University of Amsterdam

Research Project 1

Exploring Programmable Network Techniques for Data Plane Optimization

Benchmarking Heavy Hitter Detection Algorithms in a Custom Testbed

January 23, 2026

Students:
Tomás E. Agata
15767647

Muhammad Mansour
15294234

Tutor:
Dr. Marios Avgeris

Course:
Research Project 1

Course code:
53841REP6Y

Abstract

This study explores **heavy-hitter detection** in programmable networks by comparing three relevant algorithms, **Count-Min Sketch**[1], **Bloom Filter**[2], and **PRECISION**[3]. As an additional contribution, a flexible benchmarking testbed was developed using virtualization and BMv2 to provide future researchers with a high degree of replicability and evaluation performance under varying traffic conditions. Results show that PRECISION offers superior accuracy and memory efficiency, making it an ideal choice for most scenarios, while Count-Min and Bloom Filter fit to more specific scenarios where false negatives are not allowed. On the other hand, the benchmark testbed proves be useful for the tested environments; however, a hardware implementation and greater development is needed to verify its usability.

1 Introduction

Rapid development of network technologies pertinent to 5G and next-generation networks (6G and beyond), has created large demand for scalability, adaptability, and immediate or real-time responsiveness. Traditional network architectures struggle to meet these needs, leading to the rise of programmable networks as a solution. Technologies such as Software-Defined Networking (SDN) and programmable data planes enable network operators to flexibly adjust and optimize traffic flows, enforce policies, and deploy smart algorithms [4].

One application in programmable networks is identifying heavy hitters, which refer to flows or devices that produce an unusually high volume of traffic. Recognizing heavy-hitters is crucial for many reasons: enhancing bandwidth efficiency, identifying unusual behavior such as distributed denial of service (DDoS) attacks, and effectively managing high-priority services (as seen in Figure 1). They can originate from standard user-facing services caused by a peak in demand or can come from unusual activities started by malicious actors in the form of DDoS attacks [5].

Network programmability is a promising technology that aims to solve problems like these by allowing a developer to specify the behavior of a network through code. An example of these types of networks can be achieved by the use of programmable network switches (such as Intel Tofino[6], Broadcom Trident/Tomahawk[7], BMv2[8], and Cisco Silicon One[9]), which allow for developing efficient applications that are executed every time a packet arrives. As a result of this

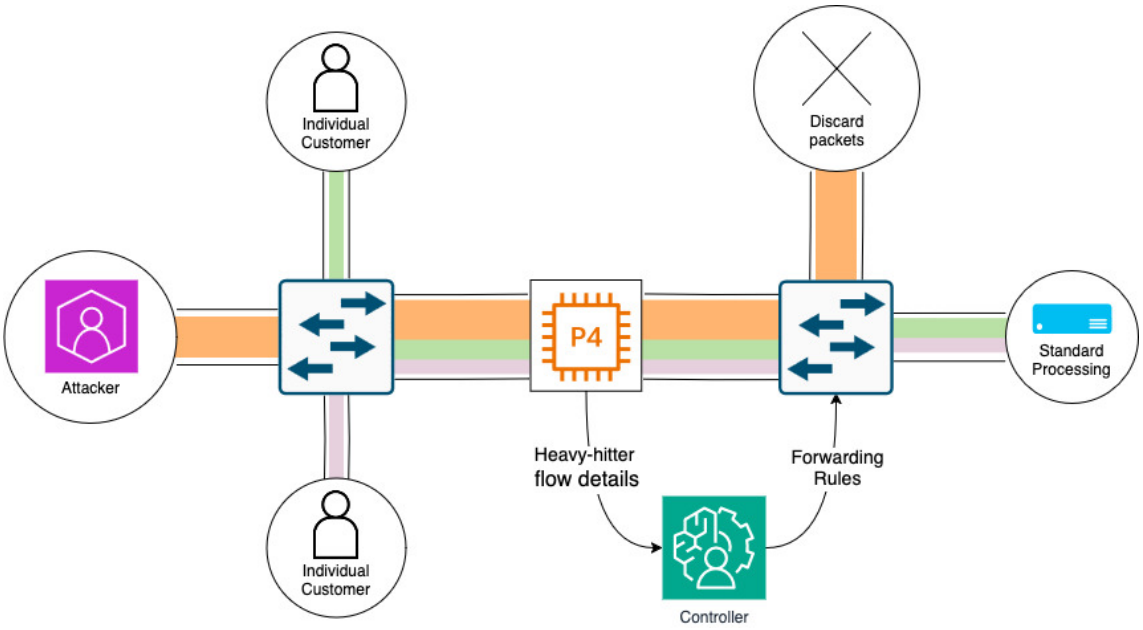

Figure 1: Example cases of the utility of Heavy-Hitter detection algorithms.

flexibility, multiple approaches have been developed to achieve the same goal, each of them with its own advantages and disadvantages. The integration of heavy-hitter detection methods into the programmable data plane is beneficial for ensuring the reliability and security of networks, allowing them to manage the requirements of newer technologies and various applications.

As with any new technology being developed, new challenges arise; one of these comes from the way benchmarks are performed. A common way to create these tests is to set up a simple experimental testbed using elements already obtained by the research team. This might include previously generated software or hardware for other experiments that is tweaked to fit the needs that arise in this new experiment. Although this way of doing things is suitable when documenting the entire experimental setup, achieving full precision in all aspects can be difficult as there might be undiscovered biases caused by testbed implementations.

Another problem that arises from in situ experimental testbeds is that they might be difficult to replicate, as they can involve specialty hardware or NDA-protected documentation for the elements involved. As a result, obtaining the necessary elements to retry the experiment could turn from a difficult reality to an impossible wish.

Our initial concept for this report was to explore the state-of-the-art of heavy-hitter detection and create a fair comparison between two major algorithms while analyzing their accuracy in different scenarios. It was only when we were exploring the different implementations that we discovered that there was *a need for a standard benchmarking with clear documented advantages and disadvantages.*

As a result, the following project will not only focus on the comparison of *two* major heavy-hitter detection algorithms (Count-Min Sketch [10] and Bloom Filter [11]), but it extends our original vision to include the development of a documented benchmark testbed for these algorithms with clear advantages and limitations in order to provide future researchers with clear guidelines and a simplified experimentation environment if they choose to use it. In addition, it will also be used to serve as a testbed for another promising algorithm called PRECISION [3].

On the original proposal for this project we defined the following research questions:

- *What are two modern representational techniques for implementing heavy-hitter detection in programmable networks?*
- *How do these techniques compare in terms of performance and reliability?*

However, after focusing deeper on the research methods to answer these questions, we came up

with complementary subquestions that help clarify the contributions of this RP:

- *What are two modern representational techniques for implementing heavy-hitter detection in programmable networks?*
  - *What insights can we derive about their suitability in different settings/criteria/applications?*
- *How do these techniques compare in terms of performance and reliability?*
  - *How can an experimental testbed be designed to ensure fair, and accurate comparisons?*
  - *What are the main challenges in developing such a testbed?*
  - *What insights can we derive about their suitability in different settings/criteria/applications?*

The rest if the report is structured as follows. Section 2 will focus on the related works used to serve as a guide for this project. Section 3 will explain the methodology used to create the testbed along with our objectives, our design decisions to meet these objectives, why did we choose to compare the aforementioned algorithms and the methodology that we used for testing them. Section 4 explains the results obtained from the benchmark. Section 5 discusses some of the benefits and limitations of the benchmark, while section 6 and 7 conclude the report and discuss future extensions to it.

2 Related works

Research around the development of tools for detecting (and subsequently mitigating) network disruptive phenomena such as heavy hitters has been fueled by the recent advancements of programmability in networks. Kianpisheh et al. [4] presented a survey of the general state-of-the-art on programmable networks, enabled by languages like P4[1] and frameworks such as Software Defined Networks (**SDN**), including its challenges, approaches and solutions. The survey highlights the increasing importance of in-network computing in domains like 5G/6G networks, machine learning, and Network Function Virtualization (**NFV**). It provides a very complete explanation of the current topics that will be addressed in this project as well as their impact on real-world scenarios.

From all of the presented topics, we focused on the efficient detection of heavy-hitters (**HH**), as it is essential in modern networking to manage traffic, avoid congestion, and reduce security threats such as Distributed Denial of Service (**DDoS**) attacks. Various ways and algorithms have been suggested to address the issues of accuracy, resource efficiency, and scalability in terms of heavy-hitter detection and flow monitoring.

Sivaraman et al. [12] presented HashPipe, an algorithm created to detect heavy hitters entirely within the data plane of programmable switches. Using a pipeline of hash tables, HashPipe dynamically removes lighter flows while keeping heavier ones, enhancing limited switch memory and processing resources. Techniques such as rolling minimum tracking and feedforward processing allow the algorithm to identify top-k heavy hitters with more accuracy. HashPipe showed high scalability and efficiency, realizing significant accuracy improvements with minimal memory usage.

Ben Basat et al. [3] introduced the PRECISION algorithm, which improves heavy-hitter detection through partial recirculation, allowing unmatched packets to reenter the processing pipeline. Combining a pipeline design with probabilistic recirculation of unmatched packets it only has to recirculate a small portion of packets. This method improves accuracy and reduces memory usage compared to HashPipe. PRECISION also extends capabilities to Hierarchical Heavy-Hitters (**HHH**) detection, enabling detection of traffic-heavy subnets, and detecting groups of flows that together consume significant traffic, which is important to detect DDoS attacks. The algorithm balances resource efficiency and high throughput.

Rodrigues and Verdi [13] addressed the difficulties of identifying heavy-hitters in multi-pipe programmable switches, where separated pipes can lead to fragmented flow counts across pipes. They suggested two methods: *Local-pipe* approach, where each pipe reports independently to the control plane, and *Accumulator* approach, which aggregates flow counts across pipes. Both

[1] Programming Protocol-independent Packet Processors (P4), `https://p4.org/`

methods utilize adaptive thresholds to flexibly modify detection thresholds. The *Accumulator* approach achieves a considerable decrease in communication overhead with the control plane, which make it more scalable.

Heseding et al. [5] presented SAFFIRRE, a system aimed at reducing DDoS attacks through a combination of Hierarchical Heavy-Hitter algorithms and machine learning methods. SAFFIRRE uses Aggregate Composition Assessment (**ACA**) to estimate attack traffic ratios to the whole traffic and Filter Rule Refinement (**FRR**) to modify filtering rules in real-time. This guarantees accurate filtering of attack traffic while maintaining legitimate flows. Having a false positive rate of only 0.024%, SAFFIRRE offers an efficient approach for real-time DDoS mitigation in dynamic network environments.

Cai et al. [14] introduced cReFeR, a flow monitoring framework for Software-Defined Networks that achieves a balance between accuracy and efficiency in detecting resource-critical flows. The Compression Report-Feedback-Report (cReFeR) system minimizes the size of flow monitoring data using compression techniques such as the IP Compressor and Value Compressor. These methods allow for a 40% reduction in monitoring data volume while keeping an error rate below 3%. This method is effective in identifying heavy-hitters and tackling DDoS attacks.

An extensive review of publicly accessible P4 projects on GitHub was performed in order to acquire insight into the existing implementations of heavy-hitter detection in programmable data planes. We believe it should be noted that the quantity of such projects is limited, suggesting that this domain remains an available area for additional research and advancement. Table 1 provides a summary of the identified P4-based heavy-hitter detection repositories. The table includes the detection algorithm used in each project, the targeted platform (such as software simulation using BMv2, and Tofino-based architectures), and a link to the repository. As a side-note, relevant studies that are directly linked to the project were placed on the "Reference" column of the table. This table can act as a guide for researchers and developers interested in investigating, enhancing, or building on the current heavy-hitter detection mechanisms in P4.

Algorithm	**Target Platform**	**Repositories**	**Reference**
Bloom filter	Software	[15] [19] [20] [21] [22]	[2], [16]–[18]
Count-Min Sketch with Identifier Sampling (CMSIS)	Tofino 2	[23]	[24]
Count-Min Sketch	Software Tofino	[25] [29]	[1], [26]–[28]
HH-IPG	Tofino	[30]	[31]
HashPipe	Software	[32] [33]	[12]
Voting-based	Tofino and Tofino 2	[34]	[35]
dSketch	Hardware-based	[36]	[37]
PRECISION	Software	[38]	[3]

Table 1: Reviewed P4 Projects for Heavy Hitter Detection

Our initial idea was to compare two representative algorithms on the matter. However, when analyzing other researchers' projects to see how they benchmarked their algorithms, it was clear to us that there was not a clear baseline for reporting the accuracy of a heavy-hitter detection. Some of them used software virtualization to emulate a topology, others used NDA-protected hardware implementations using real-life Tofino switches or ASICS, while others simply reported their results of an implementation on a C/Python program. Using this information, we deduced that no standard testing environment exists for testing their performance; as a result, we decided to try our hands at developing one of our own.

However, despite our best efforts, we could not design a testbed that accommodates all possible configurations from the start, as we cannot foresee future implementations or designs. So, in order

to limit the scope of this project to fit our time constraints, we decided to follow Agile [39] development methodologies to create a flexible and extensible testing environment. Our approach prioritizes adaptability, ensuring that our testbed can evolve alongside emerging algorithms and methodologies. By breaking down our development into manageable iterations, we can focus on refining core functionalities first, then gradually incorporating support for additional configurations as needed. This way, we ensure that our benchmarking framework remains relevant and applicable across different testing environments, while also allowing us to make incremental improvements based on feedback and newly available technologies.

Agile methodologies are inherently iterative, which means that they focus on continuous cycles of development, feedback, and improvement rather than a linear, step-by-step process. This iterative nature helps teams adapt to changes, incorporate stakeholder feedback, and deliver working products in small incremental steps. This helps in coordinating the addition of support for new types of algorithm in an incremental way. One of the many implementations of Agile is Kanban[40].

Kanban follows a continuous flow-based development process in which tasks move through a visualized workflow on a Kanban board, typically with columns like To Do, In Progress, and Done. Team members pull tasks when they have capacity rather than being assigned work, ensuring steady flow and reducing bottlenecks. Work-in-progress (WIP) limits prevent overloading, encouraging focus and efficiency. Unlike Scrum, Kanban does not use fixed iterations; instead, work is delivered as soon as it's ready, promoting continuous delivery and quick feedback. Teams regularly analyze cycle time, lead time, and bottlenecks to optimize efficiency and improve the process over time. As we had a clear deadline for the duration of this project and no clearly defined roles on our team, we believed this was a good fit for us.

In order to try our hands at making full use of this methodology, we also decided to add a third algorithm to the project. In this way, we can investigate the current limitations of our testbed and try to provide a more complete benchmark testbed as a result of the iterative Kanban development process.

3 Methodology

In order to provide a consistent experimentation platform that would accommodate our goal of benchmarking the Heavy-Hitter detection algorithms, we identified the following objectives for which the testbed should account for:

Hardware-agnostic environment: Since acquiring application-specific hardware to test the different algorithms can be difficult to achieve and set up correctly, this objective ensures that the research contributes an accessible, open and verifiable testing framework. By designing a hardware-agnostic testbed, others can benchmark their new algorithms or reproduce and verify existing experiments without requiring specific hardware.

Replicable traffic parameters: The performance of heavy-hitter detection algorithms depends significantly on the nature of the traffic they process. Allowing fine-grained control over traffic generation ensures that different scenarios (e.g., varying packet rates, burstiness, and flow distributions) can be systematically tested, leading to a comprehensive comparison.

Unified performance reporting: This objective ensures that the benchmarking process captures relevant accuracy indicators while maintaining consistency in format and metrics. Consequently, these metrics can be used to facilitate a comprehensive evaluation of each algorithm's strengths and weaknesses under varying conditions.

3.1 Design

To achieve our objectives, we decided on the following design choices.

Replicability and hardware-agnostic design To make sure experiments can be replicated and are not reliant on particular hardware limitations, we used software virtualization in order to emulate the underlying switch internals. To achieve this, we created the network topology using Mininet[2] as it offers a lightweight, virtualized networking environment that enables us to mimic diverse network topologies and settings while ensuring portability across hardware.

Replicable traffic generation To enable controlled experimentation, we allowed recreating traffic from a pre-generated packet capture file. This packet capture is essentially a recording of data packets that traveled through a real network in a specific point in time. By replicating the packets in this recording, it ensures that experimental results can remain consistent across tests, allowing for reproducible environments and accurate performance comparisons.

Unified performance reporting In order to accurately report the performance of the selected algorithms, we had to classify them into two different groups: stateless and stateful; each of them with their own internal detection report mechanism. Once having defined this, we developed a custom topology that allows seamless integration of both classes of algorithms into a unified report to assess accuracy metrics of each of them, ensuring that our evaluation framework aligns with state-of-the-art requirements.

3.2 Heavy-Hitter detection algorithms

Before delving deep into our decision for the heavy-hitter algorithms, we believe it is necessary to provide a very high-level explanation of how a heavy-hitter algorithm works. These algorithms essentially extract some elements from the packets called flow identifiers. These set of elements serve as a unique tuple that can identify a continuous flow of data from a source to a destination among many packets. They can vary from algorithm to algorithm, but generally they are 5: the source IP address, destination IP address, the transport protocol number, the source port number and destination port number. A heavy-hitter algorithm essentially extracts these values from any packet and uses a technique to keep track of the amount of traffic that this flow generates.

As previously stated, our initial intentions going into this project was to essentially use state-of-the-art approaches to compare two heavy-hitter detection algorithms. These selected algorithms however, had to meet the following self-imposed requirements:

- First off, we believe that these algorithms should utilize the same underlying technique, but with a key differential in their internals, such as a use of a different data structure or function. We believe that by comparing two algorithms that use the exact same technique, their difference in results could be more probably recognized by this key differential in their internals rather than a bias in the testing environment.
- On the other hand, we believed that these results could be especially useful if applied to well-known or representative algorithms in the matter. This recognition could be caused by them being one of the very first algorithms to learn on the matter by prospective researchers, or could be due to them being used as the base for comparing the accuracy of new state-of-the-art approaches in the matter.

The algorithms we chose for this matter were *Count-Min*[10] and *Bloom-Filter*[11].

The first reason why we decided on the mentioned algorithms is due to the fact that both utilize the exact same technique: they use n different hash function on the flow identifiers to obtain n different indexes in a table, where they keep a record of the packet count for that specific flow. The main difference between Count-Min and Bloom-Filter is how they store the flow identifiers: while Count-Min traditionally uses n different tables for each hash function, Bloom-Filter uses a data structure of the same name to store all flow counts in the same table. As a result, they were a strong match to what we were looking for.

The second reason why we chose the two is due to the fact that Count-Min is one of the most representative algorithms on this topic as it not only is used as an entry point in the heavy-hitter

[2]Mininet, an emulator for rapid prototyping of Software Defined Networks. [41]

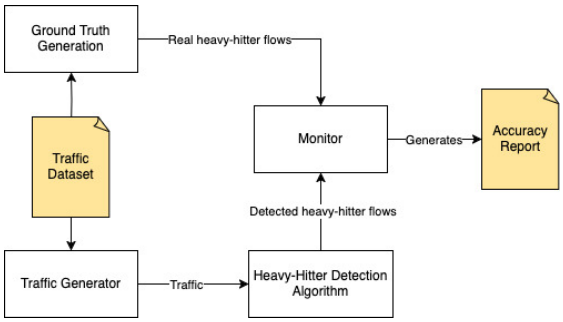

Figure 2: Architectural overview of the system.

detection problem by the official tutorials of the Networked Systems Group (NSG) [3], but also it is still being used as a point of comparison in state-of-the-art algorithms. However, after having decided on creating a benchmark, we believed it would be a unique opportunity to use a different algorithm with a different type of nature, so as to try to build a testbed that essentially bridges the gap between the algorithms that we had decided on and this new "different" algorithm. Hopefully, this could be used as a motivator for future researchers to try to improve on our design by adding the necessary features they need while still providing a necessary contribution to the community. As a result, we decided to go with the *PRECISION* algorithm [3] which, unlike Count-Min and Bloom-Filter that maintain a packet counter for all traversing flows, it works by keeping a record of the top-k flows along with their identifiers. This allows it to filter much more information to only keep record of that which considers relevant.

Prior to experimentation and benchmarking, we implemented the proposed heavy-hitter detection algorithms, performing any required modifications. The algorithms were configured on the switch as P4 programs, with slight adjustments applied on the general functions of the switch, such as forwarding behavior, to unify their structure while preserving the detection functionality. When designing the mechanism of reporting detections of heavy-hitters, we discovered a crucial internal characteristics of the algorithms that led us to further classify them into two distinct groups:

- Bloom Filter and Count-Min: These algorithms could be considered as *stateless*, as they do not store flow identifiers. Instead, when processing a packet, they hash the flow identifiers and discard them, using the resulting hash values to determine the positions of the corresponding counters. Consequently, if the memory contents of the switch were extracted, one would only obtain a list of the counters, with no direct way to obtain the flow identifiers from them. As a result of this, these algorithms can only detect and report a heavy-hitter when a packet from the corresponding flow is actively processed.

- PRECISION: This algorithm on the other hand could be considered as *stateful*, as this algorithm keeps track of the flow identifiers internally. When processing a packet, it hashes and stores the flow identifiers along with its packet counters. As a result of this, on these types of algorithms one could directly extract the memory contents of the switch in order obtain the counters and the identifiers for each flow to test if one flow is a heavy-hitter.

3.3 Testbed

The complete testbed can be represented as a series of modular components, where each of them follows a single responsibility in accordance to the SOLID principles[39]. An overview of their interactions represented as a flow diagram can be found in Figure 2.

The first step to be performed is called the `Ground Truth analysis`. This process extracts the real heavy-hitters from the traffic dataset which then will serve as the basis for reporting the accuracy of the algorithm. The result of this process is then fed into the monitor component which will then use it to judge the accuracy of the algorithm.

[3] `p4-learning`, compilation of P4 exercises, examples, documentation, slides for learning or teaching. `https://github.com/nsg-ethz/p4-learning`

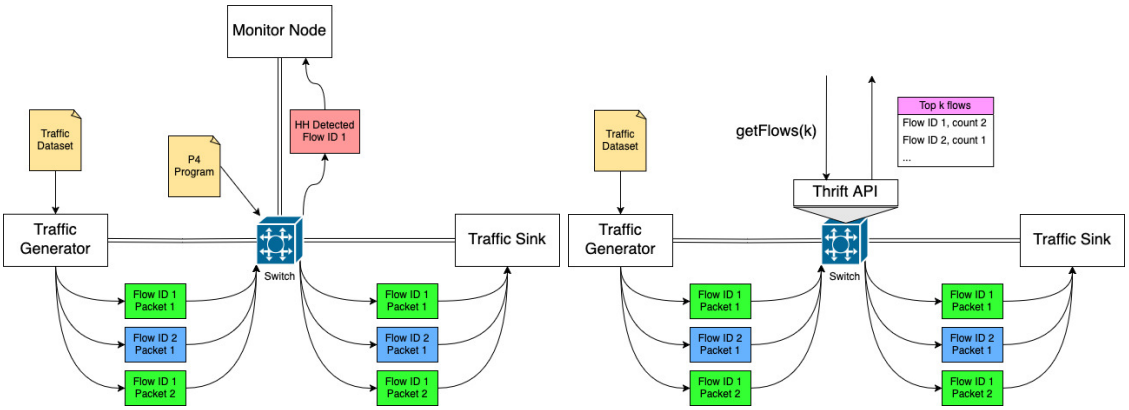

Figure 3: Overview of the mechanism used to obtain flow identifiers from stateless (left) and stateful (right) algorithms.

Continuing with the `Traffic Generator`, this component receives a traffic dataset in the form of a packet capture which then replays in the form of real traffic. This will then be captured and analyzed by the heavy-hitter detection algorithm.

The `Heavy-Hitter Detection` component processes incoming traffic by analyzing each packet to extract its flow identifier. What happens next depends on the type of algorithm. As explained above, if the algorithm maintains a persistent record of heavy-hitter flows by storing their identifiers, it is classified as stateful. Alternatively, if the algorithm does not retain flow identifiers and can only determine whether a packet belongs to a heavy-hitter flow at the moment of processing, it is considered stateless. When utilizing a stateful algorithm, in order to retrieve the heavy-hitter flows, one could just dump the entire switch's memory using a control-plane API and extract the flow identifiers from there; stateless algorithms, however, cannot do this. As they can only identify whether a flow is a heavy-hitter when a packet from this flow comes; as a result, the solution we have devised is to make the switch actively report when a new heavy-hitter is detected.

The final implementation of these mechanisms can be observed in figure 3. For stateful algorithms, we utilized one of the features of `p4-utils`, which is the addition of a control-plane API called Thrift API, in order to communicate with the P4-enabled switch. One of the features of this API is that it allows dumping parts of the memory of the switch, so that it can be used and analyzed by other programs. In the final implementation, we utilized this API to extract the flow identifiers along with their packet counts, which would then be sent to the `Monitor` component. On the other hand, for stateless algorithms we would create special heavy-hitter reporting packets that removed all data but the flow identifying fields. This way they would allow reporting every heavy-hitter while optimizing space to a maximum.

Finally, the `Monitor` component receives every detected heavy-hitter flow and compares it against the real heavy-hitter list (the ground truth). By comparing both lists, it creates the final accuracy report containing specific metrics of the experiment. The software code implementation of this testbed is open-source and can be found on GitHub [42].

3.4 Evaluation

In order to evaluate the performance of the algorithms under different examination constraints, we propose to verify their accuracy under the influence of the three following variables:

- **Packet Threshold**: Determines the minimum amount of packets that a flow must receive in order to be treated as a heavy-hitter. The heavy hitter threshold significantly affects the number of flows categorized as heavy hitters. A lower threshold classifies more flows as heavy-hitters, usually resulting in a higher sensitivity but also risking more false positives. On the other hand, a higher threshold is more selective, minimizing false positives but sometimes missing real heavy hitters. In practice, selecting a threshold requires balancing the chance of missing real heavy-hitters (false negatives) against the impact of incorrectly classifying too

many flows as heavy hitters (false positives). Applications that prioritize security may prefer lower thresholds for stronger detection, whereas network management could select higher thresholds to minimize overhead. Two detection thresholds for heavy-hitters were examined, set at 10 and 100, in order to test the effects of a lower and a higher threshold on detection accuracy.

- **Packet Volume**: It refers to the volume of traffic, specified in number of packets that the algorithm will be subjected to during the experiment. The volume of traffic passing through the switch can impact the performance of the program. A lower amount of traffic can equal to a lower number of distinct flows, which results in infrequent hash collisions and more accurate flow counts. A higher volume of traffic might result in the detection of big numbers of small flows which, even under large amounts of memory, hashing functions might start to generate collisions and noise. In order to test the effects of this, experiments were conducted with pre-generated traffic datasets of 1000 and 10000 packets to assess the impact on detection performance.
- **Register Sizes**: A register is a special type of data store that persists across packets. This variable essentially represents the amount of memory available to the program during the experiment. The capacity of the flow table or register sizes has a direct impact on the number of unique flows an algorithm can track at the same time. Modification of this parameter has to be done carefully as a low number of registers might lead to inaccurate detection, while a large number of registers might lead to unfeasible hardware implementations. In our tests, we examined two registers' capacity values: 192 and 1536 entries.

The experiments used the 2016 CAIDA Anonymized traffic dataset which comes from the internal Equinix-Chicago backbone. [43]

3.4.1 Accuracy measurements and metrics

In order to measure the performance of a specific algorithm, we decided to follow the steps of Singh et al. [31] and Ben Basat et al. [3] by using the following metrics:

- **True Positives (TP)**: Heavy-hitter flows correctly identified as heavy-hitters.
- **False Positives (FP)**: Non-heavy-hitter flows incorrectly identified as heavy-hitters.
- **False Negatives (FN)**: Heavy-hitter flows incorrectly identified as non-heavy-hitters.
- **Precision**: Fraction of correctly detected heavy-hitters among all detected flows.

$$\frac{TP}{TP + FP}$$

- **Recall**: Fraction of correctly detected heavy-hitters among all real heavy-hitters.

$$\frac{TP}{TP + FN}$$

- **F1 Score**: Harmonic mean of precision and recall.

$$\frac{2}{\text{precision}^{-1} + \text{recall}^{-1}}$$

These metrics were obtained from comparing existing entries in a list of the real heavy-hitters (obtained during ground truth generation) against entries from a list of detected heavy-hitters (obtained from the switch).

4 Results

This section shows the experimental findings and analysis of the three major detection algorithms across different scenarios. The experiments were designed to stress the compared algorithms on the three main variable parameters mentioned in the previous section, i.e., heavy-hitter detection threshold, switch registers' sizes, and number of packets, with accuracy reports for all possible combinations. In the following subsections we present the results of the accuracy of the algorithms when changing the variable under study while keeping the other variables with static fixed values. The value of these variables' values are shown in the form of `[register size, traffic volume, packet threshold]`. Detailed results can be found on Tables 2, 3, and 4, which show the performance metrics of each algorithm across various experimental situations, including packet count, register capacity, packet threshold, and calculated accuracy metrics.

4.1 Performance analysis across different thresholds

Figure 4 illustrates the accuracy of various algorithms across different threshold values. The F1 score is selected as the primary evaluation metric due to its ability to balance precision and recall effectively. Two configurations are examined here, namely [1536, 10000, 10] and [1536, 10000, 100]. From the results of this experiment, we can extract the fact that the packet threshold has a significant impact on the algorithms' ability to accurately distinguish heavy-hitters from regular flows. While a lower threshold classifies more flows as heavy-hitters, it also means that the algorithms are much more susceptible to "noise" in the network. This "noise" usually adopts the form of ephemeral flows with very few packet counts.

Here we can distinguish that both Count-Min and Bloom Filter are much more susceptible to this noise as their performance has had a big decrease. We believe this is because they aim to keep track of all flows, and in order to do so they use hash functions which result in hash collisions and in turn makes them overestimate the number of packets of a single flow, creating false positives in the process. PRECISION does not have this problem as it only has to keep track of the top-k flows. However, if the threshold is set too low for precision, then it might be the case that the number of real heavy-hitters is greater than k, which generates false negatives.

On the other hand, when setting a higher packet threshold, the noise in the network does not affect that much the algorithms. In the case of Count-Min and Bloom Filter, although setting a higher packet threshold improves accuracy, hash collisions still exist internally. This time however they are not enough to be able to create false positives. On the other hand, by increasing the threshold with PRECISION, then the number of real heavy-hitters is lower than k and false negatives cease to exist.

4.2 Performance analysis across different packet volumes

Figure 5 shows the accuracy (F1 score) of the different algorithms across different number of packet values. The parameters in the shown experiments were set to [192, 1000, 10] and [192, 10000, 10], respectively. We observe that the Bloom Filter method generally achieves a high recall, which means it rarely misses actual heavy hitters, with both packet count numbers, as long as there is enough registers capacity to track flows. However, when the registers' capacity is restricted (e.g., 192) and the packet count grows to 10,000, its false positives can significantly increase (e.g., exceeding 1,600 in certain instances), reducing its precision. Count-Min shows a similar trend, achieving perfect detection (TP = 18, FP = 0) with lower packet counts and high registers capacity, but facing a considerable increase in false positives at 10,000 packets when registers' capacity is small (e.g., FP = 1,775 with threshold = 10). On the other hand, when the flow table capacity is large (1,536), it maintains relatively high precision. PRECISION generally benefits from the recirculation mechanism, especially at high packet volumes (10,000). It achieves respectable results even with moderate capacity (192) (e.g., TP = 118, FP = 51 at threshold = 10). Interestingly, in the majority of cases, PRECISION achieves near-perfect accuracy at big capacity (1,536) (TP = 184, FP = 3, FN = 0, leading to an F1-score above 0.98).

These findings demonstrate that as the packet count rises, stateful algorithms that depend on more detailed flow monitoring and continuous updating for the stored flows using recirculation

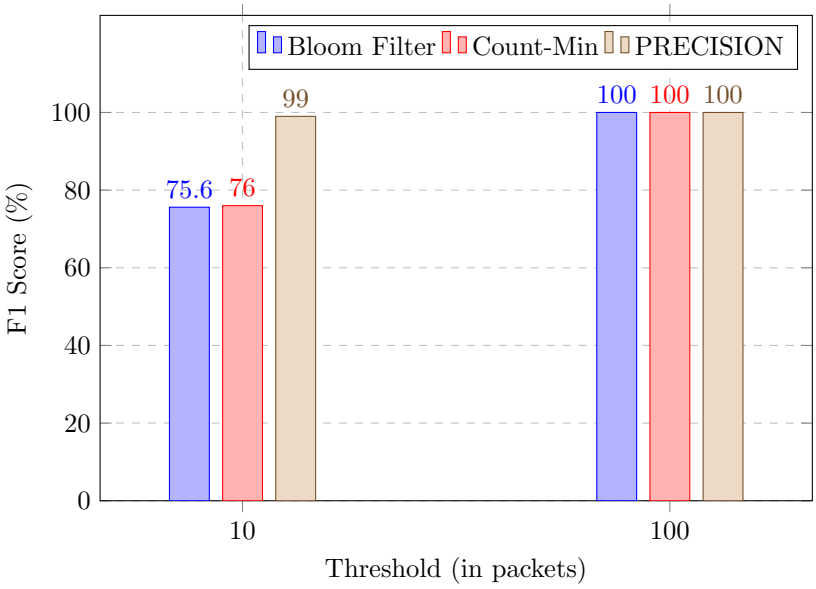

Figure 4: Comparison of Heavy Hitter Detection Accuracy at Different Thresholds

(such as PRECISION) can utilize the extra information to enhance their accuracy. At the same time, stateless and count-based algorithms (such as Bloom Filter and Count-Min) may be susceptible to increased false positives if their flow tables are limited. This corresponds with expected results that higher traffic volume provides more data for detection but also places greater demands on memory and data structures. In real-world scenarios, network administrators need to take into account both packet volume and memory limitations when choosing or adjusting a heavy hitter detection algorithm.

4.3 Performance analysis across different register sizes

Figure 6 shows the accuracy (F1 score) of different algorithms across different sizes for registers. The parameters [192, 10000, 10] and [1536, 10000, 10] are chosen to represent Registers capacity, number of packets, and threshold, respectively. The capacity of the flow table or register sizes, which can represent the number of unique flows an algorithm can track at the same time, strongly affects detection accuracy and false positives. In our tests, we examined two registers' capacity values: 192 and 1536. Bloom Filter: At lower registers capacity (192), it shows a significant increase in false positives (FP) when handling high traffic volumes (10,000 packets). Raising the capacity to 1536 greatly decreases FPs while preserving a high recall value. Count-Min: displays a similar trend, flow capacity limitations at 192 lead to high FP rates (e.g., 1,775 FPs for threshold=10, 10k packets). With the availability for 1,536 entries, FP significantly decreases, particularly for large traffic volumes. Overall, Count-Min benefits noticeably with extra memory. PRECISION: With a capacity of 192, it continues to achieve moderate FP levels (for instance, 51 at threshold=10, 10k packets), while a capacity of 1,536 often produces near-perfect results (for example, only 3 FPs under similar conditions). This enhancement shows that PRECISION is more efficient when there is sufficient memory.

In general, the results highlight that sufficient capacity is essential for avoiding collisions or forced evictions in heavy-hitters tracking. PRECISION, that keep a more detailed flow state and uses recirculation, face some performance drop when capacity is limited, but the effect is typically less severe than in Bloom Filter and Count-Min. These results align with the expectations that higher memory allocations result in improved accuracy for detecting heavy-hitters.

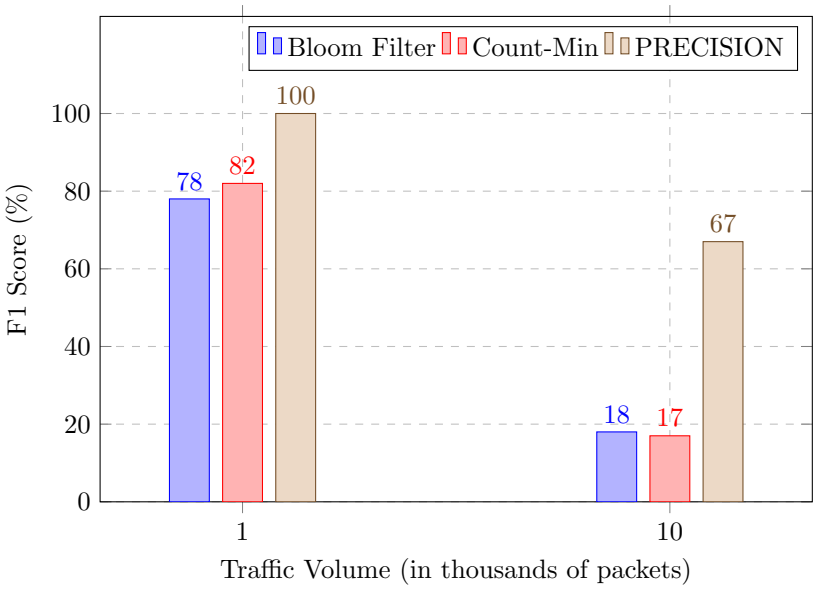

Figure 5: Comparison of Heavy Hitter Detection Accuracy at Different Packet Counts

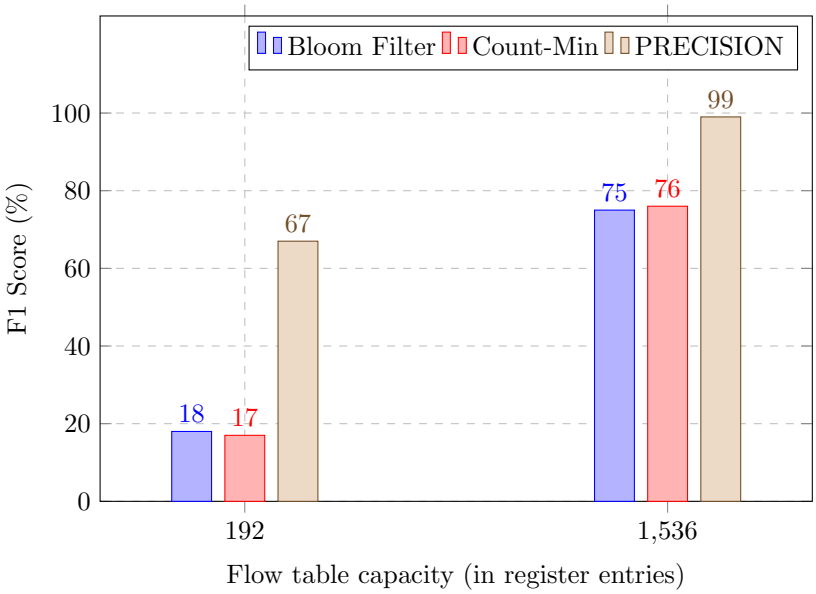

Figure 6: Comparison of Heavy Hitter Detection Accuracy at Different sizes of registers

Table 2: Bloom filter Results

Capacity	Packets	Thresh.	TP	FP	FN	Precision	Recall	F1
192	1k	10	18.0	10.0	0.0	0.6429	1.0	0.7826
192	1k	100	1.0	0.0	0.0	1.0	1.0	1.0
192	10k	10	189.0	1687.0	0.0	0.1007	1.0	0.1831
192	10k	100	11.0	97.0	0.0	0.1019	1.0	0.1849
1536	1k	10	18.0	0.0	0.0	1.0	1.0	1.0
1536	1k	100	1.0	0.0	0.0	1.0	1.0	1.0
1536	10k	10	189.0	122.0	0.0	0.6077	1.0	0.7560
1536	10k	100	11.0	0.0	0.0	1.0	1.0	1.0

Table 3: Count-Min Results

Capacity	Packets	Thresh.	TP	FP	FN	Precision	Recall	F1
192	1k	10	18.0	8.0	0.0	0.6923	1.0	0.8182
192	1k	100	1.0	0.0	0.0	1.0	1.0	1.0
192	10k	10	189.0	1775.0	0.0	0.0962	1.0	0.1756
192	10k	100	11.0	244.0	0.0	0.0431	1.0	0.0827
1536	1k	10	18.0	0.0	0.0	1.0	1.0	1.0
1536	1k	100	1.0	0.0	0.0	1.0	1.0	1.0
1536	10k	10	189.0	119.0	0.0	0.6136	1.0	0.7606
1536	10k	100	11.0	0.0	0.0	1.0	1.0	1.0

Table 4: PRECISION Results

Capacity	Packets	Thresh.	TP	FP	FN	Precision	Recall	F1
192	1k	10	17.0	0.0	0.0	1.0	1.0	1.0
192	1k	100	1.0	0.0	0.0	1.0	1.0	1.0
192	10k	10	118.0	51.0	66.0	0.698	0.641	0.669
192	10k	100	9.0	0.0	1.0	1.0	0.900	0.947
1536	1k	10	17.0	0.0	0.0	1.0	1.0	1.0
1536	1k	100	1.0	0.0	0.0	1.0	1.0	1.0
1536	10k	10	184.0	3.0	0.0	0.984	1.0	0.992
1536	10k	100	10.0	0.0	0.0	1.0	1.0	1.0

5 Discussion

Having analyzed each of the experimental results individually, we discuss the benefits and limitations that we found for each of the algorithms. Starting off with Count-Min and Bloom Filter, there is no very distinguishable performance difference between the two. In some scenarios, Count-Min Sketch performs slightly better than Bloom Filter, while in others, the opposite is true. We believe that this performance difference ($\leq 5\%$) is inconclusive of the existence of a differential between the two algorithms as, while it could be the result of using a different data structure, it could also be the result of the dataset used. These algorithms, on the other hand, achieve perfect recall score, only being susceptible to false positives in the cases where noisy networks are involved. Even though this downfall can be counteracted by raising the packet threshold or, raising the amount of memory available, these algorithms will eventually start producing false positives in a very noisy network segment if their internal tables are not refreshed periodically.

Carrying on with PRECISION, this algorithm achieved the highest F1 score in all test scenarios. It is very resistant to noise and can even achieve somewhat good results under very low amounts of memory with high amounts of noise and low thresholds. Overall, **PRECISION** emerges as **the most effective choice among the three algorithms in most scenarios**; it achieves very good results even under the worst conditions. However, in situations where **false negatives must be completely avoided**, **Count-Min Sketch** and **Bloom Filter** are be preferable. Since both algorithms consistently achieve **perfect recall**, they ensure that all heavy-hitter flows are detected, even if this comes at the cost of higher false positives.

Next, we discuss the beneficial contributions of our benchmark as well as all of the limitations that we are aware of. An essential component of this research is the ability to easily utilize a network packet capture as dataset to assess the algorithms. This practical approach enables any researcher to copy or create very specific scenarios to try the performance of an algorithm under those conditions. On the other hand, by utilizing community-provided captures, it also enables any prospective researcher to see how every algorithm operates with real-world traffic patterns. Another key contribution of this work is the developed software-based test environment that enables repeatable experiments with different parameters. Due to the limited availability of open-source implementations for heavy-hitter algorithms, our testbed and source code[42] can be used to evaluate other heavy-hitter algorithms, providing a stable benchmark for upcoming research.

As a conclusion, we would like to remark that to our knowledge, this is one of the first studies to directly examine these particular algorithms, offering new insights into their strengths and weaknesses. Despite these advances, there are still important limitations. Our experiments were implemented on a software-based setup, which does not consider hardware restrictions such as memory limitations or processing constraints. Although the real traffic dataset improves the practical importance of our findings, additional validation in a hardware setting would be important to verify if these results are consistent in the real networks environment.

6 Conclusion

In this research project we compared three heavy-hitter detection algorithms, Count-Min, Bloom Filter and PRECISION by analyzing their performance under different simulated scenarios. These scenarios involved the analysis of the effects on accuracy when modifying key variables such as packet threshold, traffic volume and available memory. In all test cases, PRECISION achieved the highest accuracy results specially when there is a high degree of noise in the network, making it the ideal choice out of the three algorithms for most scenarios. On the other hand, Count-Min and Bloom Filter achieved perfect results for recall under all test cases, making them useful for scenarios where false positive results are not tolerated.

On the other hand, this project also produced a replicable, hardware-agnostic testbed that future researchers can use to benchmark new detection methods. This implementation is currently software-only, that is why we believe that a hardware implementation would be necessary in order to get other types of results that complement and verify this work.

All in all, we believe this work provides the necessary contributions to advance the development of future heavy-hitter detection algorithms and in consequence, the field of programmable networks.

7 Future work

There are multiple directions to expand and improve this research:

1. **Hardware Implementations**: Whereas our current research is focused on software, implementing the code on real hardware is a logical next step. Taking hardware limitations into account could verify whether the results persist in real-world conditions.

2. **Support for different architectures**: Currently, our benchmark only supports P4 programs targeted at BMv2 architectures. Adding support for Tofino-based algorithms could be a definite upgrade on the capabilities of the testbed.

3. **Network-Wide Design**: Modifying the algorithms for a more realistic, multi-switch setting would allow monitoring across an entire network, improve scalability, and enhance results accuracy.

4. **Enhanced Traffic Generation**: Upcoming experiments might investigate various traffic patterns, such as bursty traffic or traffic with fluctuations, which would provide a greater understanding of the robustness and limitations of the algorithms. Adding the ability to create such patterns declaratively could also be a huge advantage for most tests.

References

[1] J. A. Hernández, D. Scano, F. Cugini, *et al.*, "Count-min sketches for telemetry: Analysis of performance in p4 implementations," arXiv:2406.12586, 2024. [Online]. Available: `https://arxiv.org/pdf/2406.12586`.

[2] Z. Zhang, B. Wang, and J. Lan, "Identifying elephant flows in internet backbone traffic with bloom filters and lru," *Computer Communications*, vol. 61, pp. 70–78, 2015. DOI: `10.1016/j.comcom.2014.12.003`.

[3] R. B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing heavy-hitter detection algorithms for programmable switches," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1172–1184, 2020.

[4] S. Kianpisheh and T. Taleb, "A survey on in-network computing: Programmable data plane and technology specific applications," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 701–761, 2023. DOI: 10.1109/COMST.2022.3213237.

[5] H. Heseding, F. Bachmann, P. S. Bien, and M. Zitterbart, "Saffirre: Selective aggregate filtering through filter rule refinement," in *2023 14th International Conference on Network of the Future (NoF)*, 2023, pp. 42–46.

[6] I. Corporation, *Intel tofino series*, Accessed: February 5, 2025, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

[7] B. Inc., *Merchant silicon networking chips*, Accessed: February 5, 2025, 2024. [Online]. Available: https://www.broadcom.com/info/switching/merchant-silicon.

[8] P. L. Consortium, *Behavioral model (bmv2)*, Accessed: February 5, 2025, 2024. [Online]. Available: https://github.com/p4lang/behavioral-model.

[9] C. Systems, *Cisco silicon one product family*, Accessed: February 5, 2025, 2024. [Online]. Available: https://www.cisco.com/c/en/us/solutions/silicon-one.html.

[10] G. Cormode, "Count-min sketch," in *Encyclopedia of Database Systems*, 2009.

[11] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, 7 1970.

[12] 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 (SOSR)*, 2017, pp. 164–176. DOI: 10.1145/3050220.3063772. [Online]. Available: https://doi.org/10.1145/3050220.3063772.

[13] T. H. S. Rodrigues and F. L. Verdi, "Detecting heavy hitters in network-wide programmable multi-pipe devices," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2024)*, 2024, pp. 1–8.

[14] M. Cai, Y. Liu, L. Kong, *et al.*, "Resource critical flow monitoring in software-defined networks," *IEEE/ACM Transactions on Networking*, vol. 32, no. 1, pp. 396–408, 2024.

[15] S. Sourabh, *HeavyHitterDetection*, https://github.com/sourabh4800/HeavyHitterDetection, GitHub repository, 2024.

[16] Y. Yao, S. Xiong, J. Liao, M. Berry, H. Qi, and Q. Cao, "Identifying frequent flows in large datasets through probabilistic bloom filters," in *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*, 2015, pp. 279–288. DOI: 10.1109/IWQoS.2015.7404747.

[17] F. Hauser, M. Häberle, D. Merling, *et al.*, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103 561, 2023. DOI: 10.1016/j.jnca.2022.103561. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804522002028.

[18] S. Zhang, H. Luo, Z. Wu, Y. Sun, Y. Wang, and T. Yuan, "Efficient heavy hitters identification over speed traffic streams," *Computers, Materials & Continua*, vol. 63, no. 1, pp. 213–222, 2020. DOI: 10.32604/cmc.2020.07496. [Online]. Available: https://cdn.techscience.cn/uploads/attached/file/20200320/20200320023115_74566.pdf.

[19] Pocper1, *p4_heavy_hitter*, https://github.com/pocper1/p4_heavy_hitter, GitHub repository, 2024.

[20] A. Ghanbari, *ECMP and HHD P4*, https://github.com/alighanbari2002/ECMP-and-HHD-P4-Exercises, GitHub repository, 2024.

[21] S. Yeole, *SDN Heavy Hitter Detection 2*, https://github.com/swarupyeole11/sdn_heavy_hitter_2, GitHub repository, 2023.

[22] 2. Gao, *P4 Heavy Hitter*, https://github.com/Gao2000/P4_HeavyHitter, GitHub repository, 2020.

[23] R. Abboud, *CMSIS-tofino2*, https://github.com/RaniAbboud/CMSIS-tofino2, GitHub repository, 2023.

[24] R. Abboud and R. Friedman, "Practical heavy-hitter detection algorithms for programmable switches," in *2024 IFIP Networking Conference (IFIP Networking)*, 2024, pp. 377–385. DOI: 10.23919/IFIPNetworking62109.2024.10619799.

[25] D. Damu, *Network-wide Heavy Hitter Detection*, https://github.com/DINGDAMU/Network-wide-heavy-hitter-detection, GitHub repository, 2020.

[26] Y. B. Mazziane, S. Alouf, and G. Neglia, "A formal analysis of the count-min sketch with conservative updates," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2022, pp. 1–6. DOI: 10.1109/INFOCOMWKSHPS54753.2022.9798146.

[27] V. Clemens, L.-C. Schulz, M. Gartner, and D. Hausheer, "Ddos detection in p4 using hyperloglog and countmin sketches," in *NOMS 2023 - 2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–6. DOI: 10.1109/NOMS56928.2023.10154315.

[28] D. Ding, M. Savi, G. Antichi, and D. Siracusa, "An incrementally-deployable p4-enabled architecture for network-wide heavy-hitter detection," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 75–88, 2020. DOI: 10.1109/TNSM.2020.2968979. [Online]. Available: https://ieeexplore.ieee.org/document/8967165.

[29] 1. Liuzhengzheng, *Heavy Hitter*, https://github.com/liuzhengzheng12/heavy_hitter, GitHub repository, 2024.

[30] INTRIG-UNICAMP, *P4-HH*, https://github.com/intrig-unicamp/P4-HH, GitHub repository, 2023.

[31] S. K. Singh, C. E. Rothenberg, M. C. Luizelli, G. Antichi, P. H. Gomes, and G. Pongrácz, "Hh-ipg: Leveraging inter-packet gap metrics in p4 hardware for heavy hitter detection," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 3536–3548, 2023. DOI: 10.1109/TNSM.2022.3227065.

[32] 8. Khooi, *P4-HashPipe*, https://github.com/khooi8913/p4-hashpipe, GitHub repository, 2020.

[33] R. Huang, *Side-Project-Heavy-Hitter-with-P4*, https://github.com/RaymondHuang210129/Side-Project-Heavy-Hitter-with-P4, GitHub repository, 2017.

[34] PlinkPlunkT, *votePipe*, https://github.com/PlinkPlunkT/votePipe, GitHub repository, 2022.

[35] D. Li, N. Tian, K. Qiu, H. Chang, X. Yu, and J. Zhao, "Votepipe: Efficient heavy hitter detection in programmable data plane," in *Frontiers of Networking Technologies*, X. Wang, M. Xu, X. Shi, and F. Wu, Eds., Singapore: Springer Nature Singapore, 2024, pp. 146–166, ISBN: 978-981-97-3890-8. [Online]. Available: https://doi.org/10.1007/978-981-97-3890-8_11.

[36] 8. Khooi, *dSketch*, https://github.com/khooi8913/dsketch, GitHub repository, 2021.

[37] X. Z. Khooi, L. Csikor, J. Li, M. S. Kang, and D. M. Divakaran, "Revisiting heavy-hitter detection on commodity programmable switches," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, 2021, pp. 79–87. DOI: 10.1109/NetSoft51509.2021.9492531.

[38] chenxiaoqino, *PRECISION*, https://github.com/p4lang/p4-applications/blob/master/research_projects/PRECISION, GitHub repository, 2019.

[39] R. C. Martin, *Agile software development, principles, patterns, and practices* (Alan Apt series), en. Upper Saddle River, NJ: Pearson, Oct. 2002.

[40] S. Bartel and A. Bartel, *The Official Guide to the Kanban Method*. Kanban University, Feb. 2021, Accessed: 2025-02-03. [Online]. Available: https://kanban.university/kanban-guide/.

[41] Mininet Website. [Online]. Available: https://mininet.org/.

[42] T. Agata and M. Mansour, *Hh-detection*, https://github.com/tomasagata/hh-detection, GitHub repository of the benchmark, 2025.

[43] CAIDA, *Anonymized Internet Traces 2016*, https://catalog.caida.org/dataset/passive_2016_pcap, Accessed: 2025-02-08, 2016.