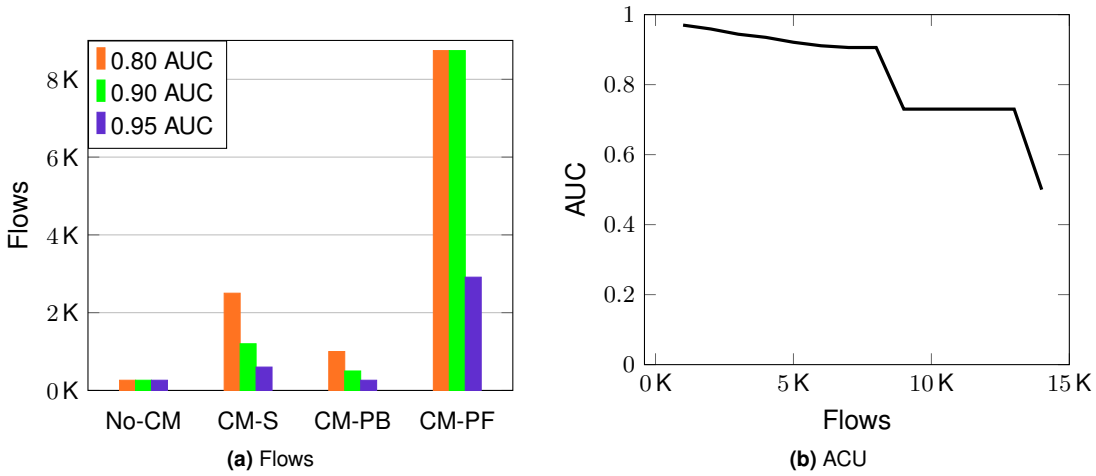
4.2.3 Metrics

Previous works [5, 7, 8] studying the detection of covert channels on the Internet use the Receiver Operating Characteristics Area Under Curve (ROC AUC) [39] metric to measure the success achieved in the differentiation of flows. Briefly, AUC summarizes the relationship between a classifier's true positive and false positive rates. In our case, the true positives would be the amount of Skype flows identified as such and the false positives would be the same for the covert channel flows. In the context of this work, a classifier with the ability to make a random guess about the class of a flow displays an AUC = 0.5, while a perfect classifier is characterized by an AUC = 1. We use the same metric to evaluate the accuracy of the different types of sketches. Additionally, we use the supervised classification algorithm *XGBoost*, as proposed by Barradas et al. [5]. The classifier is trained using a set of flows reserved for training, collected in rounds. In each round, we select a flow sub-sampling at 95%/5%. Then an approximate representation of the distribution of these flows is obtained through the sketch under analysis. The classifier is trained with a set of balanced samples (with the same amount of Skype and covert channel flows) obtained from the representations of the distributions generated by the sketch. This process is repeated for 200 rounds.

4.3 Single Sketch Evaluation

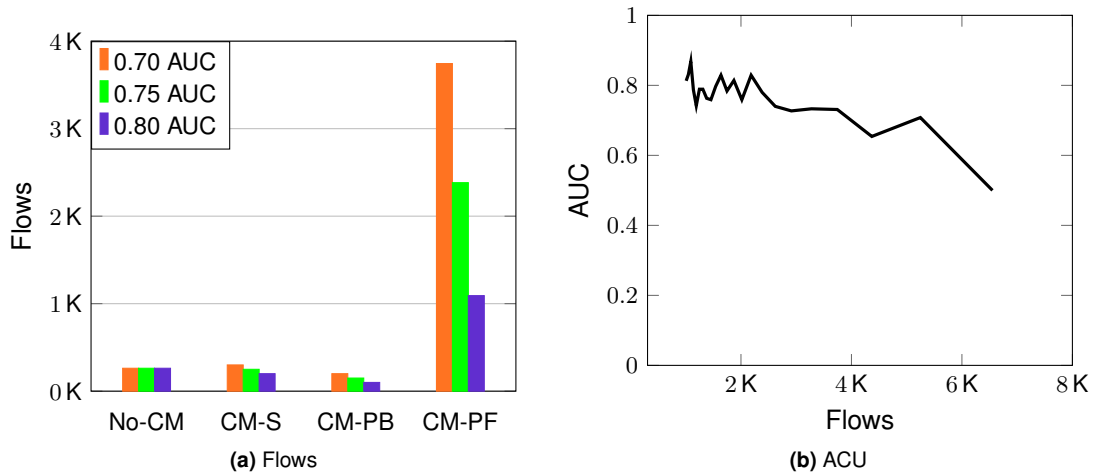
4.3.1 Setup

We begin by comparing the performance of the different variants of the CM sketch to a solution that does not resort to sketches. All variations use all available memory. In this case, the first 30s of traffic corresponding to each flow were collected by the sketch with 0.3MB of memory available. In this experiment, we intend to understand the number of flows that can be analyzed by the different types of sketches while obtaining a classification accuracy above three different values of AUC appropriate for each system.



(a) Flows **(b)** ACU

Figure 4.1: Performance of CM sketch for Facet.



(a) Flows **(b)** ACU

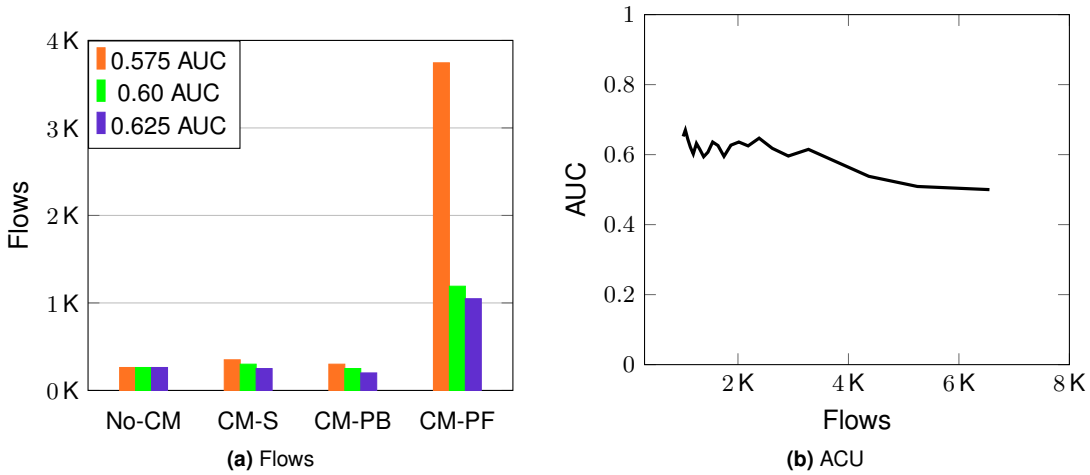
Figure 4.2: Performance of the CM sketch for DeltaShaper 320.

4.3.2 Results

The results are shown in Figures 4.1, 4.2, and 4.3. In the graphs on the left we compare all the different sketches with each other while in the graphs on the right we track the AUC values for the best performing sketch variation as we increase the amount of flows being tracked.

All systems show somewhat similar results in Figures 4.1a, 4.2a and 4.3a. In all three, CM-PF provides the best performance, showing significantly better results than the other variants, CM-S appears to be the next best variation closely followed by CM-PB, both of which often show results similar to No-CM in all figures.

Facet, as seen when compared with the other two systems, shows the best results, being able to support close to 3K flows concurrently at the highest AUC value we set for it (0.95). We believe this is due



(a) Flows **(b)** ACU

Figure 4.3: Performance of the CM sketch for DeltaShaper 160.

to Facet possessing the highest base accuracy (where the metrics are collected without compression) at approximately 0.99 AUC whereas DeltaShaper 320 possessed a base 0.86 AUC and DeltaShaper 160 possessed 0.66 AUC. Since employing sketches incurs some form of compression on the metrics the poorer the initial accuracy of the system, the poorer the results will be after they are compressed. In the absence of sketches, with the previously defined configuration, it is possible to measure a total of 262 concurrent flows. Since no sketch is used the AUC value is always equal to the base AUC of each system.

The CM-S sketch can measure more flows than simple registers for all AUC values, indicating that despite the various collisions between flows and buckets it is still possible to distinguish the two types of traffic with high accuracy. The CM-S sketch was able to successfully measure approximately 500 flows concurrently for the Facet system at highest AUC value, while for the other two it obtained results equal to that of the No-CM variation.

The CM-PB sketch shows values below CM-S, which is due to the fact that CM-PB does not make efficient use of the available memory, since the different buckets are not used uniformly by the different flows. This phenomenon can be seen in Figure 4.4, which illustrates the heat map for buckets 13 and 100. Each line corresponds to the application of a distinct hash function (as previously illustrated in Figure 3.3) and the memory division provides 87 registers for each hash function for each bucket. According to the heat map, this number of records is not sufficient to avoid a large number of collisions as the number of flows accounted for by the sketch is increased. It is then found that for the flows under analysis there are buckets with a high packet count while others have a low count. The redistribution of the space reserved for buckets rarely used to reduce the number of collisions in the remaining buckets is a time-consuming process that requires manual tuning of the sketch. The CM-PB sketch was able to successfully measure approximately 200 flows concurrently for the Facet system at highest AUC value,

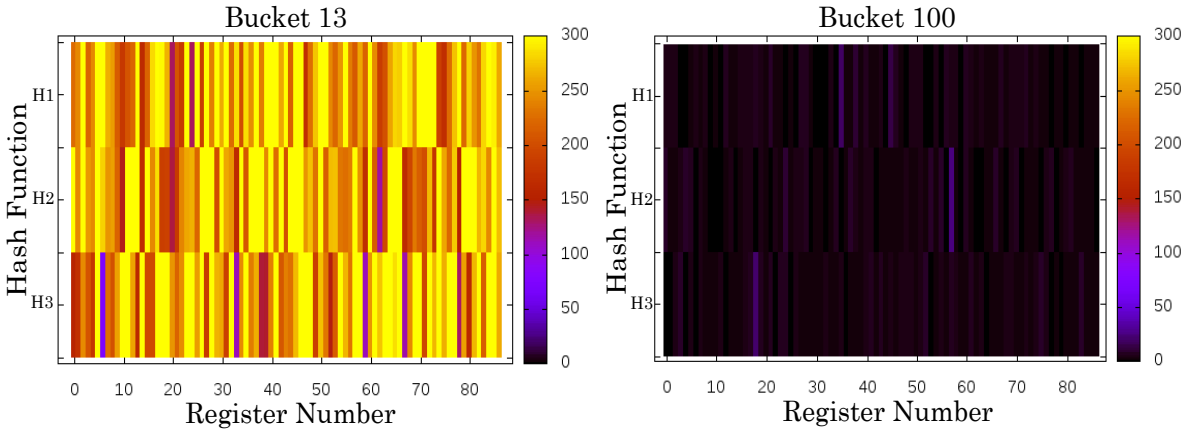


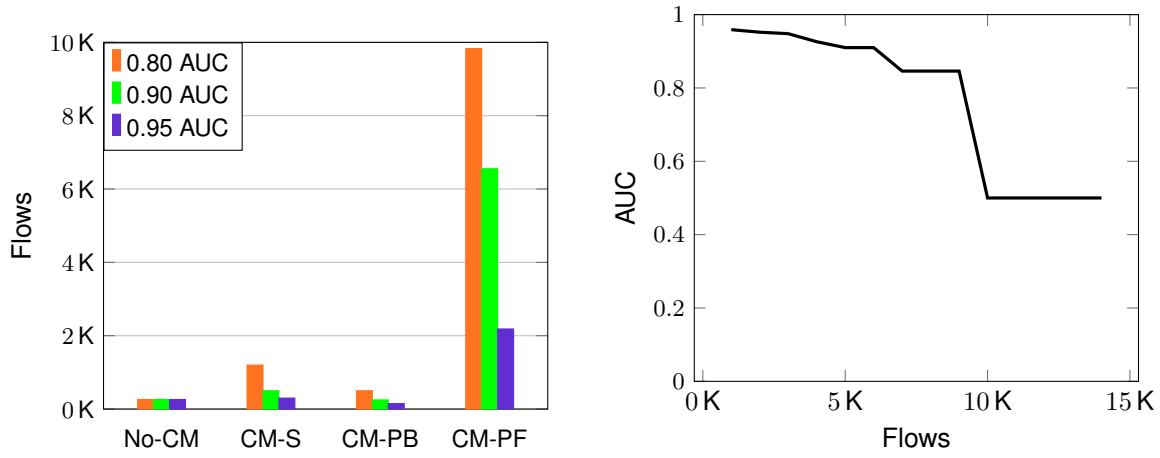
Figure 4.4: Heat Map for buckets 13 and 100 of the CM-PB sketch.

while for the other two it obtained results equal to that of the No-CM variation.

Finally, the CM-PF sketch gets the best results. This is due to the fact that each flow is kept separate from all others, preventing the collision between legitimate Skype flows and flows carrying a covert channel. Additionally, the two types of flows are easily distinguishable based on the packet size distribution, and these differences remain visible even when several buckets collide in the same register. It should be noted that CM-PF supports the same number of flows for AUC 0.8 and 0.9 as the compression at the register's level is not significant. The CM-PF sketch was able to successfully measure approximately 3K flows concurrently for the Facet system at highest AUC value, while for the other two it was able to measure approximately 1K flows.

For the graphs 4.1b, 4.2b and 4.3b we can see how as the packet length distributions are further compressed when using the CM-PF sketch, the accuracy decreases accordingly. By using CM-PF to measure large quantities of flows it is necessary to reduce the size of each CM sketch, leading to that, from a certain number of flows, the number of collisions between distinct buckets leads to loss of ability to distinguish traffic.

The results shown in Figure 4.1b show that the CM-PF sketch reaches an AUC of between 0.9 and 1 for a number of flows of less than 8K, falling to 0.7 at 13K flows. At this point, a sharp decrease in AUC is observed, approaching the value 0.5, and therefore reducing the classification decision to a random guess for a number of flows greater than 14K. For both the DeltaShaper graphs, the expected decrease is not as linear as for the Facet system. In accordance with our previous observation, of inferior base AUC's leading to decreased accuracy when performing compression, so too does the progress of the AUC become more unpredictable as we add more flows. In short, the figure shows that as the size of each CM sketch decreases, each counter that composes it will be shared by a greater number of buckets. In this way, the values corresponding to reading these buckets will tend to be shared. Of course, the more buckets share the same values, the greater the degree of overlap of approximations of



(a) Comparing all sketches.

(b) CM-PF progression.

Figure 4.5: Analysis capacity of flows for each variation of the CM sketch for the Facet system using 4 hash functions.

distributions of different flows. When the system in question possesses a high base AUC, the decrease in the AUC as more flows are added seems almost linear. Something which becomes less possible as the base AUC decreases.

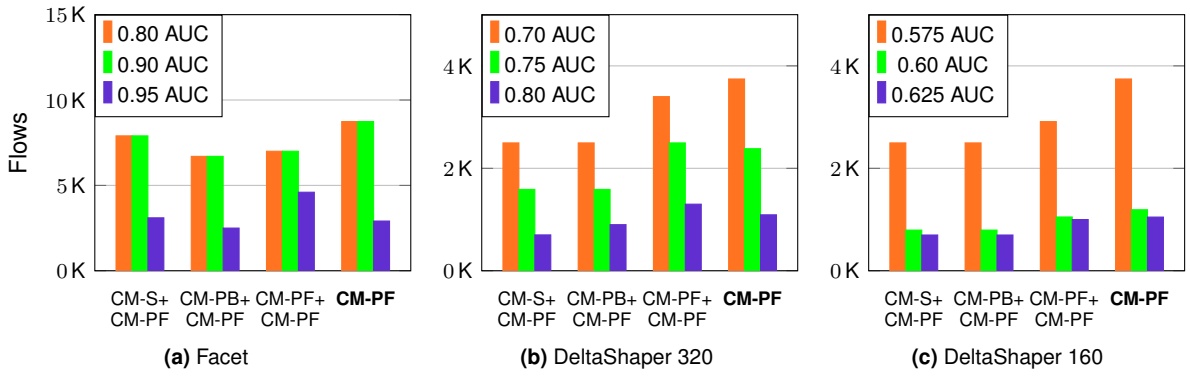
4.4 Hash Function Variation

4.4.1 Setup

Next, we seek to investigate how the addition of an extra hash function impacts the accuracy of the classifier while maintaining all other parameters of the previous configuration. We utilize the Facet test bed as it showcased the best results in the previous tests and possesses the highest base accuracy of all three systems. By increasing the number of hash functions we increase the likelihood of the query being within a certain error margin from approximately 95% to 98% while increasing our error by approximately 25%.

4.4.2 Results

While the results from the graphs in Figure 4.5 seem similar to the previous a closer inspection shows that the addition of a fourth hash function was unable to produce superior results to the previous test's, in some cases there was a decrease of almost 50% of flows measured concurrently. An increase of the number of hash functions results in a decrease in the number of counters per sketch row which results in more collisions. This increase in the number of collisions leads to an increase in the expected error



(a) Facet **(b)** DeltaShaper 320 **(c)** DeltaShaper 160

Figure 4.6: Analysis capacity of flows using the two-phase architecture.

margin for each item read by approximately 25%, resulting in metrics further compressed than seen in previous tests.

From graph 4.5a, when compared with 4.1a, we can observe a decrease in the number of flows that can be concurrently measured for both the CM-S and CM-PB sketches. For the CM-PF sketch the same can be seen for the AUC values of 0.90 and 0.95. While 0.80 AUC does show an increase, this is due to it being an extreme case where neither sketch could be pushed anymore allowing the 4 hash functions case to surpass the 3 hash functions case due only to having just enough counters to provide a decent accuracy score. In this case, adding any further compression would have left the system with the probability of correctly identifying the type of flow being classified is similar to that of a coin flip, therefore what we see at 0.80 AUC is an outlier that does not fully represent the impact of the additional hash function. Graph 4.5b also shows a much more rapid drop in accuracy than with only 3-hash functions and an overall decrease in the amount of measured flows.

Overall, it seems that for our purposes increasing the number of hash functions does not provide us with any benefits. The increase of the likelihood of the query being within a certain error margin does not compensate for the added error due to having less counters per hash function.

4.5 Evaluation of the Two-Phase Architecture

4.5.1 Setup

In this section we assess whether the proposed architecture, which is based on the use of two sketches in sequence, offers advantages in relation to the use of a single sketch. Instead of experimenting with all possible combinations of different sketches, we chose to always use the sketch that presented the best results in the decision phase (i.e. using CM-PF as an DP sketch) and to vary only the sketch used in the filtering phase. To simulate the differentiation of flows in real time, the filtering phase is fed with

the first 30s of traffic of each stream, while the decision phase is fed with the remaining 30s (making up the total duration of each flow in our data set).

4.5.2 Results

In Figure 4.6 we present the results for a configuration where 0.2 MB are reserved for the FP sketch and 0.1 MB are used for the DP sketch for all 3 systems. Note that in these tests only the DP sketch maintains the indicated AUC; the first phase may admit a lower AUC as the flows selected for the second phase will be re-analyzed.

Taking as an example the 9K flows Facet can collect for AUC values of 0.80 and 0.90 using 0.3MB as seen in Figure 4.1a, the two-phase architecture should, therefore, be able to classify 3K flows when using 0.1 MB in the decision phase. In this way, to be competitive, the architecture in two phases will have to filter more than 2/3 of the flows in the filtering phase. The values of Figure 4.6 show that this is not always accomplished. Specifically, for lower AUCs, like 0.80 and 0.90, the two-phase architecture cannot classify the same number of flows as seen previously using a single CM-PF sketch.

On the other hand, when aiming for 0.95 AUC values, the two-layer architecture can offer tangible advantages. In fact, using a combination of the two-layer CM-PF sketch it is possible to increase the number of flows that can be monitored from 3K to about 4.5K, which represents an increase of about 50% for the Facet system. Seeing as the original 3K flows for this AUC value is much lower than the other values, the 4.5K flows seen here is a sufficiently low amount that the filtering phase can successfully filter out a great majority of the flows.

The only tools that show increases over the single phase architecture are the Facet and DeltaShaper 320. This relates once more to the base accuracy of each system. Since the accuracy of DeltaShaper 320 and 160 are inferior to Facet's then the number of flows that are filtered out by the filtering phase are lower than Facet's as well. This means that a higher percentage of flows are considered suspicious and are passed on to the decision phase. Since the number of flows that can be measured concurrently in the DP sketch is limited, the initial number of flows fed to the filtering phase must also be lower in order to account for this limit. As such, DeltaShaper 320 (which has a base AUC of 0.86, lower than Facet's 0.99) does not show an increase in flows of the same proportion as Facet's. Additionally, DeltaShaper 160 (which has a base AUC of 0.66) does not show any improvements at all.

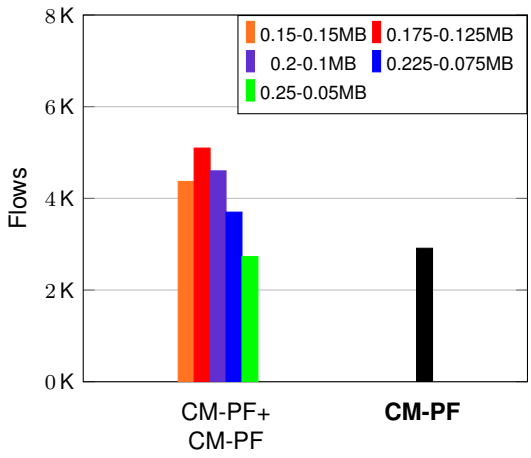


Figure 4.7: Analysis capacity of flows using varied memory divisions between the two phases of the two-phase architecture.

4.6 Varying the Filter's Memory

4.6.1 Setup

In this section, we analyze how the available memory for the different phases of analysis affects the ability that the system displays in the differentiation of flows. More concretely, we compare different configurations of the architecture in two phases, varying the proportion of memory reserved for each of the sketches, namely by fixing the available memory for FP, between 0.15 MB and 0.25 MB, in successive increases of 25KB. In this experience, we use the best combination of previously identified sketches (CM-PF + CM-PF) and best performing system (Facet).

4.6.2 Results

As can be seen from Figure 4.7, different proportions show different results, as the memory reserved for the DP needs to be sufficient to accurately classify all flows that are not filtered in FP. Therefore there is a balance between the filtering capacity of the first stage and the accuracy of the second stage. For the study settings, the proportion that offers the best results is to reserve 0.175 MB for FP and 0.125 MB for DP. With this configuration it is possible to monitor about 5k flows, a gain of about 66% relative to the use of a single sketch.

Summary

This chapter details the experimental evaluation of our architecture, including tests conducted in order to ascertain whether the metrics gathered by our system still allow for accurate covert channel identification

and how the configuration of different parameters affect that accuracy. In the course of these tests the accuracy obtained in classifying the vast number of flows analyzed showcased that our system can indeed provide us with a high degree of precision, even with the error introduced by the sketches. Additionally, the use of sketches allowed us to measure a much higher amount of concurrent flows than we would otherwise. The next chapter will conclude the thesis with a brief discussion of our architecture and findings, as well as some ideas for future work.

5

Conclusion

Contents

5.1 Conclusions

For years, network traffic measurement has provided network analysts and technicians with the tools and technology to efficiently maintain and optimize their networks through several measurement tasks like the ones discussed in Callado et al. [19] and Mohan et al. [20] surveys. The introduction of modern networking advancements, specifically SDNs, programmable switches and P4, have allowed for never before seen levels of efficiency and accuracy through simple to implement, and extend, mechanisms.

While several works and techniques have been introduced to take advantage of these advancements, some of the limitations of older tools are still present in these implementations, most often suffering from storage space limitations and having to resort to alternative solution that require sampling or additional hardware. And even then, those that have surpassed those limitations and drawbacks still do not provide the networks with measurements that possess the level of detail, accuracy and efficiency that can be drawn out from the right combination of SDNs, programmable switches, P4 and counting sketches, which are key for overcoming the previously mentioned limitation.

In this work we studied the possibility of classifying covert channels in real time and efficiently, taking advantage of the use of software-defined networks and P4 to capture an approximate representation of the distribution of the flows to be monitored at a programmable switch. In order to capture an approximation of the distribution efficiently, we use one of several variations of the CM sketch [4]. We also presented a possible P4 switch implementation of this process in a simulator using one of our sketch variations. The server can then retrieve this representation to be classified. In addition, we propose a two-layer filter architecture, where in a first phase a sketch allows for filtering a significant fraction of the flows, the remaining flows being classified using a sketch configured to obtain better accuracy.

We performed an extensive evaluation on our experimental simulator, testing several different parameter configurations using flows altered by three different systems. From this evaluation, we perceived how classification is impacted by the different parameters. In some tests, our architecture was able to measure more than 20 times the number of concurrent flows than when in the absence of sketches. Additionally, our two-phase architecture shows gains of more than 66% from the use of a single sketch, being able to monitor about 5K flows simultaneously and offering an AUC of 0.95 in the identification of covert channels, using for this purpose only 0.3 MB of memory in the switch.

5.2 Future Work

For future work, it would be interesting to take up different use cases where our architecture could be of use, like website fingerprinting. Additionally, a more in-depth investigation on the behaviour of our sketch variations could provide a better understanding of their behaviour and therefore allow for better prediction of results and optimized configurations.

Bibliography

[1] SDN, https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf, accessed: 2019-10-26.

[2] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," *NSDI '13 Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)*, pp. 29–42, April 2013.

[3] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *ICALP '02 Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pp. 693–703, July 2002.

[4] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, April 2005.

[5] D. Barradas, N. Santos, and L. Rodrigues, "Effective detection of multimedia protocol tunneling using machine learning," *Proceedings of the 27th USENIX Security Symposium*, pp. 169–185, August 2018.

[6] J. Hayes and G. Danezis, "k-fingerprinting: A robust scalable website fingerprinting technique," in *25th USENIX Security Symposium*, Austin, Texas, USA, August 2016, pp. 1187–1203.

[7] S. Li, M. Schliep, and N. Hopper, "Facet: Streaming over videoconferencing for censorship circumvention," in *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, Scottsdale, AZ, USA, 2014, pp. 163–172.

[8] D. Barradas, N. Santos, and L. Rodrigues, "Deltashaper: Enabling unobservable censorship-resistant tcp tunneling over videoconferencing streams," in *Proceedings on Privacy Enhancing Technologies*, vol. 2017(4), Minneapolis, MN, USA, 2017, pp. 5–22.

[9] Wireshark, https://www.wireshark.org/docs/wsug_html_chunked/, accessed: 2019-10-26.

[10] SNMP, http://www.net-snmp.org/docs/readmefiles.html, accessed: 2019-10-26.

[11] Netflow, https://www.ietf.org/rfc/rfc3954.txt, accessed: 2019-10-26.

[12] Ping and Traceroute, https://www.cisco.com/c/en/us/support/docs/ios-nx-os-software/ios-software-releases-121-mainline/12778-ping-traceroute.pdf, accessed: 2019-10-26.

[13] Barefoot, https://www.barefootnetworks.com/products/brief-tofino/, accessed: 2019-10-26.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, July 2014.

[15] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[16] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications," *IMC '04 Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pp. 101–114, October 2004.

[17] G. Carl, G. Kesidis, R. R. Brooks, and S. Rai, "Denial-of-service attack-detection techniques," *IEEE Internet Computing*, vol. 10, no. 1, pp. 82–89, January 2006.

[18] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, January 2013.

[19] A. Callado, C. Kamienski, G. Szabó, B. P. Gero, J. Kelner, S. Fernandes, and D. Sadok, "A survey on internet traffic identification," *IEEE Communications Surveys & Tutorials*, vol. 11, no. 3, pp. 37–52, August 2009.

[20] V. Mohan, Y. R. J. Reddy, and K. Kalpana, "Active and passive network measurements: A survey," *International Journal of Computer Science and Information Technologies*, vol. 2, no. 4, pp. 1372–1385, 2011.

[21] Fluke, https://assets.tequipment.net/assets/1/26/Fluke_Tap_Solution_Network_Monitoring_and_Analysis_Techniques_Using_Taps_and_SPAN_Switches.pdf, accessed: 2019-10-26.

[22] N. K. Ahmed, J. Neville, and R. Kompella, "Network sampling: From static to streaming graphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 8, no. 2, pp. 7:1–7:56, June 2014.

[23] M. Zhang, M. Dusi, W. John, and C. Chen, "Analysis of udp traffic usage on internet backbone links," in *2009 Ninth Annual International Symposium on Applications and the Internet*, Bellevue, Washington, USA, July 2009.

[24] W. John and S. Tafvelin, “Analysis of internet backbone traffic and header anomalies observed,” *IMC ’07 Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pp. 111–116, October 2007.

[25] D. Murray and T. Koziniec, “The state of enterprise network traffic in 2012,” in *2012 18th Asia-Pacific Conference on Communications (APCC)*, Jeju Island, South Korea, October 2012.

[26] S. González-Bailón, N. Wang, A. Rivero, J. Borge-Holthoefer, and Y. Moreno, “Assessing the bias in samples of large online networks,” *Social Networks*, vol. 38, pp. 16–27, 2014.

[27] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, “Opensample: A low-latency, sampling-based measurement platform for commodity sdn,” in *2014 IEEE 34th International Conference on Distributed Computing Systems*, Madrid, Spain, July 2014.

[28] sFlow, http://sflow.org/about/index.php, accessed: 2019-10-26.

[29] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, “Opennetmon: Network monitoring in openflow software-defined networks,” in *2014 IEEE Network Operations and Management Symposium (NOMS)*, Krakow, Poland, May 2014.

[30] S. Narayana, A. Sivaraman, V. Nathan, M. Alizadeh, D. Walker, J. Rexford, V. Jeyakumar, and C. Kim, “Hardware-software co-design for network performance measurement,” *HotNets ’16 Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 190–196, November 2016.

[31] N. Duffield, C. Lund, and M. Thorup, “Estimating flow distributions from sampled flow statistics,” *SIGCOMM ’03 Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 325–336, August 2003.

[32] Y. Li, R. Miao, C. Kim, and M. Yu, “Flowradar: A better netflow for data centers,” *NSDI ’16 Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI)*, pp. 311–324, March 2016.

[33] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, “One sketch to rule them all: Rethinking network flow monitoring with univmon,” *SIGCOMM ’16 Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 101–114, August 2016.

[34] A. Kumar, J. Xu, and J. Wang, “Space-code bloom filter for efficient per-flow traffic measurement,” *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2327–2339, November 2006.

[35] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger, “Network monitoring as a streaming analytics problem,” *HotNets ’16 Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 106–112, November 2016.

[36] N.-F. Huang, C.-C. Li, C.-H. Li, C.-C. Chen, C.-H. Chen, and I.-H. Hsu, “Application identification system for sdn qos based on machine learning and dns responses,” in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Seoul, South Korea, September 2017.

[37] S. Lee, J. Kim, S. Shin, P. Porras, and V. Yegneswaran, “Athena: A framework for scalable anomaly detection in software-defined networks,” in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Denver, Colorado, USA, June 2017.

[38] P. Simulator, https://github.com/p4lang/tutorials, accessed: 2019-10-26.

[39] T. Fawcett, “Roc graphs: Notes and practical considerations for data mining researchers,” *ReCALL*, vol. 31, pp. 1–38, 01 2004.