UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

SECURE NETWORK MONITORING USING PROGRAMMABLE DATA PLANES

Fábio Miguel Canto Pereira

Mestrado em Engenharia Informática
Especialização em Arquitetura, Sistemas e Redes de Computadores

Dissertação orientada por:
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves
e co-orientada por: Prof. Doutor Fernando Manuel Valente Ramos

2017

Acknowledgments

Throughout this year there were many people who helped me overcome the obstacles I faced. First of all, I would like to thank my family for giving me the opportunity to get where I am and for supporting me while I wrote this thesis and over all my life, in general.

I would also like to thank my friends and colleagues with whom I shared great moments over the year. They made my work days more pleasant and our break times refreshing. They were always there to motivate me and to discuss ideas when I got stuck.

I am also grateful to Faculdade de Ciências, specially its Informatics Department and the LaSIGE research group, for providing me all the conditions to perform my work.

A special acknowledgement for my advisors, Professor Nuno Fuentecilla Maia Ferreira Neves and Professor Fernando Manuel Valente Ramos, for giving me the opportunity to join this project, for their guidance, and availability over the year.

Funding

This work was partially supported by the European Commission through project FP7 SEGRID (607109) and project H2020 SUPERCLOUD (643964), and by national funds of Fundação para a Ciência e a Tecnologia (FCT) through project UID/CEC/00408/2013 (LaSIGE).

To my family and friends

Resumo

A monitorização é uma ferramenta fundamental na gestão das redes de computadores ao oferecer uma visão sobre o seu comportamento ao longo do tempo. Diferentes técnicas de monitorização têm sido aplicadas na prática, das quais se destacam duas: as baseadas em amostras e as baseadas em *sketches*. Enquanto as técnicas baseadas em amostras processam apenas um subconjunto do tráfego total (uma amostra), as técnicas baseadas em *sketches* processam todo o tráfego, procurando obter maior precisão nos seus resultados. Para poderem processar todo o tráfego e ainda assim serem escaláveis, os algoritmos baseados em *sketches* comprimem a informação monitorizada em estruturas de dados que têm comportamento semelhante ao das *hash tables*. Apesar da inevitável perda de informação resultante das colisões que ocorrem tipicamente quando se usam estas estruturas de dados, os algoritmos baseados em *sketches* apresentam ainda assim resultados bastante precisos, uma vez que todo o tráfego contribui para a computação das variáveis estatísticas monitorizadas.

A informação fornecida pelos algoritmos de monitorização é essencial para a correta operação da rede. No entanto, se o algoritmo de monitorização puder ser corrompido, os seus resultados deixarão de ser confiáveis, tornando a monitorização inútil. No pior caso, o administrador de sistemas não deteta que o algoritmo de monitorização foi comprometido e acaba por tomar decisões inadequadas, baseadas em informação incorreta. Este problema demonstra a utilidade de algoritmos de monitorização seguros. No entanto, não temos conhecimento de nenhuma proposta que vise a segurança dos algoritmos de monitorização. De facto, a generalidade dos algoritmos de monitorização ignora as questões de segurança de forma a minimizar os seus tempos de execução e a memória utilizada, o que se justifica pelas altas velocidades a que os pacotes têm de ser processados e transmitidos nas redes atuais.

O objetivo desta tese consiste no desenho, implementação e avaliação de um algoritmo de monitorização seguro e escalável. A base da nossa solução é o *Count-Min*, um algoritmo baseado em *sketches* que permite estimar a frequência de itens observados num dado *stream* de dados. Genericamente, o *Count-Min* utiliza uma matriz com duas dimensões, definidas antes do arranque do algoritmo (número de linhas e número de colunas), para armazenar os dados monitorizados. Além disso, necessita de uma função de

dispersão diferente por cada linha da matriz, responsável por mapear os itens processados pelo algoritmo numa coluna da matriz. Cada função de dispersão fica associada a uma linha da matriz e cada item vai ser processado por todas elas, sendo responsável pelo incremento de um contador em cada linha da matriz.

Para identificar possíveis vulnerabilidades de segurança na versão original do *Count-Min*, assumimos um adversário que poderá estar localizado em qualquer ponto da rede, mas que não tem acesso ao dispositivo em que o algoritmo está instalado. Verificámos que, para diferentes capacidades do adversário (escutar a rede apenas, eliminar, modificar ou gerar pacotes), a maioria das vulnerabilidades identificadas na especificação original do *Count-Min* poderiam ser resolvidas com a utilização de funções de dispersão criptográficas (ao invés de funções de dispersão pouco complexas, como as sugeridas pelos autores do *Count-Min*) e de um mecanismo para impedir que os contadores excedam a sua capacidade máxima.

Os algoritmos baseados em *sketches* foram desenhados para monitorizar uma dada métrica durante um período finito de tempo, após o qual a sua estrutura de dados começa a ficar demasiado preenchida e o número de colisões aumenta. Por essa razão, no fim desse período a estrutura de dados deverá ser reiniciada. No entanto, no contexto da monitorização de redes de computadores é necessário que o algoritmo de monitorização esteja continuamente a executar a sua função, sem momentos de pausa. Nesse sentido, além de adicionar segurança à versão original do algoritmo, desenvolvemos um mecanismo que permite utilizar algoritmos baseados em *sketches*, como o *Count-Min*, no contexto da monitorização de redes. Para tal, no final de cada período de monitorização, definido pelo administrador de sistemas, a estrutura de dados usada é reiniciada em tempo de execução.

Os *switches* e *routers* atuais não têm, no entanto, a capacidade de executar estas técnicas avançadas de monitorização (isto é, os sketches). Felizmente, nos últimos anos surgiram *switches* programáveis – existindo já alguns em produção – que criam finalmente a possibilidade de adicionar ao plano de dados de uma rede estas funcionalidades. Desta forma, o algoritmo de monitorização que propomos foi implementado em P4, uma linguagem recente que permite a programação dos dispositivos de encaminhamento reprogramáveis. A utilização do P4 permitiu-nos programar diretamente no plano de dados, dando-nos inclusivamente a possibilidade de alterar valores mantidos pelo algoritmo de monitorização sem ter de parar a sua execução.

Decidimos usar o MD5 (Message-Digest Algorithm 5) para gerar as funções de dispersão criptográficas, por este ter uma complexidade temporal menor comparativamente a outras funções criptográficas e porque ainda é considerado seguro se usado em conjunto com uma chave de 128 bits. Esta chave é um número aleatório, gerado no arranque do algoritmo de monitorização e guardado na memória do switch programável, podendo ser acedida internamente pelo próprio código do algoritmo ou externamente através de uma

interface oferecida pelo dispositivo. Uma vez que a segurança das funções de dispersão vai depender desta chave, é fundamental impedir que o adversário a descubra. Por essa razão, e porque os algoritmos baseados em *sketches* necessitam de reiniciar a sua estrutura de dados periodicamente, como já referido, desenvolvemos uma solução que não só altera a chave que está a ser utilizada por uma nova, como também reinicia a estrutura de dados do algoritmo, logo após a ter serializado e copiado para um ficheiro. Esta cópia é necessária pois sempre que há um pedido ao algoritmo de monitorização para estimar a frequência de determinado item, todas as estruturas de dados têm de ser consultadas, incluindo as armazenadas no ficheiro, o que é feito de forma transparente pelo nosso algoritmo.

Durante a implementação da nossa solução, tivemos de ultrapassar algumas dificuldades decorrentes não só das peculiaridades da linguagem P4 como da própria interface entre o código P4 e o *software* utilizado para emular um dispositivo de encaminhamento. Entre as principais dificuldades que o P4 nos colocou e que resulta das peculiaridades de um switch, nomeadamente a necessidade de processar pacotes a altas taxas de transmissão, está o facto de este não permitir definir ciclos, o que nos era necessário para repetir as ações para cada linha da matriz. Acabámos por resolver a situação com sucesso de uma forma não convencional. A interface oferecida pelo dispositivo de encaminhamento virtual (*software switch*) também nos colocou algumas dificuldades, entre as quais o facto de só permitir que as funções de dispersão devolvam um resultado com no máximo 64 bits. Uma vez que a execução do MD5 devolve 128 bits, para o seu resultado poder ser utilizado tivemos de modificar o software do dispositivo de encaminhamento de forma a garantir a interoperabilidade com o programa P4 desenvolvido.

A avaliação que executámos focou-se no desempenho e funcionalidade, comparando a nossa solução segura com o *Count-Min* original (que também implementámos em P4) e com um algoritmo base que apenas encaminha o tráfego sem fazer qualquer tipo de monitorização. Ao nível da latência, observámos que a monitorização através de um algoritmo baseado no *Count-Min* induz um atraso no processamento efetuado pelo dispositivo de encaminhamento de cerca de 0,7 milissegundos por pacote (com uma matriz de 20 linhas). O atraso adicional inserido pela nossa versão segura foi, em média, de menos de 0,2 milissegundos. Avaliámos também a taxa de transferência que o dispositivo de encaminhamento consegue atingir quando corre a nossa solução, tendo observado que esta se mantém sempre muito próxima da taxa de transferência obtida pela versão original do *Count-Min*. Comparámos ainda o erro das estimativas dadas pelo algoritmo com o erro máximo teórico apresentado na especificação do algoritmo original para uma dada probabilidade. Não observámos diferenças relativamente ao erro entre a versão original do *Count-Min* e a segura. Assim, pudemos concluir que a utilização de uma versão segura do *Count-Min* não introduz penalizações relevantes no desempenho e na funcionalidade do algoritmo de monitorização, apesar das garantias de segurança oferecidas.

Palavras-chave: Monitorização, Segurança, Redes de Computadores, *Sketches*, Planos de Dados Programáveis.

Abstract

Monitoring is a fundamental activity in network management as it provides knowledge about the behavior of a network. Different monitoring methodologies have been employed in practice, with sample-based and sketch-based approaches standing out because of their manageable memory requirements. The accuracy provided by traditional sampling-based monitoring approaches, such as NetFlow, is increasingly being considered insufficient to meet the requirements of today's networks. By summarizing all traffic for specific statistics of interest, sketch-based alternatives have been shown to achieve higher levels of accuracy for the same cost. Existing switches, however, lack the necessary capability to perform the sort of processing required by this approach. The emergence of programmable switches and the processing they enable in the data plane has recently led sketch-based solutions to be made possible in switching hardware.

One limitation of existing solutions is that they lack security. At the scale of the datacenter networks that power cloud computing, this limitation becomes a serious concern. For instance, there is evidence of security incidents perpetrated by malicious insiders inside cloud infrastructures. By compromising the monitoring algorithm, such an attacker can render the monitoring process useless, leading to undesirable actions (such as routing sensitive traffic to disallowed locations).

The objective of this thesis is to propose a novel sketch-based monitoring algorithm that is secure. In particular, we propose the design and implementation of a secure and scalable version of the Count-Min algorithm [16, 17], which tracks the frequency of items through a data structure and a set of hash functions. As traditional switches do not have the capabilities to allow these advanced forms of monitoring, we leverage the recently proposed programmable switches. The algorithm was implemented in P4 [11], a programmable language for programmable switches, which are now able to process packets just as fast as the fastest fixed-function switches [12]. Our evaluation demonstrates that our secure solution entails a negligible performance penalty when compared with the original Count-Min algorithm, despite the security proprieties provided.

Keywords: Monitoring, Security, Computer Networks, Sketches, Programmable Data Planes

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Monitoring is the activity of supervising something in order to ensure it is operating as expected. Monitoring of a computer network is the only way a network administrator has to know the state of the network, enabling quick responses to anomalies or to make the required configuration adjustments. Selecting the right metrics to monitor is an important decision to make, in order that the most useful information can be retrieved without wasting resources. Some of the most used metrics include: network availability (amount of time, in a specific time interval, during which the network infrastructure is operational), utilization (ratio of the bandwidth used by the traffic being sent/received over the overall capacity), packet loss rate (ratio of packets lost with respect to packets transmitted) and network latency (time a packet takes to get from one designated point to another).

Different approaches can be used for monitoring. Ideally, for complete accuracy, the monitoring task should store all transmitted packets for subsequent analysis. In practice, however, this technique would lead to storage and processing scalability issues. Fortunately, exact results are usually not necessary, and a high quality approximation is enough. This fact suggests the use of probabilistic algorithms, that use smaller amounts of memory and require less computation to achieve the desired goals.

Traditional Network Monitoring

To avoid the storage and processing of all packets, as would be required by naive monitoring, traffic data can be reduced by *sampling*, with only a subset of the traffic being captured. The frequency at which packets are collected is the *sampling rate*: the number of samples taken per unit of time.

A proprietary (Cisco) protocol, NetFlow [3], uses sampling since the introduction of Cisco 12000 [7] and has been considered a reference. Netflow is a protocol for monitoring of network traffic flow data generated by switches that support it. A network flow can be defined as a unidirectional sequence of packets that share the following values: ingress interface, source and destination IP addresses, source and destination TCP ports,

IP protocol, and IP type of service.

A major problem of solutions based on sampling is the potential lack of accuracy achieved, as many packets are ignored. To be scalable, the sampling frequency of these solutions is kept at low levels, with sampling rates of 1:1000 (one packet in 1000) being common [8]. This reduces the accuracy to a level that precludes its use for many of the advanced monitoring capabilities required in today's large scale networks that enable cloud computing.

Sketch-Based Algorithms

To maintain memory and processing at acceptable levels, sketch-based algorithms summarize the network data streams in the data plane (by employing hashing, counting, and filtering techniques). These solutions have been shown to offer an interesting trade-off between the accuracy achieved and the memory used, outpacing the alternative for various monitoring tasks. Existing switches, however, lack the necessary capability to enable this approach.

Sketches are data structures that use *sub-linear* space, meaning that the memory size used grows sub-linearly with the input data. Whenever the size of the memory used is smaller than the input, the accuracy loss is inevitable, leading to probabilistic results. Sketch-based algorithms, however, still provide high-quality results approximations, which very commonly is as useful as the exact results.

Programmable Networks

In this thesis we focus our attention in sketch-based algorithms. An initial problem that we thus face is on their practicality. Until recently, network switches and routers did not have the required capabilities for implementing sketches in practice. The emergence of programmable switches has given operators the opportunity to run complex processing in the data plane, radically changing the state of affairs. Recent proposals [37, 30] have shown the feasibility of sketch-based solutions in real hardware data planes.

In traditional networks, network devices have the control plane, used to populate the forwarding table, and the data plane, that entails the process of consulting the forwarding table to decide the interfaces where packets should be transmitted, coupled together inside the same piece of equipment. Being hardware appliances, to achieve the required performance, this kind of networks tends to be static due to the little flexibility hardware provides for evolution.

On the other hand, software-defined networks (SDN) [28] decouples the data plane from the control plane, allowing flexible control of the network. The decoupling of the control plane makes logically centralized network control possible, which allows, among

other things, to observe the whole network from a single vantage point. The SDN concept has been recently extended to the data plane. Production-level programmable switches are now available (e.g., Barefoot Tofino), allowing programmability of the data plane itself – i.e., it is now possible to define precisely how packets should be processed in these switches using a high-level programming language (such as P4 [11]).

These software-defined networks can be monitored using traditional techniques, but also newer ones, such as those based on sketches. Some work has indeed already been done in order to adapt sketch-based algorithms to the SDN architecture. OpenSketch [39], for instance, is a software defined measurement architecture, with the data plane having a library of predefined sketches that can be combined in the control plane to create the required measurement algorithm. Hashpipe [37] is a very recent solution in P4, that takes advantage of programmable data planes.

1.1 Motivation

Many efficient sketch-based algorithms have been proposed over the years to face the requirements of real-time monitoring. With the growing network speeds, the proposed solutions had to be able to fulfill their job faster. Since their focus has been on this requirement, these solutions tend to neglect security in favor of optimal execution time and memory usage.

Indeed, if the monitoring algorithm itself is not secure, its results may be corrupted. In the worst-case scenario, the network administrator does not notice the results are corrupted, and takes improper actions. For instance, an attacker may persuade the monitoring system to route sensitive traffic to a location he or she controls. Unfortunately, there is evidence that the problem is real. A recent report mentions malicious insiders as one of the top threats in cloud computing [31], what is evidenced by the occurrence of instances of this problem in companies such as Google [2, 27]. The security limitation of current approaches is therefore already a serious concern.

There is already some initial work [33, 35] addressing the security of traditional SDN monitoring but, to the best of our knowledge, no attempt has hitherto been made to address the security of sketch-based algorithms. Our work starts filling that gap.

1.2 Goals

The objective of our work is to design, implement and evaluate a secure version of a sketch-based algorithm – Count-Min [16, 17] – that enables secure traffic monitoring, while still guaranteeing acceptable execution speeds and memory usage requirements.

The sketch should take advantage of the benefits SDN networks have to offer, namely the possibility to program the data planes. For this purpose, the sketch will be imple-

mented in P4 [11], a language that allows the programming of switches. The solution should allow a network administrator to monitor his network securely. By employing a sketch-based approach, monitoring outputs, despite not being exact, will be approximations of higher-quality than those provided by algorithms based on packet sampling. Compared to the previous sketch-based alternatives, the monitoring results will be trustworthy, increasing the network administrator confidence that he is taking the proper managing actions.

1.3 Contribution

The main contribution of this work is the design of a secure version of a sketch-based algorithm, the Count-Min, which should be able to perform, in a secure way, the monitoring task it was designed for. Our solution addresses several technical challenges, many of which arise from the constraints imposed by real switches. These include the use of cryptographic hash functions (not supported in existing switches), avoiding loops (not directly available as they would limit throughput), and techniques for secret key renewal. We prototyped our solution in P4 [11], a programming language for network switches.

In terms of performance, we measure the latency and throughput our solution achieves and compare it with two other algorithms, also implemented in P4: the original Count-Min algorithm and an algorithm that only forwards the traffic. We also calculate the errors in the estimations returned by our solution, while monitoring by source IP address and by flow. Our evaluation using the public-domain behavioral P4 switch model [1] demonstrates that securing the sketching algorithm introduces a negligible performance penalty. We also observe that flow-based monitoring requires the use of a larger data structure to achieve the same errors in estimations, when compared with monitoring based on source IP addresses only.

1.4 Planning

This section presents the proposed work plan. To help visualize it, a Gantt chart illustrating the schedule is shown bellow.

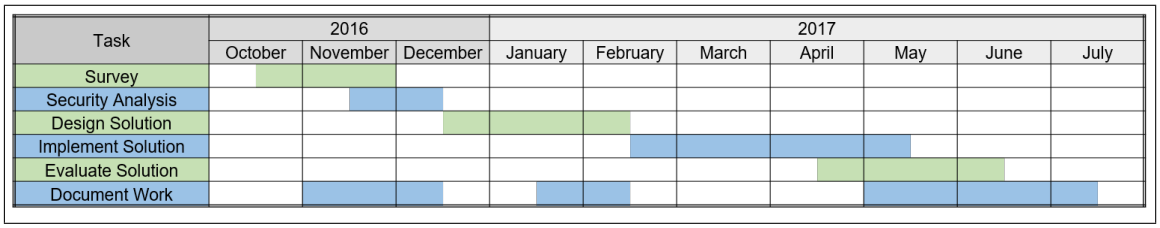


Figure 1.1: Work Plan

In the *Survey* task, a survey of sketch-based algorithms was done alongside reading of other related work. The *Security Considerations* task included the investigation of potential security problems of sketches. In *Design Solution* phase, a secure version of a sketch was designed. The sketch was then implemented in the *Implement Solution* task, and evaluated in the *Evaluate Solution* task. The *Document Work* task is related to the writing of this document.

1.5 Structure of the document

This document is organized as follows:

- Chapter 2 - Related Work: This chapter presents a survey of sketch-based algorithms, security considerations about them, and some context about programmable switches.
- Chapter 3 - Design: In this chapter the design of the solution will be presented, including the way the memory data structures are reconfigured over time.
- Chapter 4 - Implementation: In this chapter the most important P4 implementation details are presented.
- Chapter 5 - Evaluation: This chapter presents the evaluation performed in terms of performance and functionality of the proposed solution.
- Chapter 6 - Conclusion: A conclusion about the work done is presented in this chapter.

Chapter 2

Related Work

In this chapter, we start with a survey on sketch-based monitoring algorithms. For each algorithm, we describe the data structure it requires, the methods it provides, and the accuracy that it is able to guarantee. Afterwards, we investigate these algorithms, giving special attention to the Count-Min, with respect to security. For different adversary capabilities we tried to identify the possible actions a malicious user could take. Finally, in the last section, we describe the emergent programmable switches. We give special attention to the P4 language, that is used to program these switches, and that we use to implement our solution. Besides identifying the language's goals, structure and architecture, we also present recent proposals that make use of the P4 language for network monitoring.

2.1 Sketch-Based Monitoring

Network monitoring algorithms used today are mostly sample-based or sketch-based. These kind of algorithms, which aim to be memory and CPU efficient, are probabilistic because they are not able to guarantee that exact results are always provided. Instead, they aim to guarantee that high quality approximations are returned, which may be as useful as the exact results. Sample-based algorithms monitor only a subset of the traffic that arrives at that device. For that reason, many packets are not monitored, which leads to accuracy problems. On the other hand, sketch-based algorithms process *every* packet, performing a summarization (mainly by hashing and counting) for a specific statistic of interest. Importantly, the algorithms are designed with provable accuracy-memory tradeoffs.

This section presents some of the most well-known sketch-based algorithms that can be used to monitor networks. The algorithms are categorized according to the problem they propose to solve. It is assumed a monitoring model where there is an external entity that periodically collects the sketch's counters. Immediately after this, all counters are restarted and a new monitoring cycle begins.

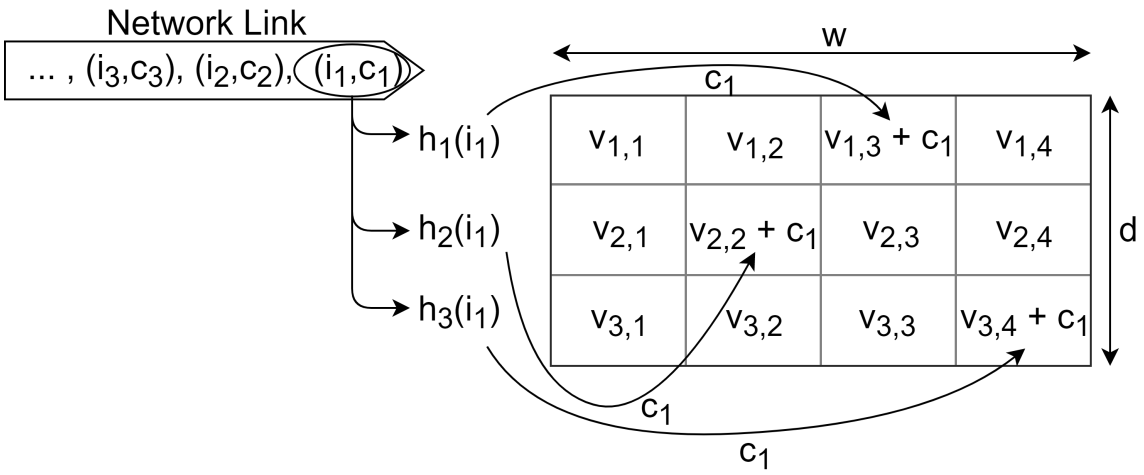


Figure 2.1: Count-Min sketch data structure with width $w = 4$ and depth $d = 3$

2.1.1 Heavy Hitters

This section presents a set of sketches designed to identify flows that are larger (in number of packets or bytes) than a fraction of all flows seen during a time interval. Identifying heavy-hitters is important for several network applications, such as traffic engineering, anomaly detection and DDoS prevention.

Count-Min Sketch

The Count-Min sketch [16, 17] identifies the heavy hitters in a stream by solving the *Count Tracking* problem, where the goal is to find the frequency of each item in a stream with a large number of items.

Data Structure The data structure used is a two-dimensional array of counters with width *w* and depth *d*, both fixed at the time of creation. These values, *w* and *d*, are chosen based on the desired accuracy of the estimates. The counters are initialized with zero.

In addition, *d* hash functions must be chosen uniformly at random from a pairwise-independent family. At update time, each of these functions maps the item onto the range $\{1, 2, \ldots, w\}$.

Methods The sketch provides two methods: *update(i,c)*, which updates the frequency of item i by c, and *estimate(i)*, which gives the estimated frequency of i.

Update(*i*,*c*): When a new item *i* arrives, for each *d* row the corresponding hash function is applied to *i* in order to determine the position in that row of the target counter. Value *c*, which may be positive or negative, is then added to the target counter.

Estimate(*i*): In order to estimate the frequency of an item *i*, for each of the *d* rows the corresponding hash function is applied to *i*. This gives the position of the target counter in every row. After the *d* counters are found, the one with the smallest value is chosen. The value of that counter is then returned.

Accuracy Summarizing a stream normally results in the loss of some accuracy. To minimize this loss, the sketch dimensions should be as high as possible. This way, the probability of collision is lower and, as a consequence, the average accuracy of the estimates will be higher. Another factor contributing to the accuracy is the duration of the monitoring cycles. Whenever the counters are restarted, the accuracy of the sketch is perfect, starting to decrease after the occurrence of collisions.

If N is the sum of the values of all the counters in a row of a sketch of size $w \times d$, the frequency of an item i returned by the algorithm is at most $\frac{2}{w}$ of N more than its true frequency, with a probability of at least $1 - \left(\frac{1}{2}\right)^d$.

Example: For a query to have an error of at most 0.001 of N with a probability of at least 0.999, the sketch dimensions should be the following:

- $\frac{2}{w} = 0.001 \Leftrightarrow w = 2000$
- $1 - \left(\frac{1}{2}\right)^d = 0.999 \Leftrightarrow \left(\frac{1}{2}\right)^d = 0.001 \Leftrightarrow d = \frac{log(0.001)}{log(0.5)} \Leftrightarrow d \simeq 10$

Note: If the resulting value is not an integer, it must be rounded up in order to preserve the guarantees.

Count Sketch

The Count Sketch [14, 15] can also be used to identify heavy hitters. While the Count-Min sketch can be used to count packets or bytes, the Count sketch can only be used to count packets, as there are only two possible update values: $+1$ and -1.

Data Structure The data structure is, like for the count-min sketch, a two dimensional array with w width and d depth. The dimensions of the array are going to have an effect on the accuracy achieved in the estimates. Each d row should be interpreted as an hash table with all its slots initialized to zero.

The count sketch also needs d hash functions to map objects onto $\{1, ...w\}$ and another d hash functions to map those same objects to $+1$ or -1. Let i be the object to map. Then the hash functions are: $h_1...h_d : i \rightarrow \{1...w\}$ and $s_1...s_d : i \rightarrow \{+1, -1\}$.

Methods For the operation of the algorithm, two methods are provided: the *update* method, called whenever a new item arrives, and the *estimate* method, that returns the estimated frequency of the queried item.

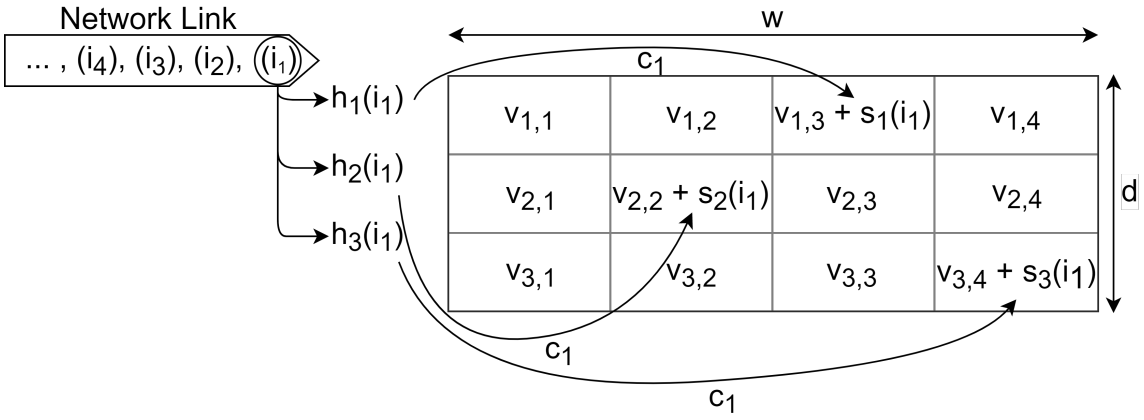


Figure 2.2: Count sketch data structure with width w = 4 and depth d = 3

Update(*i*): For a given item i, the algorithm applies for each row the corresponding hash function h to i. This operation gives the position in that row where the target counter is located for i. After that, a second hash function, s, is applied to i, which will produce the value $+1$ or -1. Finally, the result of the function s is added to the value in the target counter.

Estimate(*i*): Let j be an iterator over the rows, h_j a function that identifies a position in row j and s_j a function that returns $+1$ or -1. For each row j, the product of $h_j(i)$ and $s_j(i)$ is calculated. The median of these j products is the estimated frequency returned.

The median should be used instead of the mean because of the mean sensitivity to outliers. For example, if there is a counter with a value far from the others, which will probably happen due to collisions, the inaccuracy of the returned value would be higher if the mean was used instead of the median.

Accuracy Let $F2$ be the sum of the squares of the frequencies of the items. For this version of the Count Sketch algorithm, the data structure used should have $w = \frac{1/\varepsilon^2}{2}$ and $d = log(\frac{1}{\delta})$, in order to have an error of at most $\varepsilon\sqrt{F2}$ with a probability of at least $1 - \delta$.

2.1.2 Frequency Moments

The problem of calculating the frequency moments was defined in [9] as described next. Consider a sequence of items $S = (a_1, a_2, ..., a_m)$, where each a_i is a number between 1 and n and m_i denotes the number of occurrences of i in S. For each $k \geq 0$, the kth frequency is defined as:

$$F_k = \sum_{i=1}^{n} m_i^k \tag{2.1}$$

There are several frequency moments that have different applications by providing useful statistics about the sequence. For example, F_0 represents the number of distinct elements

in a stream and F_1 is the number of elements of a stream. The sketch shown below is used to estimate F_2, which is the *Gini's index of homogeneity*, an index that is required in the calculation of the *surprise index* [25] of a sequence. The surprise index is a measure of the degree of surprise associated with the occurrence of an event. The larger the index, the more surprising the occurrence of the event is. An event is surprising if its probability is small compared with the probabilities of occurrence of other events. Considering P_r the probability of the event that actually occurred, the surprise index is calculated with:

$$Surprise\ Index = \frac{F_2}{P_r} \tag{2.2}$$

AMS Sketch

The AMS Sketch is useful to estimate, using a compact data structure, the value of F_2 of the frequency vector containing the data stream. The second frequency moment ($F2$) of a vector v is defined as the square of its Euclidean norm (also called L_2 norm), which can be represented as $||v||_2^2$.

The sketch was proposed originally in 1996 [9] but since then other authors have optimized it [15]. In this newer version of the sketch, the update time is reduced by $O(\frac{1}{\varepsilon^2})$, maintaining the same guarantees and requirements of space.

Data Structure The data structure used by this sketch is an array of width $w = \frac{1}{\varepsilon^2}$ and depth $d = log(\frac{1}{\delta})$. For each d rows, an hash function h maps the items to $\{1, 2, ...w\}$. A second hash function, g, is needed to map those same items to $\{+1, -1\}$. Function g must be *fourwise* independent [38]. All entries of the array are initialized to zero.

Methods The algorithm uses an *update* method to update the data structure whenever a new item arrives and an *estimate* method that returns the second frequency moment, F_2, of the vector containing the data stream.

Update(*i,c*): For each row j between 1 and d, $h_j(i)$ is computed to obtain the position of the target counter in that row. After that, the result of $c \times g_j(i)$ is added to the value in the target counter, positioned in row j and column $h_j(i)$. At the end of each j iteration, the target counter value is: $target\_counter = target\_counter + c \times g_j(i)$.

Estimate(*i*): Let D be the data structure where $D[j, k]$ represents the entry in row j and column k. For each row j, compute $\sum_{k=1}^{w} D[j, k]^2$. The median of these sums is the estimate of F_2 returned by the algorithm.

Accuracy Recall that v is an imaginary vector containing the full data stream. The sketch guarantees that, given a certain sketch dimension, with a probability of at least $1 - \delta$ the estimate returned by the algorithm is between $(1 - \varepsilon)||v||_2^2$ and $(1 + \varepsilon)||v||_2^2$ or, in a simplified version, $(1 - \frac{\varepsilon}{2})||v||_2$ and $(1 + \frac{\varepsilon}{2})||v||_2$.

2.1.3 Detection of Traffic Changes

The detection of traffic anomalies is crucial to identify failures and attacks in a network. However, to do this perfectly it is necessary to analyze each flow individually, which may be too expensive. For this reason, a solution that uses sketches is preferable due to its capability to summarize the traffic and still provide an accurate estimate when queried.

There are two main approaches to identify traffic anomalies: (1) looking for anomalies that match the behavior of some known anomaly and (2) compare the traffic with a model of normal behavior constructed based on past traffic history. The first approach has the same problem of blacklists: new anomalies/attacks are not detected because to identify them it is required to know their behavior in advance. The second approach does not require anything to be known a priori, allowing the detection of new anomalies. However, sometimes what is considered an anomaly by this solution may in fact be the normal behavior that is new.

K-ary Sketch

The k-ary sketch [29] was developed to identify traffic anomalies in an efficient, accurate, and scalable way. The k-ary operation follows the second approach described above. It looks for significant changes in the behavior of the traffic when compared to a model of a behavior considered normal, constructed based on past traffic history. The algorithm has three modules:

- **Sketch module**: Similar to other sketches, where the data stream is summarized in a sketch S_o;
- **Forecasting module**: Produces a forecast sketch S_f using some forecast model based on the observed sketches S_o;
- **Detection module**: A forecast error sketch S_e is calculated $S_e = S_o - S_f$. Using the S_e sketch, the change detection module verifies if the forecast error is above a defined threshold and if so, a potential anomaly is identified.

Data Structure The k-ary sketch uses a two dimensional array of counters of width w and depth d. Each row has an associated hash function that maps the items of the data stream onto $\{1, 2, ...w\}$. These hash function must be, like in the AMS Sketch, *fourwise* independent [38], in order to keep the sketch guarantees of accuracy.

Methods The algorithm provides 4 methods: the *update* method, to update the sketch, the *estimate*, to reconstruct an approximation of the stream of items with specific key, the *estimateF2*, used to estimate the second frequency moment of the vector containing the data stream, and *combine* to compute the linear combination of multiple sketches.

Update(*S*, *i*, *c*): Let $S[j,k]$ represent the counter located in row j and column k of sketch S. For each row j between 1 and d, add to the counter in [j, $h_j(i)$] the value c.

Estimate(*S*, *i*): For each row of a sketch S, the following is calculated, where $sum(S) = \sum_{k=1}^{w} S[1,k]$ is the sum of all values in the sketch (computed only once):

$$\frac{S[j,h_j(i)] - sum(S)/w}{1 - 1/w} \tag{2.3}$$

The estimate returned for the given key i is the median among the results of all rows.

EstimateF2(*S*): For each row j of the sketch S, the following is calculated:

$$\frac{w}{w-1} \cdot \sum_{k=1}^{w} (S[j,k])^2 - \frac{1}{w-1} \cdot (sum(S))^2 \tag{2.4}$$

The second frequency moment estimated is the median between the results of all rows.

Combine($\mathbf{c_1}$, $\mathbf{S_1}$, ..., $\mathbf{c_\ell}$, $\mathbf{S_\ell}$): Let $c_1 ... c_\ell$ be scalars and $S_1 ... S_\ell$ be sketches. The linearity of the sketch data structure allows the linear combination of multiple sketches:

$$Result\ Sketch = \sum_{k=1}^{\ell} c_k \cdot S_k \tag{2.5}$$

Forecasting Module There are several models that can be used for forecasting and change detection. There are relatively simple smoothing models that work with the weights assigned to each previous sketch and models belonging to the family of AutoRegressive Integrated Moving Average (ARIMA) models [13], which identify the linear dependency of the future values on the past values.

Change Detection Module This is the module responsible for detecting the variations. Initially, the forecast error sketch $S_e(t)$ is constructed, based on the observed sketch and the sketch that results of the application of the forecast module: $S_e = S_o - S_f$.

For any given i key, it is possible to reconstruct its forecast error in $S_e(t)$ at any time using the *Estimate($S_e(t)$,i)* method. The detection alarm is raised if the estimated forecast error is above a threshold T_A. The value of T_A is calculated based on a variable T, defined

by the application, and the estimated second moment of frequency of the forecast error sketch $S_e(t)$:

$$T_A = T \cdot [EstimateF2(S_e(t))]^{\frac{1}{2}} \tag{2.6}$$

Considering the data stream as a series of (*key, value*) pairs. The algorithm can only indicate if, for a given key, the pairs with that key have considerable change. This process is irreversible, meaning it is required to know the keys to query to find the streams that changed more than the threshold.

One possible solution is to brute-force the keys. In this solution, all the keys of the stream in an interval t are recorded and then replayed after the $S_e(t)$ is constructed. The problem is that this approach is not scalable with a large set of keys. A solution that reverses the k-ary sketch is presented in [34]. By modifying the algorithm's update procedure with a set of techniques, it allows to efficiently infer the keys of target flows from sketches.

Accuracy Like in the other sketches, the dimensions of the sketch, w and d, are crucial to the accuracy that can be achieved. Consider v_i^{est} the estimated returned for the item i by the method *estimate* and F_2^{est} the estimated frequency moment returned by the method *estimateF2* of the algorithm.

***Estimate* method:** For an item i, $T \in (0, 1)$ and $\alpha \in [1, \infty)$, if $|v_a| \geq \alpha T\sqrt{F_2}$, then

$$\Pr\left\{|v_i^{est}| \leq T\sqrt{F_2}\right\} \leq \left[\frac{4}{(w-1)(\alpha-1)^2T^2}\right]^{d/2} \tag{2.7}$$

***EstimateF2* method:** For any $\lambda > 0$,

$$\Pr\left\{|F_2^{est} - F_2| > \lambda F_2\right\} \leq \left[\frac{8}{(w-1)\lambda^2}\right]^{d/2} \tag{2.8}$$

2.1.4 Counting the Number of Distinct Flows

The problem of counting the number of distinct header patterns (flows) seen during a measurement interval is addressed by the algorithms presented in this section. An intrusion detection system (IDS) looking for port scans, for example, can count, for each active source address, the distinct flows defined by destination port and IP address. If a source IP has more than a defined number of distinct flows opened during the measurement interval, it is probably performing a port scan.

Direct Bitmap

A Direct Bitmap [20] is a sketch-based algorithm that addresses the problem of counting the number of distinct flows among packets received on a link during a time period. This task may be specially difficult if the right algorithm is not used because nowadays network links work at very high speeds, allowing the execution of only a small number of accesses to memory per packet.

Data Structure The data structure is an array of bits, also called a bitmap, of size b, with all its bits set to zero at the beginning. It is also required an hash function h to map each flow to a bit of the bitmap. Considering N as the maximum number of flows and ε as the acceptable average relative error, then the size of the bitmap b is the result of $\lceil \frac{N}{ln(N\varepsilon^2+1)} \rceil$.

Methods The algorithm has only two operations: *update*, called whenever an item comes in; and *estimate*, used at the end of the measurement interval to get the number of distinct items.

Update(*i*): Whenever an item (packet) i arrives, the hash function is applied to its header pattern (used to identify the flow). The hash function returns the position of the array where the bit associated to that flow is located. That bit is then set to 1, if it was not already set by a previous item belonging to the same flow (or to a flow that maps to the same bit).

Estimate(): Let z be the number of unset bits. The number of unique elements returned is given by the following equation, where $\hat{n}$ is the estimated number of distinct elements (flows):

$$\hat{n} = b \ln \left(\frac{b}{z} \right) \tag{2.9}$$

Accuracy The algorithm's accuracy is not perfect since the bitmap size is smaller than the number of existing flows. Because of that, collisions will occur with a random probability.

Let n be the real number of distinct elements and ρ the flow density, defined as the average number of flows that hashes to the same bit. In order to achieve the best possible accuracy, the value for ρ should be the one that maximizes the accuracy. The standard deviation of the ratio $\frac{\hat{n}}{n}$ given by this algorithm is calculated with the following equation:

$$SD \left[\frac{\hat{n}}{n} \right] \approx \frac{\sqrt{e^{\rho} - \rho - 1}}{\rho \sqrt{b}} \tag{2.10}$$

Virtual Bitmap

The Virtual Bitmap [20] derives from the Direct Bitmap algorithm, described above. However, it uses less memory than the Direct Bitmap, by covering only a portion of the flow space. A Virtual Bitmap that covers the entire flow space is a Direct Bitmap.

Because of the limited memory space, this algorithm samples the flow space. This sampling factor must be chosen before the execution of the algorithm, based on the expected number of flows. For a given memory size, the larger the number of flows, the smaller the flow space covered. For this reason, the sampling factor must be chosen carefully because if the number of flows is too large, the virtual bitmap will have the same problems as an underdimensioned Direct Bitmap.

Data Structure Like in the Direct Bitmap, the Virtual Bitmap also uses an array of bits of size b and an hash function h to map the flows to specific positions of the array.

Consider a threshold on the number of distinct flows that are allowed before the raise of an alarm by the algorithm. In order to maximize the accuracy, at the threshold the value of the flow density ρ (number of flows / b) should be 1.593624. By minimizing the algorithm's average error with equation 2.11, the algorithm's authors concluded that this was the optimal value for ρ. For that reason, the sampling factor chosen should allow the value of ρ to be around 1.6 at the threshold. The value of b should be $\frac{1.54413865}{\varepsilon^2}$, where ε is the average relative error, for the best results.

Methods Similar to the Direct Bitmap, this algorithm also provides two methods: *update* and *estimate*.

Update(*i*): An item i is hashed by h whenever it arrives. If the result of h is a position in the Virtual Bitmap, the bit in that position is set to 1. Otherwise, i is ignored and the Virtual Bitmap remains unchanged.

Estimate(): Let b be the size of the bitmap and s the flow space size. The number of distinct active flows $\hat{n}$ is given by the equation:

$$\hat{n} = s \ln\left(\frac{b}{z}\right) \tag{2.11}$$

Accuracy

Like the other sketch-based algorithms, the Virtual Bitmap does not provide perfect accuracy. Consider $\hat{n}$ the estimated number of unique elements, n the real number of unique elements and ρ the flow density. The standard deviation of the ratio $\frac{\hat{n}}{n}$ is given by the next equation:

$$SD\left[\frac{\hat{n}}{n}\right] \lessapprox \frac{\sqrt{e^{\rho}-1}}{\rho\sqrt{b}} \tag{2.12}$$

2.1.5 Count Traffic

The algorithms presented in this section are useful if one wants to count the number of distinct source addresses that send traffic to a set of destinations. For example, this problem can be addressed with a combination of a bloom filter, to keep the set of destinations, and a PCSA sketch, to maintain the count of distinct sources.

Bloom Filter

The Bloom Filter [10] is a sketch-based algorithm used to test whether an item i is contained by a set s. Its main contribution is to allow this task to run in a space-efficient way.

Data Structure The data structure needed for this algorithm is an array of bits of size b. In addition, it requires k different hash functions, so that each one maps each element of the set to a position in the array. Let n be the maximum number of items of the set and p the false positive probability of the test that determines if an item is contained by a set. To minimize the probability of false positives, b should be set to $-\frac{n \ln p}{(\ln 2)^2}$ and k to $\frac{b}{n} \ln 2$.

Methods The Bloom Filter provides an *add* method to insert an item into the set and a *test* method to determine whether an item is contained by the set.

Add(i): Whenever a new item i arrives, it is hashed by all the k hash functions. Each hash function returns a position in the array where the bit stored is then set to 1.

Figure 2.3 represents a Bloom Filter of size b = 16 and $k = 3$ to which three elements are inserted: a, b and c.

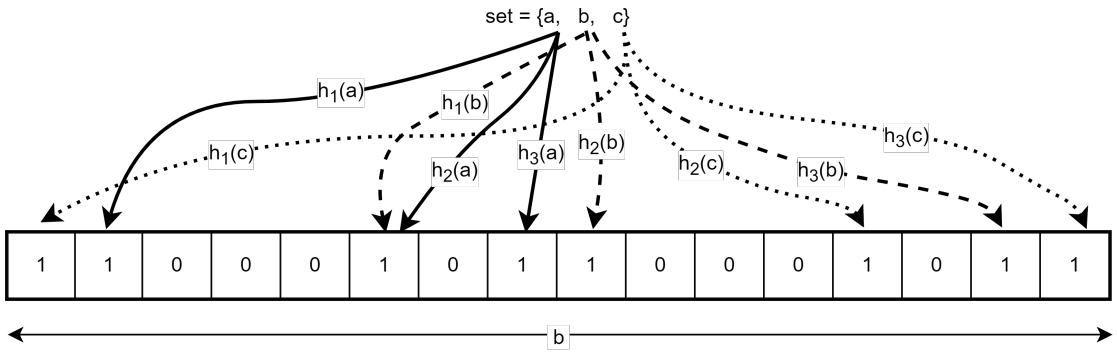


Figure 2.3: Inserted items in the Bloom Filter array with $b = 16$ and $k = 3$

Test(i): To find out if an item i is in the set, the item is hashed by the k functions. If the bits stored in the returned positions are all set to 1, then the item is contained by the set with a probability of p. Since the algorithm does not support the removal of items, which would generate false negatives, every bit set to 1 at update time remains with that value. So, if at least one of those bits is unset, the item is definitely not in the set. The removal of an item i_{remove} would require the bits mapped by the k functions to be unset. If at least one of the bits recently unset was shared with another item i_{keep} that is kept in the set, the test operation for the item i_{keep} would generate a false negative.

Figure 2.4 shows how the operation test is able to determine whether item i is contained by the set. The test failed as one of the hash functions (h_2 in this case) is mapped to a 0-bit, meaning that the item i has not been inserted in the set.

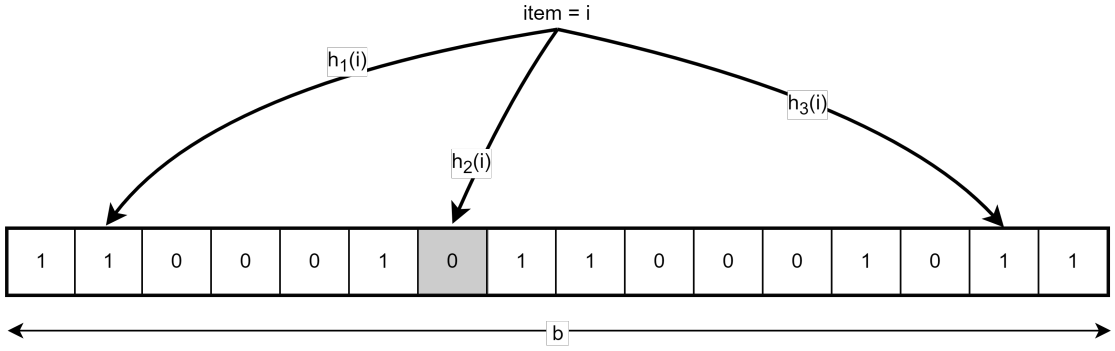


Figure 2.4: Test if the item i is in the set, with $b = 16$ and $k = 3$

Figure 2.5 illustrates a false positive. Item j is not in the set but it is mapped by all hash functions of the test to positions with 1-bits. For that reason, the test will consider that j is in the set, which is not true.

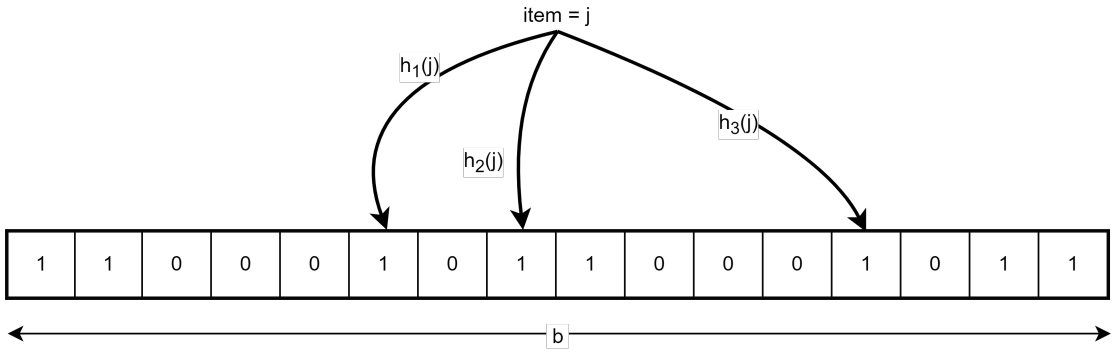


Figure 2.5: Test if the item j is in the set, with $b = 16$ and $k = 3$

Accuracy There are two different situations to consider regarding the result of the algorithm's *test* method: (1) it considers that the item is not contained in the set or (2) it considers that the item is contained in the set. Since the algorithm does not allow false

negatives, as explained above, the algorithm enjoys perfect accuracy in the first situation, meaning that the item is definitely not in the set. In the second situation, however, there is the possibility that the item has not been inserted in the set, which is a false positive. The probability of the *test* method to return a false positive is the probability of all the k bits stored in the positions where i hashes be set to 1. So, the probability p of a false positive is given by the equation bellow:

$$p = \left(1 - \left[1 - \frac{1}{b}\right]^{kn}\right)^{k} \approx \left(1 - e^{-kn/b}\right)^{k} \tag{2.13}$$

Probabilistic Counting with Stochastic Averaging

The Probabilistic Counting with Stochastic Averaging (PCSA) algorithm [23] provides the estimated number of distinct items in a collection of data. In the computer networking area, it is often used to count the number of distinct values of a header field (e.g., the source address of the packet).

Data Structure This algorithm uses w bitmaps with d positions each. The bitmaps can also be seen as a two dimensional bitmap of width w and depth d. The value of w determines the accuracy that can be achieved. The value of d should be at least $\log_2(\frac{n}{w})+4$, where n is the expected number of distinct elements. An hash function h is needed to uniformly distribute the items over the bitmaps of length d, by mapping the items onto the range $\{0...2^d - 1\}$. A function f to find the position of the least significant 1-bit in the binary representation of a value is also needed. Let bit(y, k) be a function that returns the position of the kth bit in the binary representation of y.

$$f(y) = \begin{cases} \min_{k \geq 0} \text{bit}(y, k) \neq 0 & \text{if } y > 0 \\ d & \text{if } y = 0 \end{cases} \tag{2.14}$$

Methods The *update(i)* function is called when a new item i is detected, keeping the algorithm's data structure updated. At the end of the measurement interval, the *estimate()* method is executed to estimate the number of distinct items that have been observed.

Update(i): Whenever an item i arrives, the algorithm starts by applying h to i. Let a be the remainder of $h(i)$ divided by w and b the result of applying f to the result of the integer division of $h(i)$ by w. The bit stored in $bitmap[a, b]$ are then set to 1. The update operation is represented in Figure 2.6 and described in Algorithm 1, where mod represents the module operator and div represents the integer division operator.

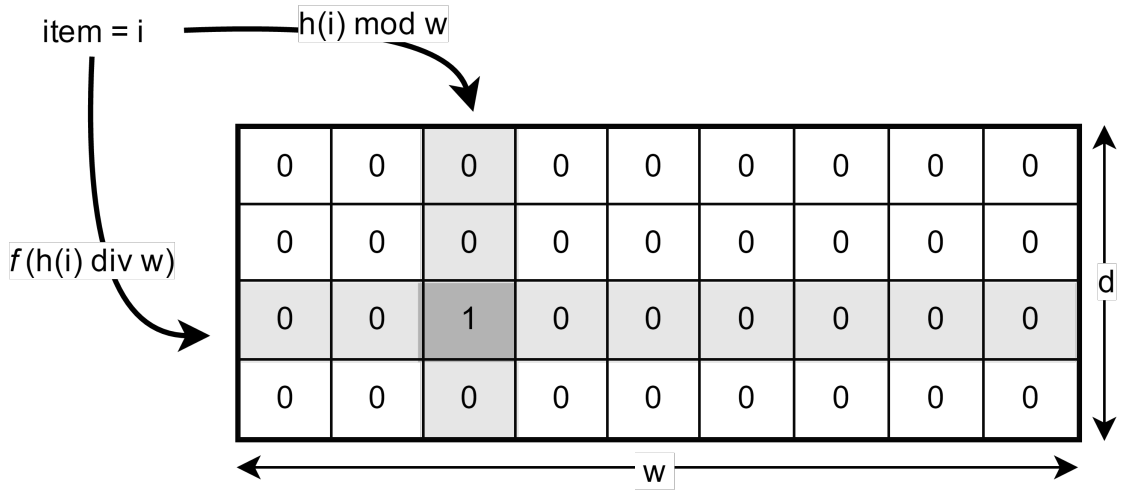


Figure 2.6: PCSA update operation with number of bitmaps w = 9 and depth d = 4

Algorithm 1 PCSA Update

1: **procedure** UPDATE(i)
2: $a = h(i) \bmod w$
3: $b = f(h(i) \text{ div } w)$
4: $bitmap[a, b] = 1$
5: **end procedure**

Estimate(): Let S and R be variables initialized to 0 and it be the iterator of a cycle made through w, starting with $it = 0$. For each iteration, if the bit in $bitmap[it, R]$ has value 1 and R is smaller than d, the value of R is incremented and, after that, the value of S is set to $S + R$. The estimated number of distinct items returned is the integer part of $\frac{w}{0.77351} \times 2^{\frac{S}{w}}$. This operation is described in Algorithm 2.

Accuracy The standard error ε of the value returned by the *estimate* method of this algorithm decreases as the number of bitmaps used increases and is approximated by the next equation.

$$\varepsilon \approx \frac{0.78}{\sqrt{w}} \tag{2.15}$$

LogLog and Super-LogLog Sketches

The LogLog and the Super-LogLog sketches [18] are used to estimate the number of distinct items in a set, by employing only a small auxiliary memory space and a single pass over each item.

The Super-LogLog Sketch is an improved version of the basic LogLog Sketch. In the following paragraphs we describe both the basic LogLog algorithm and the techniques through which the improvements are achieved. When nothing is said, the Super-LogLog

Algorithm 2 PCSA Estimate

```
1:  procedure ESTIMATE
2:      S = 0
3:      it = 0
4:      while it < w do
5:          R = 0
6:          while bitmap[it, R] = 1 and R < d do
7:              R = R + 1
8:              S = S + R
9:          end while
10:         it = it + 1
11:     end while
12:     return trunc( w/0.77351 × 2^(S/w) )
13: end procedure
```

works the same way as the basic LogLog algorithm.

Data Structure The data structure needed is an array of m memory units taking only $\lceil \log_2(\log_2(N_{max})) \rceil$ bits each, where N_{max} is the maximum number of distinct elements expected. All positions of the array are initialized to zero. The value of m determines the accuracy of the algorithm, as shown by equation 2.18.

Like in the PCSA sketch, an hash function h is needed to transform the input items into binary strings of size H. The value of H, corresponding to the length of the hashed items, must satisfy $H \geq \log_2 m + \lceil \log_2(\frac{N_{max}}{m}) + 3 \rceil$. A second function, f, is needed to find the rank of the first 1-bit, counting from left, in a sequence of bits. Thus, $f(1...) = 1$, $f(001...) = 3$, $f(0^k) = k + 1$, etc.

In the Super-LogLog algorithm, it is possible to reduce the size of each m memory unit to $\lceil \log_2 \lceil \log_2(\frac{N_{max}}{m}) + 3 \rceil \rceil$ bits.

Methods The algorithm provides a method to add a new item to the data structure and another one to estimate the number of distinct elements added.

Update(i): Whenever an item i arrives this method is called and i is immediately hashed by h. Consider $k = \log_2(m)$. The value of the first k bits of $h(i)$ is an index j to a position in the array. The position j in the array contains a value, say $M(j)$, that is then set to the maximum between its previous value and the output of function f applied to the binary representation of $h(i)$ without its first k bits.

Estimate(): This method is used to get the output from the algorithm. Consider $\alpha_m \sim \alpha_\infty - \frac{2\pi^2 + log^2 2}{48m}$, where $\alpha_\infty = 0.39701$. In a practical implementation with $m \geq 64$, α_m can be replaced by α_∞ without much detectable bias. The value E returned by the

basic LogLog algorithm, corresponding to the estimated number of distinct items added to the data structure, is given by the next equation:

$$E = \alpha_m \cdot m \cdot 2^{\frac{1}{m}\sum_j M^{(j)}} \tag{2.16}$$

In the Super-LogLog algorithm, only a portion of the array is used to calculate the number of distinct items. This portion corresponds to the $m_0 = \theta_0 m$ smallest values stored in the array. The constant θ_0 is a real number between 0 and 1, producing near-optimal results when its value is 0.7. The value returned by the Super-LogLog is given by the equation below, where $\sum^* M(j)$ indicates the sum of the values in the selected positions of the array.

$$E = \alpha_{m0} \cdot m_0 \cdot 2^{\frac{1}{m_0}\sum^* M^{(j)}} \tag{2.17}$$

Accuracy The standard error measures, in proportion to the real number of distinct items, the deviation that is expected in the estimated result. An approximation of this value, using the basic LogLog algorithm, is given by the next equation, where σ represents the standard error.

$$\sigma \approx \frac{1.30}{\sqrt{m}} \tag{2.18}$$

Using the improvements of the Super-LogLog algorithm, the accuracy increases. The standard error σ is now given by:

$$\sigma \approx \frac{1.05}{m} \tag{2.19}$$

As the quantity $\frac{1}{m}\sum_j M^{(j)}$ is closely approximated by a Gaussian, the estimate returned by the algorithms are within σ, 2σ and 3σ of the exact number of distinct items with a probability of 65%, 95% and 99%, respectively.

HyperLogLog

The HyperLogLog [22] is an improvement over the LogLog and the Super-LogLog Sketches. The algorithm was developed to estimate the distinct number of elements of a set, while being more memory efficient than its predecessors.

Data Structure The data structure needed is the same as its previous versions: an array of m buckets with $\lceil log_2(log_2(N_{max}))\rceil$ bits each. The m memory units are all initialized with zero. There was a "raw" version of the algorithm that instead of initializing the m memory units with zero, initializes them with $-\infty$. However, for small cardinalities, the algorithm was very inaccurate, since the value 0 was always assumed whenever one of the memory units was not modified.

The hash function needed, h, should map the input items into hashed values whose bits are assumed to be independent and each one to have 0.5 probability of occurring. The second function needed, f, should also be present so it is possible to find the leftmost 1-bit in a binary string (one plus the length of the initial run of 0's).

Methods The algorithm provides the same two methods as the LogLog Sketch: Update(*i*) and the Estimate().

Update(*i*): Whenever an item i arrives this method is called and the h function immediately hashes i. Let $k = log_2(m)$ be the number of bits of the hashed value that determines an index j to a position in the array. The value of j is obtained by adding 1 to the value of the first k bits of $h(i)$. Considering $M(j)$ the value contained by the array in its position j, the value of $M(j)$ in this stage is set to the maximum between its previous value and $f(w)$, where w is the binary representation of $h(i)$ without its first k bits.

Algorithm 3 HyperLogLog

```
1: procedure UPDATE(i)
2:     x = h(i)
3:     j = 1 + ⟨x_1 x_2 ... x_k⟩_2
4:     w = x_{k+1} x_{k+2} ...
5:     M[j] = max(M[j], f(w))
6: end procedure
```

Estimate(): This method returns the estimated number of distinct elements of the data set. In the "raw" version of the HyperLogLog algorithm, the value calculated with the equation bellow, where $\alpha_m \sim 0.72134$ as $m \to +\infty$, is returned directly.

$$E = \alpha_m \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M(j)} \right)^{-1} \tag{2.20}$$

In addition to initializing the memory units with zero instead of $-\infty$, some other improvements were made to the "raw" version of the algorithm. These improvements regard the algorithm's estimate operation, being applied over the value of E, calculated as described above.

Consider E^* the improved estimate. To calculate its value the following rules are applied, in this order:

1. If $E \leq \frac{5}{2}m$, let V be the number of memory units with value 0. If $V \neq 0$, $E^* = m \log(\frac{m}{V})$, otherwise $E^* = E$;
2. If $E \leq \frac{1}{30} \cdot 2^{32}$, then $E^* = E$;

3. If $E > \frac{1}{30} \cdot 2^{32}$, then $E^* = -2^{32} \log(1 - \frac{E}{2^{32}})$.

After that, the value of E^* is found and can be returned, as it represents a good estimation for the number of distinct elements in a set.

Accuracy The standard error to be expected from the estimated values of the HyperLogLog are numerically close to $\frac{1.03896}{\sqrt{m}}$.

The estimates returned by the algorithm are also approximately Gaussian and, for that reason, these values are expected to be within σ , 2σ, and 3σ of the exact count of distinct elements with respectively 65%, 95%, and 99% probability.

Multistage Filters

Multistage Filters [19, 21] is the name of an algorithm used to identify large flows, defined for sending, individually, more bytes than a defined threshold.

Data Structure The data structures used are a "flow memory", which is an array of flow IDs designed to contain the flows that sent more packets than a threshold T, and d arrays (the *stages*) of counters, each one with a different and independent hash function h_j associated.

Methods The sketch provides two methods: the *update* method that is called to every packet that arrives and an *estimate* method, that returns the flows that probably sent more than a threshold of packets.

Update(*i*): Whenever a packet i arrives, the d hash functions compute the flow ID of i. These calculations can be made in parallel. The result of each hash function is a counter, that is then incremented by the size of i. After that, if all counters that are mapped by the functions are above the threshold T, the flow ID of i is finally inserted in the flow memory. This way, the effect of collisions is decreased, attenuating the probability of false positives, as only the flow IDs that maps to counters with values above T at all stages are inserted in the flow memory.

Estimate(): In this algorithm, this operation is very simple as the IDs of the flows that are estimated to be "large" is the content of the flow memory array.

Accuracy This algorithm guarantees that all flows that sent more bytes than the threshold are in the flow memory, as there are no false negatives, only false positives.

Consider the following notation:

- b the number of counters in a stage;

- s the size of a flow (in bytes);
- d the number of stages;
- C the number of bytes that can be sent during the entire measurement interval;
- $k = \frac{T \cdot b}{C}$.

The probability of a flow of size $s < T(1 - \frac{1}{k})$ be inserted in the flow memory is given by:

$$p \leq \left(\frac{1}{k} \cdot \frac{T}{T - s} \right)^d \tag{2.21}$$

2.2 Security in Sketch-Based Monitoring

A major problem that sketch-based algorithms have to face is the limited memory and CPU available. Today's network links operate at very high speeds, decreasing the time budget a switch has to spend with each packet. This constraint has led to monitoring approaches that completely neglect security in favor of ones that minimize the time and space requirements. In some controlled environments this lack of security might be acceptable because threats are limited, but in general it is difficult to assume that no attacks will ever occur. In addition, monitoring activities are often used in the context of network defense applications, such as anomaly detection and intrusion prevention. Therefore, if the monitoring algorithms are insecure then their results may not be trustworthy, what makes their activities worthless or, in a worse case, counter-productive — since corrupted results could lead the network administrator to take inappropriate actions. Therefore, securing the monitoring function is crucial to ensure that the decisions are always adequate.

Every sketch-based algorithm makes use of one or more hash functions. If these functions are not secure, the entire sketch is vulnerable. In the implementation guidelines for the Count-Min sketch [17], for example, the authors say that the hash functions do not need to be particularly strong (as the cryptographic ones are). Some sketch's authors opt not to specify what kind of hash functions are needed and others, like the AMS Sketch's, suggest polynomial ones based on the module operation. A malicious user can exploit this fact to benefit himself, to harm someone else or to simply corrupt the correct operation of the algorithm.

The severity of an attack to a sketch depends on the level of influence the adversary has on the monitored network. We assume the adversary may be anywhere inside the monitored network but that he is not in control of the device where the monitoring solution is deployed. The next subsection tries to describe some vulnerabilities of the sketches, according to adversaries with different capabilities.

2.2.1 Adversary Capabilities

In this subsection, some malicious actions a user can take are described. We assume an adversary that might be anywhere inside the network but that has not compromised the device where the monitoring solution is deployed. All details about the implemented algorithms are known to the adversary, and therefore he may be able to perform the following actions. Depending on his privileges in the network, the malicious user may be able to insert crafted packets into the link, to drop/modify other user's packets or he may be only able to eavesdrop the link. Table 2.1 summarizes the identified attacks that can be made to the sketch-based algorithms we presented in section 2.1.

	General	Algorithm's Dependent
Eavesdrop Only	– Predict the next algorithm's actions – Find potential victims for other attacks	– *Not applicable*
Delay Packets	– *Not applicable*	– *Not applicable*
Drop Packets	– Prevent the algorithm to execute some action	– *Not applicable*
Modify Packets	– Preimage and collision attacks on the hash functions; – Corrupt packet's data	– Overflow the counters – Add negative values to counters – Corrupt hash functions that map items to +1 or -1 – Choose values that the hash function of Virtual Bitmap will map to values outside the flow space covered
Generate Traffic	– Preimage and collision attacks on the hash functions – Overcounting of fragmented packets	– Overflow the counters – Add negative values to counters – Corrupt hash functions that map items to +1 or -1 – Choose values that the hash function of Virtual Bitmap will map to values outside the flow space covered – Adjust the behavior considered normal by the K-Ary algorithm to fit the behavior of a future attack.

Table 2.1: Attacks against sketch-based algorithms

We also inspected the security vulnerabilities of one of the sketch-based algorithms in particular – the Count-Min algorithm. The results are presented in Table 2.2. This table differs from Table 2.1 by presenting the attacks naive implementations of the Count-Min algorithm would be vulnerable to. Just to give an example, adding negative values to the algorithms' counters is an example of an exclusive attack to the Count-Min algorithm. The Count-Min's specification allows this operation since the algorithm was not originally

designed to be used specifically in a network monitoring context.

	Attacks	Examples
Eavesdrop Only	– Predict the next algorithm's actions – Find potential victims for other attacks	By eavesdropping the network the adversary will be able to build up the same data structure as the monitoring entity. He can use that to find out which buckets are not close to the threshold, identifying potential victims for future attacks.
Delay Packets	– *Not applicable*	
Drop Packets	– Prevent the algorithm to execute some action	By dropping packets and using the same data structure as the legitimate monitoring entity, the adversary is able to prevent the monitoring algorithm to take some action just before it does.
Modify Packets	– Preimage and collision attacks on the hash functions – Corrupt packet's data – Overflow the counters – Add negative values to counters	*Preimage*: the adversary can choose a a packet and modify a different packet so that they both hash to the same value. Both packets will increment the same counters. *Overflow*: by overflowing the counters, the adversary prevents the monitoring entity from reading an high value from the counter. *Negative values*: by making negative the field in the packet that will be used to increment the counters, the adversary is able to decremented the counters instead.
Generate Traffic	– Preimage and collision attacks on the hash functions – Overcounting of fragmented packets – Overflow the counters – Add negative values to counters	*Fragmented packets*: the adversary can intercept packets and re-transmit those packets in smaller pieces. This action may trick the monitoring entity into counting each small packet as an individual packet.

Table 2.2: Attacks against the Count-Min algorithm

Eavesdrop only

If the adversary is placed right before the monitoring device, he can fill his own data structure, which will become the same as the built up by the legitimate monitoring task. Knowing the implementation details of the algorithm and possessing the exact same captured information as the monitoring task allows the adversary to predict the actions that

will be taken by the monitoring application. Being able to anticipate what will happen next in the target network may be a tactical advantage to an adversary trying to compromise some participant of that network.

Drop packets

Assuming that the adversary possesses the same knowledge as the legitimate monitoring task, he can drop some packets in order not to trigger a specific event by the algorithm, which could uncover an attack being executed in background. The dropped packets may be chosen in a way that simulates the usual losses of the network, without any suspicious activity.

Modify packets

If an adversary can capture, modify and then replay the packets being transmitted without being noticed, he will be able to corrupt a monitoring algorithm that does not ensure the authenticity of the monitored traffic. The adversary can, for example, modify the packets in such a way that they will collide when the algorithm's hash functions are applied to them. This would cause counters to be incorrect. Depending on the intentions of the adversary, he can attack the "collision-free" property (collision attacks) or the "one-way" property (preimage attacks) of hash functions [26].

Another example attack is the overflow of counters before the monitoring entity reads their values. If an adversary knows the capacity of each counter, he can play with that in order to overflow several counters at the same time. This may be specially destructive if some action is programmed to be taken when a counter is close to its limit [40]. For example, right before counters overflow, their values may be collected and written to a slower memory. Since in normal situations each counter overflows at a different time, the algorithm may not be designed to handle situations in which there are many counters overflowing at the same time. Depending on the algorithm being attacked, the adversary can also subtract the counters' values by passing negative values in the variable to be added.

Some algorithms use a set of hash functions to map items to +1 or -1 at update time and then calculates the median at estimation time. If the adversary corrupts the output of these functions, the estimation returned by the algorithm will not be correct.

Finally, an attack that is specific to the Virtual Bitmap algorithm. As described in the previous section, this algorithm does not cover the entire flow space, using an hash function to select which packets are ignored and which are not. The adversary can find a way to force some packets to hash to a position outside of the covered flow space.

Generate traffic

In the context of the heavy-hitters, for example, assuming that the monitoring task is keeping track of the frequency of each source IP address, the adversary can spoof his IP address to one that, by applying the algorithm's hash function, will collide with an IP address of a legitimate user. By repeating this action, the adversary can trick the monitoring task into thinking that a specific legitimate user is generating more traffic load than he truly is.

In addition to the vulnerabilities that can also be possibly exploited by adversaries with less capabilities, generating traffic allows the adversary to perform other attacks. His capability to craft packets, allows him to craft large IP datagrams that may be fragmented in multiple packets of smaller size by switches. This process helps the adversary to quickly fill specific counters of interest, as it decreases the number of crafted packets needed to achieve that goal. The k-ary sketch is vulnerable to another attack where the adversary corrupts the forecast sketch (built up based on previously observed sketches) by slowly changing the behavior that is considered "normal". To do this, the adversary can craft and inject packets to the network with sizes slowly increasing, starting with packets of a size considered normal. This way, the forecast sketch will eventually become the one the adversary pretends, allowing him to send packets of larger sizes without being detected.

2.3 Programmable Switches

Common switching chips are fixed-function. They run a fixed set of protocols, defined at manufacturing time, and the sort of packet processing available is therefore restricted.

Recently proposed switch chips are finally able to process packets just as fast as the fixed-function chips while allowing the reconfigurability of the data plane. The first proposal of such switches was made by P. Bosshart et al. [12], by means of a reconfigurable match table model that also allows new actions to be defined from a set of action primitives, enabling to specify precisely how packets are processed. Its authors demonstrate the performance penalty of having a reconfigurable match table to be small, in comparison to a fixed one. One of the first proposals to make use of programmable switches is M. Shahbaz et al. [36]. The authors propose a software switch, PISCES, derived from the Open vSwitch (OVS), whose behavior is defined through a high-level language, P4 (explained next). The use of a high-level language allows the customization of the switch forwarding behavior without requiring changes to the software switch implementation. As a result, for the same switch reconfiguration, a PISCES program is orders of magnitude shorter than the equivalent changes required to the OVS source code, while introducing a performance overhead of only about 2%.

2.3.1 P4

P4 [11] is an open-source language maintained by the P4 Language Consortium that allows the programming of reconfigurable switches, such as [12]. Instead of providing only a set of supported protocol headers (like OpenFlow, the main SDN protocol, does), P4 allows the definition of new headers programmatically, while taking advantage of the new reconfigurable switches that are capable of processing packets just as fast as the fixed-function ones. While OpenFlow has to extend its specification in order to support new headers, at the cost of increased specification complexity, a P4 program can define any header and reprogram the switch for this purpose.

A P4 program further defines the adequate parsing sequence for the expected packet headers and the sequence of match+action stages to apply to the parsed packets.

Goals

The following paragraphs describe the three main goals of the P4 language:

- Reconfigurability – The operator is able to redefine how switches process packets after they are deployed.
- Protocol independence – The language is not tied to any specific packet format. Instead, the P4 program specifies a packet parser for extracting the required header fields.
- Target independence – The P4 programmer does not need to know the details of the underlying switch.

Structure

A P4 program is divided in the following components:

- Headers – Definition of the expected packet formats and specification of the header fields name and width.
- Parsers – Extraction of the packet headers based on the parse graph defined.
- Tables – A set of control plane operated Match+Action tables used to process packets. Each table entry is composed of a lookup key and the corresponding set of actions. The P4 program defines the fields the lookup keys are compared to and the actions that may be executed.
- Actions – Definition of the manipulations to be done to the packet fields and metadata when a specific action is applied.

- Control Programs – Sequence of operations that determine the order the Match+Action tables are applied to each packet. For the definition of a control flow, if-else statements may be used in this section.

Architecture

The P4 architecture is illustrated in Figure 2.7. Each packet received as input is processed by a parser, which produces a representation of that packet so it can be matched against the match+action tables. The parser operation follows the parse graph, previously defined.

The match+action tables, populated by the control plane, contain matching data and references to the corresponding actions. When a parsed packet matches a line of a match+action table, the action referenced by that line is applied to that packet. The ingress match+action tables are responsible for the generation of the egress specification. This egress specification determines to which ports the packet will be sent as well as the number of instances of the packet that are sent to each port. When the packet is in the egress pipeline, its physical destination is already determined. The packet may be dropped or have its headers further modified but its destinations does not change.

The queues mechanism, besides processing the egress specification, generates the required packet instances and submits them to the egress pipeline. When an output port is overloaded with too many packets, then they may be temporarily stored in buffers.

When the packet finally leaves the egress pipeline, it is reconstructed from its modified parsed representation and then transmitted by the determined output port.

2.3.2 P4 for Sketch-Based Monitoring

Emerging programmable switching hardware [12] gives an unprecedented level of flexibility to packet processing. Leveraging on the advances brought by P4, recent work has proposed solutions that enable, for the first time, sketch-based algorithms to run in hardware switches.

Liu et al. [30] have proposed UnivMon, a framework that allows universal streaming: a single universal sketch that is shown to be provably accurate for estimating a large class of functions. The solution takes advantage of programmable switches through the P4 language for flow monitoring. Another P4-based monitoring solution is V. Sivaraman et al. [37], which aims to identify heavy hitter flows entirely in the data plane. For this purpose, it uses a pipeline of hash tables to keep track of the heavy flows at each moment. Taking advantage of the ability to keep and manipulate state over multiple packets that programmable switches have, hashpipe makes use of packets to carry results over multiple stages.

As we can see, there are already a few proposals in the network monitoring context that use the P4 language to leverage the flexibility provided by programmable switches. However, contrary to our proposal, none of these works considers security in their design.

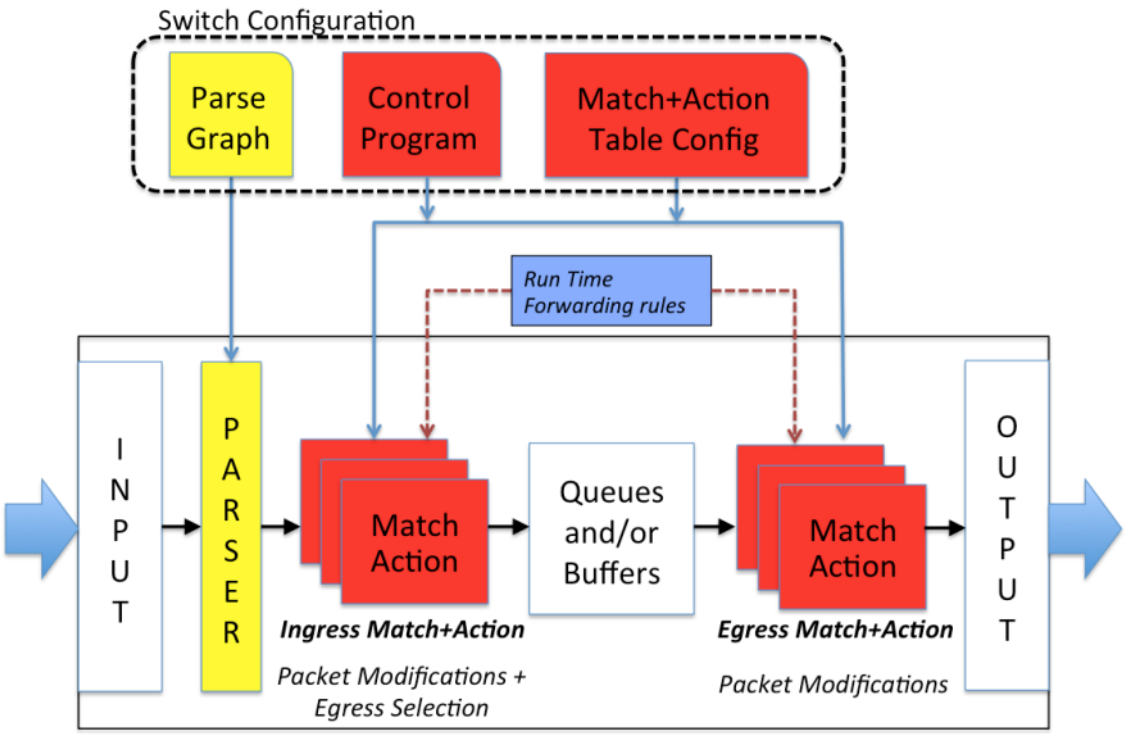


Figure 2.7: P4 architecture (from [6])

2.4 Summary

In this chapter we review the most well-known sketch-based algorithms, describing the data structures each one uses, its operations, and the accuracy guaranteed. Afterwards, we identified the main security vulnerabilities of the generality of sketch-based algorithms, in a network monitoring context. We then focused on the identification of the vulnerabilities of a specific algorithm – the Count-Min. Finally, we introduced the emerging programmable switches, and P4, a language to program them. We concluded by reviewing solutions that leverage the advances brought by P4 in the network monitoring context, but that still have limitations with respect to security.

In the following chapters we propose the design, and provide an implementation and evaluation of a secure version of the Count-Min sketch. In addition to securing the Count-Min algorithm, we also adapt the original algorithm and its secure version so they can be used as network monitoring solutions. For this purpose, we take advantage of programmable switches, through the P4 language.

Chapter 3

Design

In this chapter, the design of a secure version of the Count-Min algorithm (described in section 2.1.1) is presented. The algorithm is implemented in P4 (section 2.3.1) and can be deployed in P4-enabled forwarding devices, to allow efficient and secure traffic monitoring.

Our objective is to take the Count-min algorithm and modify its operation in such a way that it is no longer vulnerable to attacks, but without compromising its accuracy, performance, and simplicity of design. Many of the problems that were identified can actually be prevented if the attacker is no longer capable of predicting the behavior of the algorithm. In particular, if he is unable to guess which entries in the Count-min data structure are modified with the arrival of a packet, then he cannot emulate the algorithm behavior just by observing the arriving traffic. Therefore, an effective way to achieve this goal is to substitute the original hash functions by a fast cryptographic hash function that receives as input also a strong key (128-bits). Since the key is unknown to the adversary, it becomes extremely hard to brute-force in an attempt to create for instance hash collisions. Since network monitoring is often required to be continuously active, we need to provide the possibility to change this key at runtime (i.e. without having to restart the switch). The periodicity of the key change is decided by the network administrator, who should consider the accuracy guarantees of the Count-Min sketch (see 2.1.1). We also address the overflows that could occur. We set an attribute in the P4 definition of the counters, that does not allow increments if that operation will cause an overflow.

3.1 Sketch-Based Monitoring

In our sketch-based monitoring context, there is a controller, responsible for the management of switches and for making the adequate estimations, and the programmable switches, that monitor and forward the traffic.

Figure 3.1 shows the operation sequence followed by switches for each packet they process. As soon as a switch receives a packet, it stores information about that packet

(according to the monitoring algorithm deployed), and only after that, the packet is forwarded.

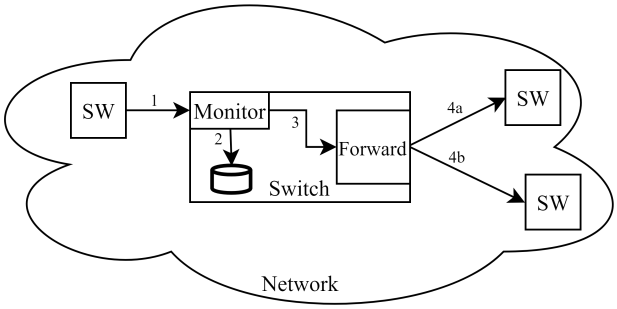


Figure 3.1: Update operation in a sketch-based monitoring solution

In order to monitor the network, the controller can query several switches to make assumptions about the network behavior. Figure 3.2 shows the operator making a query to switches 1, 2, and 3. The order of operations is illustrated in the query to switch 2. As we can see, after receiving a request from the monitor, the switch accesses the data it has stored, and returns the requested information. Before the data is transmitted to the operator, the estimate module makes the required calculations (according to the sketch being used) in order to provide information that makes sense to the operator.

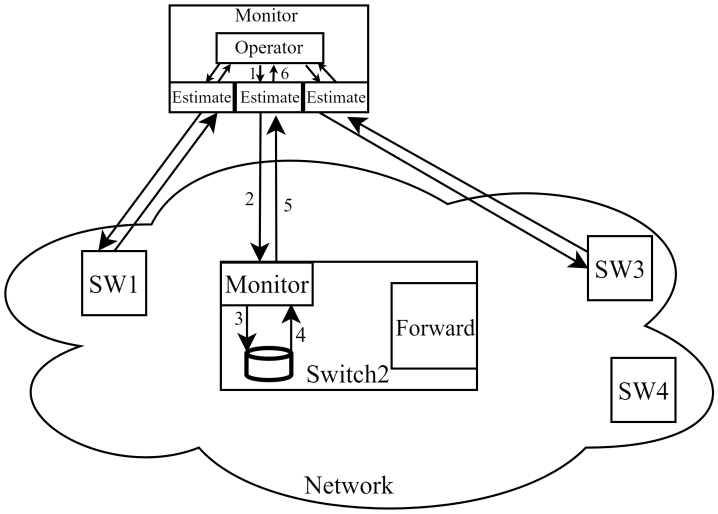


Figure 3.2: Estimate operation in a sketch-based monitoring solution

The design of our solution is illustrated in Figure 3.3. The initialization sequence must be executed first. Algorithm 4 is responsible for initiating the P4 switch. The Update algorithm 6 runs in P4, being executed automatically whenever a packet arrives at the switch. The Estimate 7 and Change Key 5 are independent algorithms, that may be called by the controller at any moment.

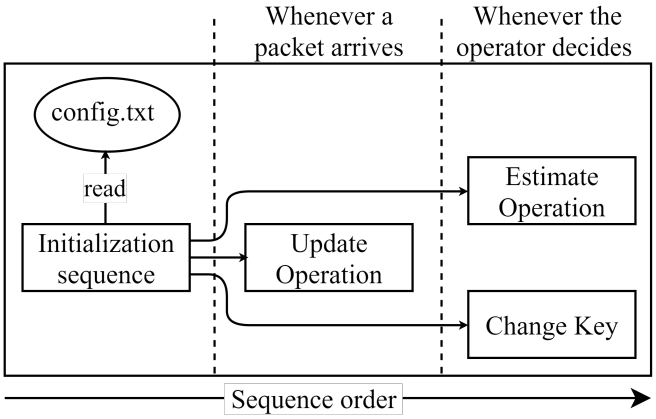


Figure 3.3: Solution design

The following sections describe the algorithms that compose our solution.

3.2 Initialization Sequence

Algorithm 4 Initialize P4 Switch

```
1:  procedure INITIALIZE_SWITCH()
2:      //read the dimensions of the sketch from a configuration file
3:      width, height, slot_size = read("config.txt")
4:      number_of_slots = width × height
5:      last_row = width × (height − 1)
6:      constants = width, height, slot_size, number_of_slots, last_row
7:      //write constants to a header file
8:      write("input_file.p4", constants)
9:      //start a new switch loaded with the p4 program
10:     start_p4_switch()
11:     key = random(2^128 − 1)
12:     //set the key to be used by hash functions
13:     switch.write_register("key_register", key)
14: end procedure
```

Algorithm 4 corresponds to the initialization sequence of the P4 switch. It starts by reading the user's desired sketch dimensions from a configuration file and, based on it, calculates a set of values that are then written to a constants file. Those preprocessed values are used inside the P4 program through the preprocessing directive `#include`, which makes the constants file contents available to the P4 code. In line 10, the P4 switch is started. The following lines show the generation and storage of the key that is going to be used by the hash functions. A 128 bits random number is read from Unix's `/dev/urandom` (line 11) and written to a switch register (line 13) as soon as it is up and ready.

The `width` and `height` constants correspond to the number of columns and rows of the data structure, respectively. The `slot_size` defines the width of each counter in bits. For example, a data structure with $width = 10$, $height = 3$ and $slot\_size = 32$ would take 120 bytes. Since the P4 syntax only allows linear arrays, the bidimensional array required by the Count-Min algorithm is flattened and represented as a linear one with length $width \times height$, which corresponds to the `number_of_slots` constant.

3.3 Change Switch Key

Algorithm 5 Change Key

```
1:  procedure CHANGE_KEY()
2:      //get data structure and key currently being used by the switch
3:      sketch = switch.counter_get_all("counters")
4:      key = switch.read_register("key_register")
5:      //write sketch and key to a file
6:      dump_to_file("old_sketches_file", sketch, key)
7:      //set a new switch key
8:      key = random(2^128 − 1)
9:      switch.write_register("key_register", key)
10:     //reset switch data structure
11:     switch.counter_reset("counters")
12: end procedure
```

Lets call *monitoring period* to the period of time during which the same key and data structure are used by a switch. Algorithm 5 changes the switch key in use, and is executed at the end of each monitoring period. In addition to changing the key, it copies the data structure's content to a different memory before cleaning it. In line 6, the data structure and the key being used by the switch are appended to a file named "old_sketches_file", which may already contain other data structures and keys from previous monitoring periods. When both the data structure and key are written to the file, the algorithm generates a new key, inserts it in a switch register (`key_register`) and cleans its main data structure (`counters`).

3.4 Update Operation

Algorithm 6 is implemented in P4 and can be run by programmable switches. Lines 5 and 7 define the memory structures that persist across packets: `counters` and `key_register`. The main data structure of the algorithm is the array `counters`, defined in line 5. The `key_register` is used to store the 128 bits key required by the hash function.

Line 9 onwards is executed for every packet that arrives. The number of packets/bytes are tracked per `item`, which, in this representation, is the source IP address of the packet.

Algorithm 6 P4 Sketch Update

```
1: // constants gathered during switch initialization
2: width, height, slot_size, number_of_slots, last_row = read("input_file.p4")
3: /* stateful memories that persist across packets */
4: //array with number_of_slots counters, each one with slot_size bits
5: counters = Array[number_of_slots] with slots of size slot_size
6: //register with 128 bits to store the hash_function key
7: key_register = Register with 128 bits
8: /* Executed to every packet that arrives */
9: procedure UPDATE(PACKET)
10:     //item can be any field of the packet header
11:     item = packet.src_ip
12:     key = read_register("key_register")
13:     //points to the start of each row
14:     target_row = 0
15:     while target_row <= last_row do
16:         //set the input of the hash function
17:         hash_input = {item, key, target_row}
18:         //get a column number
19:         target_column = hash_function(hash_input) mod width
20:         //get the position of that column in the current row
21:         target_slot = target_row + target_column
22:         //ensure incrementing the counter does not produce an overflow
23:         if counters[target_slot] < 2^slot_size − 1 then
24:             counters[target_slot] = counters[target_slot] + 1
25:         end if
26:         //point to the next row
27:         target_row = target_row + width
28:     end while
29:     forward_packet(packet)
30: end procedure
```

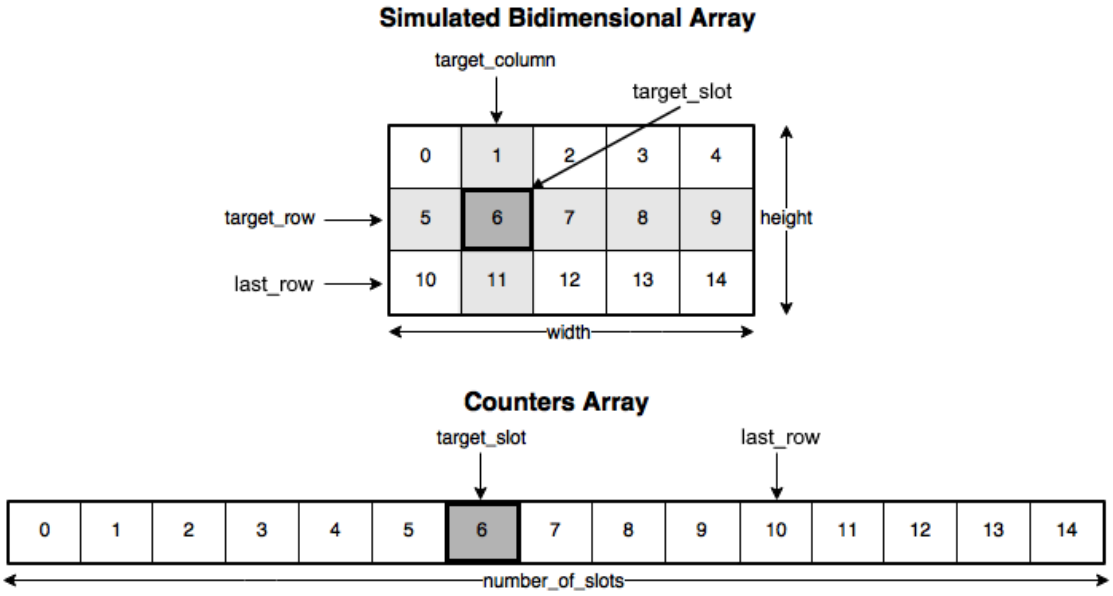


Figure 3.4: Simulated bidimensional array and counter linear array comparison

In line 12, the key previously generated and stored in the `key_register` is read so it can be used by the hash function.

The algorithm calculates an hash function to every row of the data structure. Towards this goal, an iterator-like variable is used (represented in Algorithm 6 as `target_row`) that keeps pointing to a position of the linear array that corresponds to the beginning of each row in the simulated bidimensional array. The `last_row` constant, which points to the beginning of the last row of the sketch, is compared in every iteration with `target_row`, to check if it is already the last iteration.

From line 15 to line 29 we show the operations performed to every sketch row. Line 17 defines the hash function input, function that is calculated in line 20. Here, the modulo operation (represented as `mod`) with `width` is applied to the result of the hash function. This allows the algorithm to find a value between 0 and `width` (exclusive), depending on the result of the hash function. The resulting value, `target_column`, is used as an index to a column of the sketch. The `target_slot` value is the index in the `counters` linear array of the `target_column` in the `target_row` of the simulated bidimensional array. Figure 3.4 illustrates this conversion. After the target counter index has been calculated, its value is incremented if possible, i.e., if it does not produce an overflow.

3.5 Estimate Operation

Algorithm 7 is used to estimate the number of packets/bytes associated with a given `item`. If the P4 program defines `item` as the source IP address of packets, for example, the `Estimate` procedure returns an estimation for the number of packets/bytes received

Algorithm 7 Count-Min Estimate

```
1:  /* Estimates the frequency of item */
2:  procedure ESTIMATE(ITEM)
3:      //file containing all the past data structures and their respective keys
4:      old_sketches_file = "old_sketches.txt"
5:      estimated = 0
6:      for each element in old_sketches_file do
7:          old_sketch = element.get_sketch()
8:          key = element.get_key()
9:          estimated = estimated + estimate_sketch(old_sketch, key, item)
10:     end for
11:     //get the key currently being used by the switch
12:     key = switch.read_register("key_register")
13:     estimated = estimated + estimate_sketch(NULL, key, item)
14:     return estimated
15: end procedure
16:
17: procedure ESTIMATE_SKETCH(SKETCH, KEY, ITEM)
18:     width, height, slot_size = read("configuration_file.txt")
19:     target_row = 0
20:     last_row = width × (height − 1)
21:     result = +∞
22:     while target_row <= last_row do
23:         hash_input = {item, key, target_row}
24:         target_column = hash_function(hash_input) mod width
25:         target_slot = target_row + target_column
26:         //sketch to estimate is being used by the switch
27:         if sketch == NULL then
28:             //read from data structure kept by the switch
29:             row_result = switch.counter_read("counters", target_slot)
30:         else
31:             //read from a stored sketch
32:             row_result = sketch[target_slot]
33:         end if
34:         //calculate the minimum between obtained values
35:         result = minimum(result, row_result)
36:         target_row = target_row + width
37:     end while
38:     return result
39: end procedure
```

from that particular IP address.

When the switch key is changed by Algorithm 5, the data structure being used must be cleaned before the switch restarts its monitoring process to ensure that counters were only modified by the associated `item`(s) in the present monitoring period. Otherwise, there would exist different counters of the same row for different periods, modified by the same `item`. Since it is required the estimations to be based on all monitoring periods, it is necessary to store the data structure being used by the switch before it is cleaned at the end of a monitoring period. From line 6 to 10 of algorithm 7, these past data structures are fetched and an estimation for each one is done. When the estimation to be done by procedure `Estimate_sketch` is based on the data structure currently being used, the algorithm queries the switch for its key and assign it to `key` in line 12. The sum of all estimations, including the estimation based on the data structure currently being used by the switch, is the value returned by the `Estimate` procedure.

In the `Estimate_sketch` procedure, after the `target_slot` has been calculated, if `sketch` is NULL in line 27, then this is the structure currently in use, so the algorithm queries the switch `counters` array for its content in that slot. If it is not, the algorithm simply reads the content of the sketch that was passed as input at position `target_slot`. As for the Count-Min, the minimum of all values is the result returned by the algorithm.

3.6 Management of Old Sketches

Algorithm 5 is executed manually by the network operator. In order to maintain the accuracy guarantees of the main algorithm, as explained in section 2.1.1, it should be executed regularly, to clean the data structure being used. Since the algorithm's accuracy depends on the number of items already monitored, the network administrator can use that number to decide if it is necessary to execute the algorithm.

Besides cleaning the data structure in use, Algorithm 5 also replaces the switch key. The security of the solution depends on this key, that is given as input to the hash function, influencing the counters to increment. To guarantee that this key is kept unknown by the attackers, it should be changed periodically. This way, even if an attacker finds out the key being used, he can only make use of that knowledge during that specific monitoring period. The network administrator is responsible to keep the monitoring periods small enough so it is highly unlikely that an adversary finds the key being used.

For each execution of Algorithm 5, both the data structure and the key are copied to a file, meaning that for each monitoring period the size of the data structure plus the 128 bits of the key is taken from the available storage space. To avoid the devices running out of memory, the network administrator will have to delete some of the old data structures and keys in order to move to new monitoring periods. The old data structures can not be merged because the items (source/destination IP, source/destination port, etc.) responsi-

ble for incrementing the counters are not saved. To merge two different data structures from different monitoring periods (different hash functions), sketches would have to be reversible, to allow the items responsible for incrementing each counter to be obtained. But this is not the case.

In order to keep memory usage limited, one data structure and its respective key may be deleted at the end of each monitoring period or at the end of a monitoring period series. There are a few alternatives:

- Delete the oldest one: This solution deletes the oldest data structure and its respective key from memory at the end of each monitoring period.
- Delete all: This is the simplest solution, where at the end of a set of monitoring periods all data structures and keys are deleted.
- Keep representatives: A network administrator may want to keep some monitoring data from the past but he may not need to keep data from all monitoring periods until the present. This solution suggests to keep only a subset of the data structures that will represent periods of time. An example where this solution works well is if there are events of interest, separated in time, during which it is vital to keep all monitoring data. The data structures recorded between those moments may not be as important and be deleted after a shorter period of time.

It should be noted that estimations based on more than one data structure will not have the same accuracy guarantees as the simple Count-Min algorithm, based on a single data structure. For each stored data structure, an estimation for the given item is done and the final estimation returned by the algorithm is the sum of the individual estimations. This means that the final estimation will have an error of at most the error of the Count-Min algorithm multiplied by the number of data structures the estimation is based on.

Chapter 4

Implementation

The implementation of the algorithm in P4 presented a few challenges, not only because of limitations of the language but also due to the constraints imposed by the interface between a P4 program and the software switch. This section briefly explains how the main obstacles were overcome.

4.1 One-Dimensional Array

The main data structure of the algorithm had to be linearized to an array with a single dimension (instead of a two-dimensional array like in the original algorithm's description). There are two reasons for this modification: (i) P4 does not support multi-dimensional arrays; and (ii) the size of the array has to be specified at start time (no dynamic allocation).

Consequently, to define the desired data structure's size, the user writes a configuration file with the width, height and number of bits in each entry. The P4 code bellow shows the definition of the main data structure used by the sketch, where it can be seen we count packets with a data structure which has `NUMBER_OF_INSTANCES` entries (equal to the $width * height$), each one with `SLOT_SIZE` bits. The attribute `saturating` prevents counters to wrap around by stopping to count if they reach their maximum value (according to P4 version 1.0.3).

```
counter counters{
  type: packets;
  instance_count: NUMBER_OF_INSTANCES;
  min_width: SLOT_SIZE;
  saturating; //prevents overflows
}
```

4.2 Repeat Actions

P4 does not support loops. Therefore, a workaround had to used: (i) the loop is unrolled, creating an if statement per iteration; (ii) the if condition stops processing (becomes false)

when a counter reaches a previously defined maximum value. Below, it is exemplified a loop through the lines of the data structure, which could be executed up two times (depending on the value of `LAST_ITERATION`).

```
if(custom_metadata.target_row <= LAST_ITERATION)
    apply(update_table1);
if(custom_metadata.target_row <= LAST_ITERATION)
    apply(update_table2);
```

Since P4 does not allow to apply the same table multiple times, we had to create distinct tables per iteration. To ensure the desired effect, all these tables are associated with identical actions and memory locations, as exemplified in the next code listing.

```
table update_table1{
    actions{update_row;}
    size: 1;
}
table update_table2{
    actions{update_row;}
    size: 1;
}
```

4.3 MD5 Hash Function

To use a custom hash function we had to add it to the P4 target code. We used the "simple_switch" target from the behavioral-model [1], that restricts the exchange of data between the switch and the P4 program to 64 bits, which is not enough to hold a MD5 hash function output (128 bits). To overcome this, we divided the hash computation in two calls, where one calculates the actual MD5 function and returns the 64 most-significant bits of the result and the other simply returns the 64 less-significant bits (of that same result). The following code shows how the 64 bits returned values are aggregated into a 128 bit field in P4.

```
modify_field_with_hash_based_offset(
custom_metadata.full_hash, 0, hash_p1, MAX);

modify_field_with_hash_based_offset(
custom_metadata.second_part, 0, hash_p2, MAX);

shift_left(custom_metadata.full_hash,
custom_metadata.full_hash, 64);

bit_or(custom_metadata.full_hash,
custom_metadata.full_hash, custom_metadata.second_part);
```

The full_hash is a 128 bits field but only its 64 less significant bits are being used after the first two lines of code. To merge the hash functions (hash_p1 and hash_p2) we start by shifting left 64 bits of full_hash and then we do a bitwise OR with the second_part field, which contains the 64 less significant bits of the MD5 output. After the last line of code the full_hash contains the 128 bits output of MD5.

Constant MAX has the value 2^{64}, which is required by the P4 function modify_field_with_hash_based_offset to apply a modulo operation to the result of the hash function. This way we guarantee the modulo operation has no effect when applied to the hash functions result since hash_p1 and hash_p2 return 64 bits each.

4.4 Define Flows

A flow is defined by the tuple (protocol, source IP address, destination IP address, source port, destination port). Although in chapter 3 we exemplified our solution only by counting packets based on the source IP, our solution is generic and can be used with flows, or other combination of headers. To count based on TCP flows, we can not simply hash those four fields, as traffic exchanged between the same applications, in opposing directions, would be classified as belonging to different flows. As such, we applied a bitwise exclusive OR between source and destination IP addresses, and between source and destination ports, to correctly identify the flows. Fields xor_ips and xor_ports are then used as input of the hash function, instead of the individual source IPs and ports.

```
bit_xor(custom_metadata.xor_ips, ipv4.srcAddr,
ipv4.dstAddr);
bit_xor(custom_metadata.xor_ports, tcp.srcPort,
tcp.dstPort);
```

Chapter 5

Evaluation

We carried out two sets of experiments to evaluate the performance of our algorithm. The first group measured the latency and throughput of the secure count-min sketch, while the second set tested the error estimations in several settings.

5.1 Experimental Setup

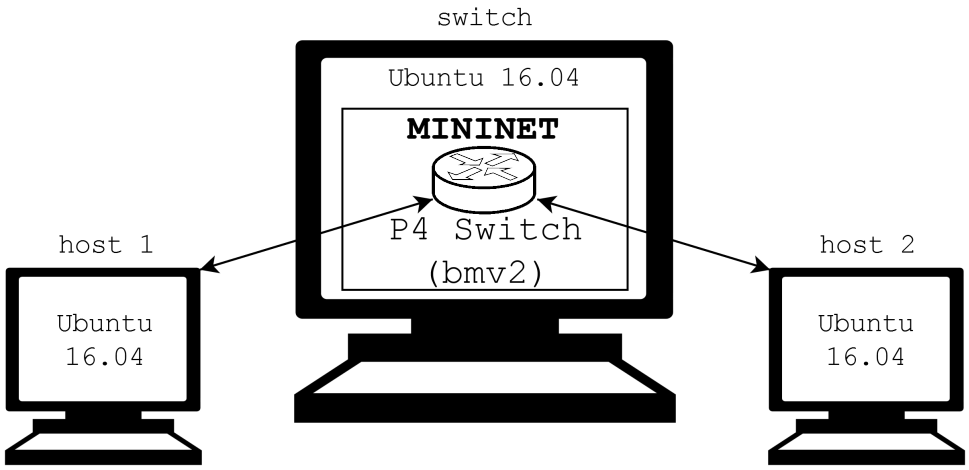


Figure 5.1: Network Topology

As illustrated in Figure 5.1, the testbed was composed of three machines: one emulating a P4 switch inside a mininet [5] instance, and the other two acting as simple hosts that send and receive traffic (host 1 and host 2). The switch machine has an Intel(R) Xeon(R) CPU E5-2407 v2 @2.40GHz and 64GB of RAM, host 1 has an Intel(R) Core(TM)2 Duo CPU E7400 @2.80GHz and 4GB of RAM, and host 2 has an Intel(R) Core(TM)2 CPU 4400 @2.00GHz and 2GB RAM. The three machines were connected by 1GB/s links.

The virtual switch used was the second version of the P4 software switch known as behavioral-model (bmv2) [1]. It was loaded with our P4 monitoring algorithm which enabled the switch to monitor and forward the traffic received.

5.2 Performance of Traffic Forwarding

To evaluate the proposed solution, we compared it against two other P4 programs: (i) the original Count-Min algorithm, used as baseline; and, (ii) a program that simply forwards traffic, to understand the cost of monitoring.

As explained previously, the size of the data structure of the count-min sketch allows to trade accuracy for overhead, as for each additional line in the data structure there should be a performance degradation due to the computation of an extra hash function. Therefore, we provide experimental results for different number of lines in the data structure. Like section 2.1.1 demonstrates, a data structure with 20 lines guarantees, with a very high probability, that the error does not exceed its maximum theoretical value.

5.2.1 Latency

To measure the delay introduced by the switch, we calculated the average round trip time (RTT) of 10000 pings between the two end hosts. Figure 5.2 shows the observed average latency and standard deviation for the three P4 programs.

As expected, the forwarding program got the smaller RTT values, with a constant latency of around 500 microseconds. The secure version of Count-Min performed worse than the original algorithm, with an average overhead of around 10%. The difference between them was of approximately 40 microseconds for a sketch with one line, and around 160 microseconds for a sketch with 20 lines. This is an interesting result because it demonstrates that with a relatively small increase in the overhead, it is possible to offer increasingly low error probabilities (see section 2.1.1). Of course, if we could have had access to hardware support for P4, one should see a significant decrease on these values.

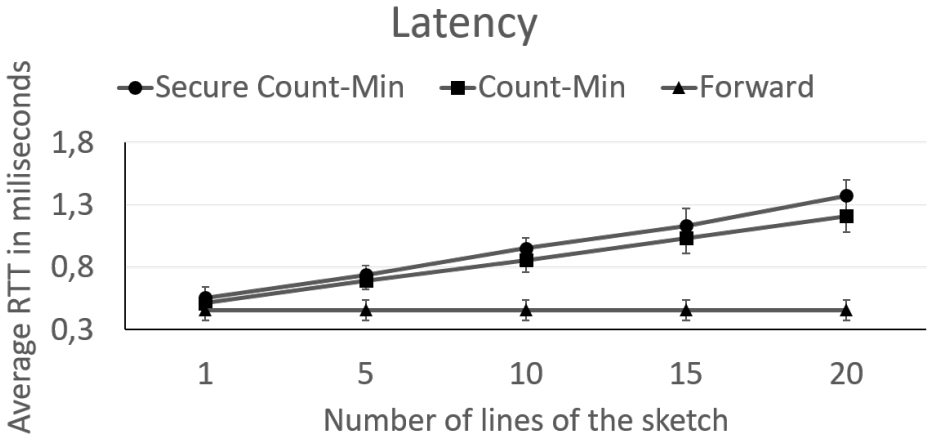


Figure 5.2: Latency between the two hosts

5.2.2 Throughput

The throughput was measured with *iPerf* [4] between the two nodes. In order to obtain the maximum throughput, the traffic rate of iPerf was increased until the network started to drop packets (i.e., loss rate > 0). Although each experiment was repeated 20 times, the calculated standard deviation was not big enough to be observable in Figure 5.3. Between the iPerf client and server, executed by the hosts, was only the switch machine loaded with our P4 algorithm.

The forward-only solution got the best results, as expected, achieving a throughput of around 159 Mbits/second. The performance cost imposed by adding security to Count-Min was on average of about 5 Mbits/second (i.e., around 7%). Our secure Count-Min algorithm achieved a throughput of 112 Mbits/second with a data structure with 1 line. With 10 lines, the throughput drop was to 44 Mbits/second, and to 27 Mbits/second with 20 lines.

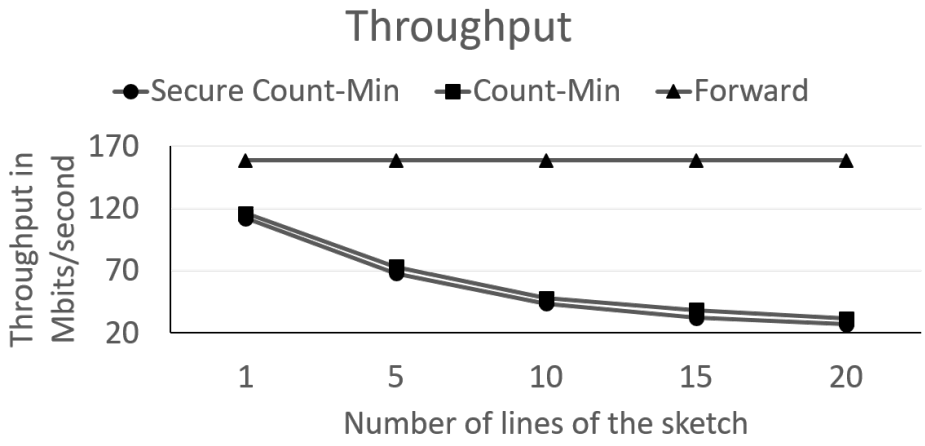


Figure 5.3: Switch Throughput

5.3 Observed Errors in Estimations

The error associated with the estimations returned by our secure sketch was also evaluated. We used a five minute trace of IPv4 traffic captured on a busy private network's access point to the Internet. We injected this set of well known packets in our network so the switch could process them all. Then, for each distinct monitored item (*source IP* or *flow identifier*, as we will explain next), we queried the sketch for that item's estimated frequency. The difference between the estimated frequency and its true frequency is the estimation's error for that item.

The goal was to compare how different data structure dimensions would affect the errors observed in estimations of two monitoring conditions — count packets by sender

(i.e., source IP address) and by flow (i.e., 5-tuple with source/destination ports and IPs for TPC connections). Since there are many more distinct flows (22310) than source IP addresses (1845), we expect to see larger errors when monitoring by flow for a given memory size. The number of lines of the data structure was fixed to 10, since it would only affect the probability of the error and not its extent. To test different memory usages, we used data structures with increasing numbers of columns, starting from 25 (1 KB), 100 (4 KB), 200 (8 KB), 400 (16 KB), 800 (32 KB), 1600 (64 KB) and 3200 (128 KB). Each individual counter occupies 32 bits.

5.3.1 Source IP Addresses

In this case the monitored item was the source IP address of 791179 packets. Calculating the estimations error of all items allowed us to find the minimum and maximum error, and the percentiles 10, 50 (median) and 90 for each experiment, as displayed in Figure 5.4.

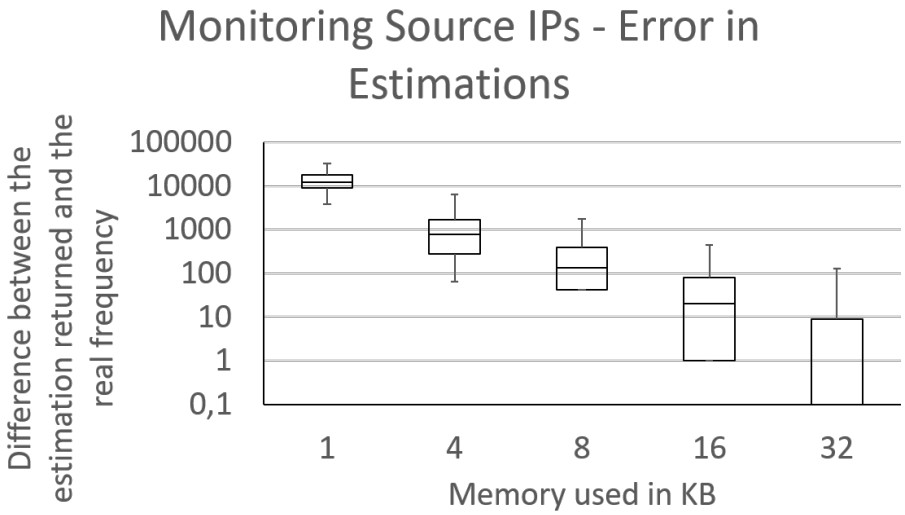
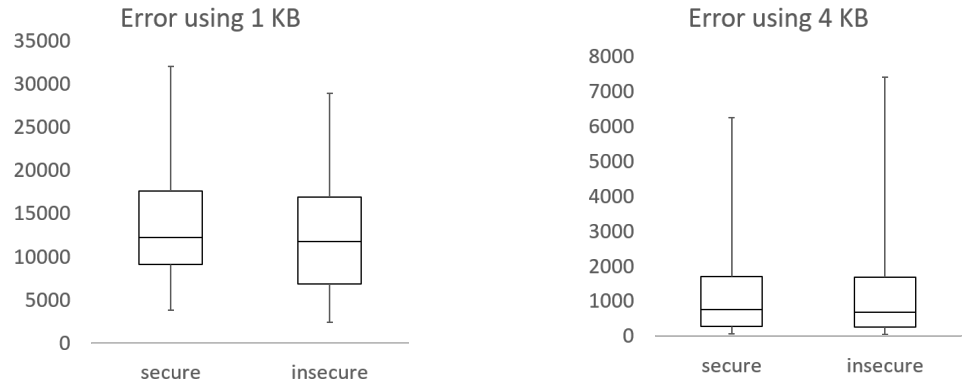


Figure 5.4: Errors in estimations using different memory sizes when monitoring by source IP address with our solution

It was observed that the error decreases as the data structure's size increases. For a data structure with width 25 (1 KB if its height is 10 and each counter occupies 32 bits) the median of the errors was around 12000, which may not be tolerable. However, for a data structure with just 400 columns (16 KB required), the sketch could already achieve relatively small errors, with a median of around 20. Finally, by using 32 KB of memory, the median of the errors was 0.

We also tested the functionality of the original Count-Min algorithm in order to compare its results against our solution. As Figures 5.5a and 5.5b show, the errors in the estimations returned by both algorithms were very similar (largest observed difference was around 2500).



(a) (b)

Figure 5.5: Errors in estimations returned by the secure and the original versions of Count-Min when monitoring by source IP address. (a) using 1 KB of memory (b) using 4 KB of memory

5.3.2 TCP Flows

We also monitored the traffic per flows, where the total number of relevant packets sent through the monitoring device was 633746 (corresponding to all TCP packets).

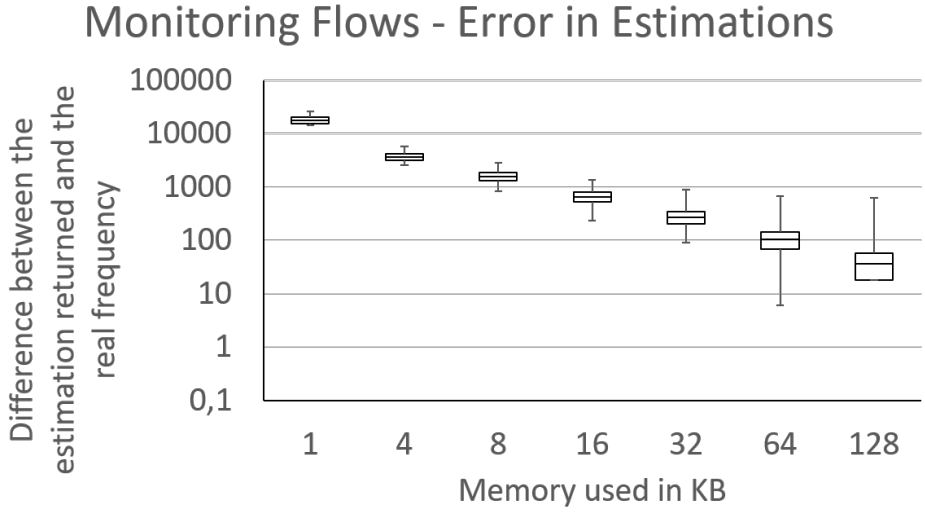
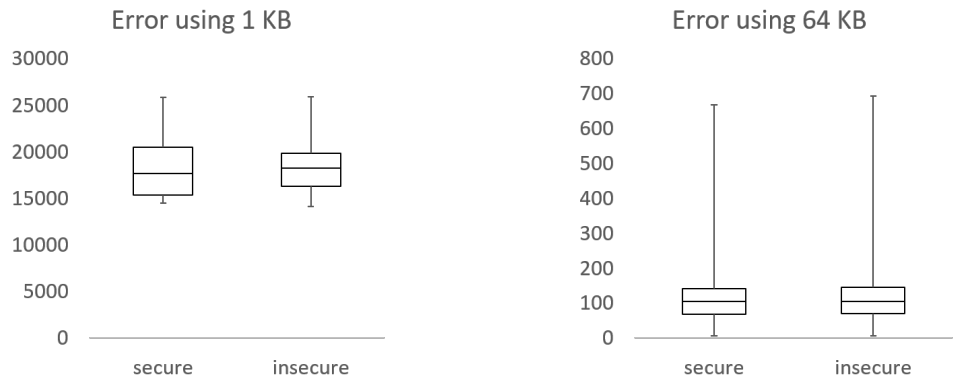


Figure 5.6: Errors in estimations using different memory sizes when monitoring by flow with our solution

Figure 5.6 shows that a data structure with 25 columns (1 KB) leads to a median of errors around 18000. As the memory used increases, the error in estimations decreases, with a median of 646 when 16 KB of memory was used. We also tested our solution using 128 KB of memory, where the median of the errors was only 36.

Again, we tested the functionality of the original Count-Min algorithm when monitoring by flow and compared its performance against our secure Count-Min version. Again,



(a) (b)

Figure 5.7: Errors in estimations returned by the secure and the original versions of Count-Min when monitoring by flow. (a) using 1 KB of memory (b) using 64 KB of memory

the errors returned by both algorithms were very similar (largest observed difference was around 1000), as we shown by figures 5.7a and 5.7b

5.4 Summary

The overhead of securing the Count-Min algorithm turned out to be relatively small, in terms of latency (about 10%) and throughput (7%). The solution that only forwards traffic (without monitoring it) performed much better than the monitoring solutions, as expected. With respect to errors in the estimations, we observed they decrease as the memory used increases, again as expected. Importantly, we observed there was no difference between the extent of errors in estimations returned by the original Count-Min and by its secure version.

Chapter 6

Conclusion

In this thesis our overarching goal was to improve the security of network monitoring. We focused on sketch-based solutions, and performed a survey about some of the best known algorithms. After describing their data structures, operations and accuracy, we explored their main security vulnerabilities, considering adversaries with different capabilities. We have chosen the well-known Count-Min algorithm as target of our work, due to its flexibility to be adapted to the context of network monitoring.

We proposed a secure version of this sketch-based monitoring algorithm. Our solution involves using cryptographic hash functions and adapting it to the context of programmable switches. We also include techniques for the solution to be used continuously, since in its original version it is only possible to monitor something during a limited period of time, after which the data structure becomes overloaded. For this purpose, we developed an auxiliary algorithm that is able to copy the data structure to a different memory, and reset it in runtime, i.e., without having to pause the monitoring algorithm.

Our secure Count-Min prototype was implemented in P4, which allowed us to take advantage of the programmability provided by emerging reconfigurable switches. Despite being presented with some challenges due to P4 peculiarities and the software switch used, we were able to overcome all of them.

To evaluate the performance of the algorithm proposed, we compared it with the original version of the Count-Min and a forward-only solution. We were able to demonstrate, within the P4 software switch limitations, that using a secure version of the Count-Min algorithm does not introduce relevant performance penalties in terms of latency and throughput, in comparison to the original version of the Count-Min algorithm.

Bibliography

[1] Behavioral-model - second version of the p4 software switch. https://github.com/p4lang/behavioral-model. Accessed: September 2017.

[2] Gcreep: Google engineer stalked teens, spied on chats. http://gawker.com/5637234/gcreep-google-engineer-stalked-teens-spied-on-chats. Accessed: September 2017.

[3] Introduction to cisco ios netflow - a technical overview. http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html. Accessed: September 2017.

[4] Iperf3. http://software.es.net/iperf/. Accessed: September 2017.

[5] Mininet - an instant virtual network on your laptop (or other pc). http://mininet.org/. Accessed: September 2017.

[6] $P4_{14}$ language specification. https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf. Accessed: September 2017.

[7] Sampled netflow. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html. Accessed: September 2017.

[8] Selecting a suitable packet sampling rate. http://blog.sflow.com/2009/06/sampling-rates.html. Accessed: September 2017.

[9] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 20–29, 1996.

[10] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[11] 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*, 44(3):87–95, 2014.

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

[13] G. Box, G. Jenkins, G. Reinsel, and G. Ljung. *Time series analysis: Forecasting and control*. John Wiley & Sons, 2015.

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

[15] G. Cormode. Sketch techniques for approximate query processing. *Synposes for Approximate Query Processing: Samples, Histograms, Wavelets and Sketches, Foundations and Trends in Databases. NOW publishers*, 2011.

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

[17] G. Cormode and S. Muthukrishnan. Approximating data with the count-min sketch. *IEEE Software*, 29(1):64–69, 2012.

[18] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617, 2003.

[19] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 2003.

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

[21] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. In *Proceedings of the International Conference on Very Large Data Bases*, 1999.

[22] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the International Conference on Analysis of Algorithms*, 2007.

[23] P. Flajolet and G. Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.

[24] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Quicksand: Quick summary and analysis of network data. Technical report, 2001.

[25] I. Good. C332. Surprise indexes and p-values. *Journal of Statistical Computation and Simulation*, 32(1-2):90–92, 1989.

[26] P. Hoffman and B. Schneier. Attacks on Cryptographic Hashes in Internet Protocols. RFC 4270, 2005.

[27] M. Kandias, N. Virvilis, and D. Gritzalis. The insider threat in cloud computing. In *International Workshop on Critical Information Infrastructures Security*, volume 6983, pages 93–103. Springer, 2013.

[28] D. Kreutz, F. Ramos, P Verissimo, C. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

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

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

[31] R. Los, D. Shackleford, and B. Sullivan. The notorious nine cloud computing top threats in 2013. *Cloud Security Alliance*, 2013.

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

[33] R. Medeiros, R. Fonseca, E. Vial, F. Ramos, and N. Neves. SDN-based Network Monitoring under Attack. Personal Communication, 2016.

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

[35] S. Scott-Hayward, G. O'Callaghan, and S. Sezer. SDN security: A survey. In *IEEE SDN for Future Networks and Services*, pages 1–7, 2013.

[36] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 525–538, 2016.

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

[38] M. Thorup and Y. Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 615–624, 2004.

[39] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, pages 29–42. USENIX Association, 2013.

[40] Q. Zhao, J. Xu, and Z. Liu. Design of a novel statistics counter architecture with optimal space and time efficiency. *ACM SIGMETRICS Performance Evaluation Review*, 34(1):323–334, 2006.