# Optimized Cuckoo Filters for Efficient Distributed SDN and NFV Applications

Aman Khalid Flavio Esposito
*Computer Science Department*
*Saint Louis University*
St. Louis, USA
{first.last}@slu.edu

***Abstract*—Membership testing has many networking applications like distributed caching, peer to peer networks, or resource routing, to name a few. Several studies have reported the advantages of using membership testing in Software Defined Networking, and Bloom Filters have been widely adopted for that purpose. Cuckoo Filters is a recently proposed alternative to Bloom that outperforms them in terms of speed and memory efficiency, with some drawbacks. In this paper, we propose an Optimized Cuckoo Filter (OCF) design that limits some of the Cuckoo Filter drawbacks and gives a better-amortized search time, with less false positives. We then present an implementation of Optimized Cuckoo Filter in distributed SDN and NFV applications, with customizable parameters that enable the data structure to adapt to different workloads. We discuss the use cases of this data structure in SDN and show the performance gain when using our solution with proper configuration. We also show the benefits of this data structure in different SDN and NFV applications by simulating real-world scenarios: content-centric caching and Virtual Firewall as a Network Function and invoke dialog for the widespread adoption of this data structure outside academia through open-source collaboration.**

***Index Terms*—Membership Testing, Cuckoo Filter, Software Defined Networking, Generalized forwarding**

## I. Introduction

A Bloom filter is a simple space-efficient randomized data structure for representing a set to support membership queries. Bloom filters allow false positives, but the space savings often surpass this drawback when the probability of an error is controlled [2]. Bloom filters have been used in database applications since the 1970s, but in recent years, they have become popular in the networking literature, especially for high-performance networking applications like NFV. This adoption is due to their speed, higher than linear lookup or binary search, and their practicalities at the scale in which distributed applications operate. An excellent survey of the many networking systems that use Bloom filters for high-speed set membership tests can be found at [1].

Bloom filters permit a small fraction of false-positive answers with excellent space efficiency. However, they do not permit deletion of items from the set, and previous attempts to extend "standard" Bloom filters to support deletion degrade either space or lookup performance. Recently, Cuckoo filters have been proposed as a new data structure to cope with the limitations of standard Bloom filters [2].

A Cuckoo Filter uses lightweight Cuckoo hashing [3], which is constructed with two arrays of buckets, and each element can be stored in either of the two buckets. The number of arrays in the filter is constant, and of because this, the lookup time of a cuckoo filter is $O(1)$.

There are three major advantages of using Cuckoo filters over Bloom filters [4]. 1) They support adding and removing items, 2) They have higher lookup performance. 3) They are easier to implement and give better space optimizations.

On the other hand, Cuckoo Filters have some limitations [5] like — the need to know the total number of items to be inserted beforehand, and the inability to delete previously uninserted keys. In this paper, we address those limitations by proposing an optimization of the cuckoo filter data structure to make it more flexible in the workload of networked systems such as Software-Defined Networking (SDN), Network Function Virtualization (NFV), or other latency-sensitive distributed systems.

**Our Contributions.** In particular, we present The Optimized Cuckoo Filter (OCF), a data structure for membership testing that has been designed to satisfy various requirements of network applications. Our OCF has customizable policies that can be tuned (programmed) to adapt to different NFV and other distributed system requirements. We show that our optimized Cuckoo filter performs better than the regular Cuckoo filter in SDN use-cases like network caching, multicast, and generalized forwarding. We discuss practical uses of this data structure and implement several use cases on a virtual network testbed to emulate real-world scenarios using OCF as a proof of concept. We further discuss the tradeoffs of using a particular configuration and what effect does a particular parameter has on the end configuration.

The rest of the paper is organized as follows. In Section II, we discuss the use cases of Optimized Cuckoo Filters in Network Function Virtualization and Software Defined Networking. Section III describes the limitations of traditional Cuckoo Filters and a scenario where the original implementation fails. Section IV contains our contribution - The Optimized Cuckoo filter with its specifications and an overview of its parameters. Section V has the evaluation results and the experimental setup's description that the results. In Section VI, we discuss

978-1-7281-8159-2/20/$31.00 ©2020 IEEE


Authorized licensed use limited to: SAINT LOUIS UNIVERSITY. Downloaded on January 02,2021 at 00:47:54 UTC from IEEE Xplore. Restrictions apply.

related membership testing applications in Software Defined Networking, and finally, we conclude our work in Section VII.

## II. Motivating Applications and Use Cases

In this section, we highlight some of the benefits that our OCF could have when applied to some use cases of interest within network (function) virtualization scenarios.

**Efficient rule lookup in generalized forwarding.** Advances in Software-Defined Networking (SDN) have alleviated some problems that network operators used to face, since having separate middleboxes at different layers of the network is suboptimal. In particular, generalized forwarding follows the "match-plus-action" paradigm. Matching operations are done on several fields of packet headers, based on values read across different packet headers. A packet can be forwarded to the desired action which can be blocking, load balancing, rewriting header values, etc. Generalized forwarding could be made more efficient by implementing an OCF in an OpenFlow or other SDN controller. The OpenFlow protocol has evolved to support generalized forwarding of over 40 header fields across the stack, and large data-center may see a matching operation as often as a few milliseconds.

**Loop detection in SDN-driven multicast.** Using SDN to manage multicast connections offers particular merits. Network operators have a complete view of the topology [6]. Earlier on, multicast algorithms had to make routing decisions without any central view of the network. We no longer need to rely on local information [7]. Application-Aware SDN multicast Routing Algorithms can make more efficient multicast decisions.

However, routing for multicast can be mapped to an NP-hard graph matching problem that is usually solved in phases. The underlying graph hosting the multipath connections has to be analyzed for possible paths and forward loops. Our Optimized Cuckoo Filter has a unique property that detects if the filter is full and displaces keys. If the keys get displaced for the entire round, that means a forwarding loop has been incurred (efficiently).

**Membership Testing in Content-Centric Caching at SDN.** Content-Centric Networking (CCN) [18] is a fairly recent Internet architecture that has seen some early adopters thanks to network virtualization. The paradigm's main premise is to decouple location from identity, and the main characteristic of the architecture is to access and retrieve content by name, without the need to use the IP protocol (that is known to have multi-homing and mobility problems [8]). CCN Peers can ping each other to check for missing content. This makes it an interesting application for OCF as the peers who maintain a filter of the keys stored at them can share them with others to convey the files they have locally.

**Virtual firewalls in VNF.** Early adopters of Network Function Virtualization technologies were primarily telecommunication providers. Therefore, most of the original use cases relate to firewalling, routing, VPN, NAT, DHCP, IPS/IDS, PBX, transcoders, or WAN optimization. Earlier firewalls on networks were physical devices that were positioned between the uplink and the network. The shift to VNF enabled the development of partially cloud-native VNFs. With more disposable computing resources, it is possible to make cloud-native VNFs that allow adding, removing, or modifying rules in the packet filter ruleset on the go in virtual firewalls.

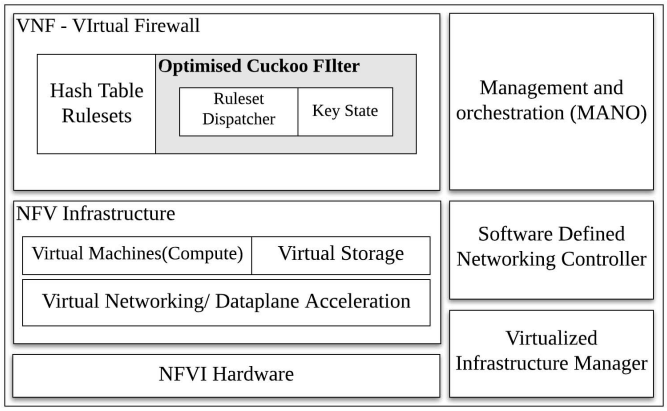



Fig. 1: A reference NFV architecture showing the placement of a firewall network function implemented using Optimized Cuckoo Filter.

(Optimized) Cuckoo Filters allow writing of rule sets in the form of hash tables that can store an extensive collection of rule sets, and effectively match packets. The hash tables can be trained for the required parameters, and the parameters in the incoming packet can be matched with high speed and accuracy. Moreover, dynamic mitigation strategies are necessary to deal with the diversity of cyber-threats. Usually, some of these threats are detected by deploying a large number of firewall rules. An Optimized Cuckoo Filter can help here as this advanced data structure can be configured for specific scenarios.

Modern NFV architectures are implemented on top of cloud operating systems, that have functionalities like compute, orchestration, and infrastructure managers. In figure 1 we show a reference data center with a single network function (for clarity) — a virtual firewall that matches packets using the hash table created using the in-memory stateful Optimized Cuckoo Filter. The Key State serves as a buffer for all the rulesets that will be used to create the filter, and ruleset dispatcher updates the hashtable.

Some of the applications described can be implemented using traditional Cuckoo filters, despite their limitations. Methods such as membership testing in content-centric or SDN caching require inserting an undefined number of keys in a node; therefore, any implementation that uses traditional cuckoo filters or bloom filters would require reconstructing the internal data-structure when the filter gets full, which may result in downtimes, depending on the implementation.



Authorized licensed use limited to: SAINT LOUIS UNIVERSITY. Downloaded on January 02,2021 at 00:47:54 UTC from IEEE Xplore. Restrictions apply.

## III. Problems with Existing (Cuckoo) Filters

**Cuckoo Filter Background.** Cuckoo filter is a data structure for membership testing proposed by Fan et al. [2]. It is an improvement over the widely used Bloom filter as it supports deletions while keeping higher performance. Cuckoo filters and its applications have been widely examined, but their adoption in commercial software has been slow. Cuckoo filters are (arguably) more straightforward to implement than many alternatives, and they use less space than Bloom Filters.

Cuckoo filters are based on cuckoo hashing [24], it is a hash-based data structure that handles hash collisions. When a key is hashed to a bucket that is not empty, a collision occurs. Such collision triggers the displacement of the already present key. A cuckoo hash has two logical buckets, each assigned to a hash function. The key is first sent to a first bucket; in case of a collision, the displaced key is moved to the second bucket. If the second key further displaces a key, such a key tries to find a spot in the first bucket. This process can go on for a long time if the hash is approaching its full capacity. Formally, the Cuckoo filter has two hash functions: $h_1(x)$ and $h_2(x)$; each hash maps a potential set member to the filter. A lightweight fingerprint $\phi(x)$ is generated from the key. Each element is stored in a cell $h_1$ or $h_1 \oplus h_2(\phi)$ in the hash table. The xor operation in the second possible location ensures $h_1(x)$ can also be calculated from $h_2(x)$ and $\phi(x)$, using the same formula. When the filter starts to get filled, the probability of false positives increases.

**What problem are we solving?** Despite having clear advantages over Bloom Filters, Cuckoo Filters do not perform efficiently in some distributed applications, such as SDN and NFV, because they lack policy programmability. Delete operations are not fail-safe, and operating at full capacity can trigger too many rehashes thus deteriorating the performance.

Even with a high fault tolerance rate, the faults per query increases exponentially. Factors like sudden changes in traffic, can lead to over or under-utilization of resources [10], [11]. Consider sets $T, U,$ & $V$ stored in different nodes in a datacenter. We need to find Cartesian product $T \times U = \{(t, u) \mid t \in T \wedge u \in U\}$ $s.t\ V_\alpha > u.T \forall\ T \times U$. This query will first create a set of size, $s = size(T) * size(U)$ . Then it will trigger $s$ queries in $V$ to filter results in $T \times U$. In this case, the number of look-ups on the node containing $T$ is much greater. This problem becomes even more severe when the number of queries grows.

## IV. Our Solution: Optimized Cuckoo Filters

A limitation of the conventional bloom filters is that it does not support deletes. Several proposals extend the traditional Bloom Filter, but Hash Table based approaches make filter less space-efficient. Also, the number of elements to be stored must be known before the filter creation. The traditional Cuckoo filter provides a higher lookup performance than Bloom Filters, and it also consumes less space as long as that the false positive rate remains below 3% [12], [13].

While the original Cuckoo filter outperforms the bloom filter in terms of memory and lookup speed, it fails when the maximum load goes beyond 0.9 [14], [15]. There have been adaptations of the Cuckoo filter in distributed databases, which suffer from this issue. The community also observed an occasional false negative when operating at this threshold [16], which breaks membership testing. Therefore to run reliably in the cloud, Cuckoo filters need to account for the unpredictable nature of traffic.

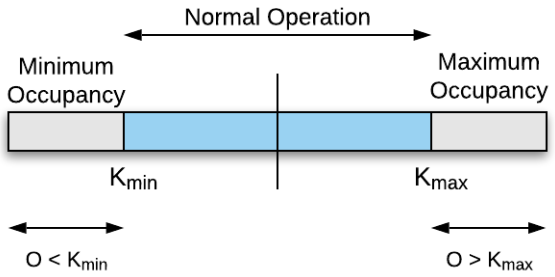



Fig. 2: Visual Representation of Optimized Cuckoo Filter's Bucket Occupancy $O$. In congestion aware (CON) mode the filter operates normally (blue region). OCF starts monitoring the rate of inserts or deletes when $O$ reaches a certain value, when the Maximum Occupancy is reached the filter is resized based on these observations.

The design of our OCF is inspired by congestion in network switches. The ability to adapt based on the extent of the load is the prime focus of our design and implementation [17]. OCF can be fine-tuned for different requirements. OCF can operate in two modes, selected during an initialization phase: Congestion Aware (CON) mode and Primitive mode (PRI). In the rest of this section, we detail the functionality of both modes.

**Primitive.** The primitive mode of operation (PRI) of OCF adjusts the size of the filter based on static parameters. The user can choose the minimum and maximum thresholds for the size of the filter. The filter is resized when the occupancy rate reaches the preset threshold. Using the PRI mode, performance levels are acceptable if the number of keys is smaller than a million records; however, it is not advisable to use PRI when the number of keys is larger than one million. At that scale, subsequent deletes cause the filter to shrink linearly. This is because, if the number of elements falls below a minimum threshold, the filter is resized. However, if the occupancy $O$ remains above the safe limit, this mode can result in undesirable false negatives.

**Congestion Aware**. The Congestion Aware mode (CON) changes the filter based on the rate of insertion or deletion in the filter. This dynamic nature is implemented by marking all the insertions and deletions beyond a value k. In Fig. 2, the area between Min Occupancy $O_{min}$ and Max occupancy $O_{max}$ represents the value of occupancy.

If the filter occupancy remains between the two thresholds, $k_{min}$ and $k_{max}$ no resize is triggered. In CON mode, when



Authorized licensed use limited to: SAINT LOUIS UNIVERSITY. Downloaded on January 02,2021 at 00:47:54 UTC from IEEE Xplore. Restrictions apply.

$O < k_{min}$ *or* $O > k_{max}$, OCF starts monitoring the changes in the filter occupancy. The new size of the filter is determined based on the rate and number of entries that are added or removed from the filter. Using this mode is safer when the number of records is larger than one million as each increase or decrease takes into account the factors that caused the previous resize. It is not recommended to use this mode for smaller workloads as PRI performs better while consuming comparatively low memory.

### A. OCF Policies and Parameters

In this section, we discuss the programmable policies of our Optimized Cuckoo Filter and their tradeoffs. These policies are critical to our Optimized Cuckoo Filter's design. Setting the right parameters for a particular use case is essential to benefit from this data structure. For example, for some NFV workloads, such as firewalls and deep packet inspection for identification of malicious IP addresses, it is desirable to optimize false-positive rates while compromising memory. In other NFV applications, for example, load balancing, operators might prefer tighter memory utilization.

We define the capacity $c$ of the OCF filter as the number of elements a filter is meant to store. We recommend setting such capacity to a value at least twice as large as the number of elements inserted; this is because as the filters get full, the number of false positives increases. The capacity of OCF can be set to be dynamic, which means the size of the filter can be reduced or increased based on the number of elements being inserted or deleted. Such intelligence can be gathered from and implemented within an SDN controller.

Another tunable policy of our OCF is the Bucket size: the size of individual buckets in the filter. In our implementation, we have set this policy to a value of 4, noting that this is enough to trigger evictions and consume a reasonable amount of space in our tests. For performance, it is not recommended to reset this policy dynamically after the bucket has been created. Another policy that should not be changed after the OCF initialization is the fingerprint size: the length of the fingerprints that will be stored in all buckets. The value of this policy should be chosen based on the total expected number of items to be stored in the node. Choosing a lower value can cause collisions. If the fingerprint size is set to 6, we have $(10)^6$ possible unique fingerprints, that is, the number of unique keys that can be stored in the OCF before *reinitializing* is $(10)^6$.

It is also possible to tune the number of evictions that occur in the Optimized Cuckoo Filter by setting the maximum displacements. This policy sets the number of times a filter will try to find a place to store the newly arrived item. After the number of retires is reached, the filter is full. Setting large values for this policy is desired when the filter's size is expected to be significant.

The maximum and minimum occupancy can be configured for both modes of operations of our Optimized Cuckoo Filter. These two parameters are crucial to tune the filter's memory usage. The maximum value for occupancy is 1, so to have fewer false positives, the occupancy should be kept low. In our implementation, the value of Maximum occupancy has been set to 0.85. Note that the filter's size will increase once this threshold has been reached. This is because a resize is triggered when the occupancy reaches the set threshold.

The following policies are only available in the Congestion aware mode (CON) of our Optimized Cuckoo Filter. The K-marker defines the minimum and maximum threshold at which the Optimized Cuckoo Filter starts monitoring the rate at which keys are being added or deleted from the filter. This information is useful to know how much the filter size should be increased or reduced instead of doubling it or reducing it to half as in the primitive mode (PRI). The Estimation Gain $g$ sets the rate at which growth factor $\alpha$ increases. In our implementation, we set a default value of $g = 1/16$. Choosing a value closer to one causes $\alpha$ to increase at a larger rate. The tradeoff is between memory usage and fewer resizes.

### B. Optimized Cuckoo Filter Resizing Algorithm

In PRI mode the OCF does not account for the rate at which the filter gets filled. The occupancy of the filter is the prime factor that decides when will the filter be resized. O is calculated by Number of Items in the bucket s and Capacity c, $O = s/c$ where $0 < O < 1$. When $O > O_{max}$ the bucket is doubled in size. In case when the items in bucket decrease below $O_{min}$ the bucket size cannot be simply reduced to half, instead the new size is calculated by $c = (c - (c/10))$.

**Algorithm 1** Algorithm to resize bucket in CON mode

```
When O > k_max | O < k_min mark the consecutive items
Once O reaches the threshold:
    Set:  M = (c' * t')/(c * t)
    Set:  α = α * (1 − g) + g * M
IF  O < O_max
    c = c − c * (1 − α)
ELSE
    c = c + c * (α)
```

The CON mode, OCF starts marking items when bucket occupancy goes beyond $k$. Once $O$ becomes greater than $O_{max}$ or less than $O_{min}$. Growth factor $\alpha$ is calculated. The value of $\alpha$ is directly proportional to $g$ and the ratio of the previous and current rates which is capacity and time before reset $c'$ & $t'$, and the capacity and time during reset $c$ and $t$ respectively.

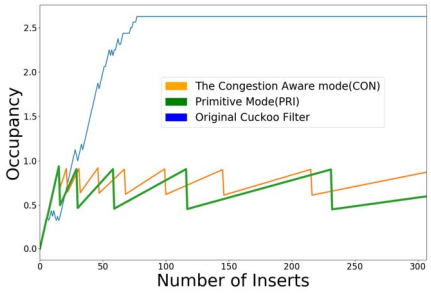
## V. Evaluation Results

### A. Membership Testing

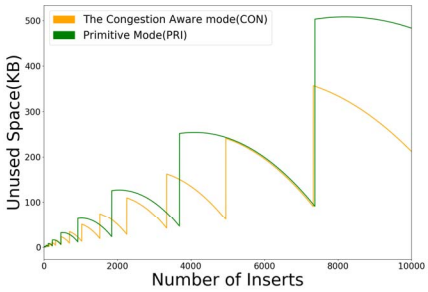
We ran our implementation on different key sizes ranging from 10,000 - 1,000,000. We test both the modes of Optimized Cuckoo Filter (OCF) for throughput and accuracy. We observed that the average number of false positives for Congestion Aware (CON) and Primitive (PRI) modes were 49 and 32 at 100,000 keys. Also, CON mode maintains a higher value of occupancy than PRI, thus having better space utilization. However, PRI has a better false positive count as



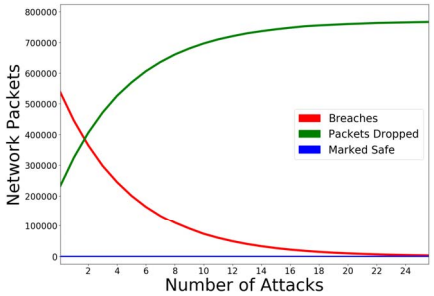
Authorized licensed use limited to: SAINT LOUIS UNIVERSITY. Downloaded on January 02,2021 at 00:47:54 UTC from IEEE Xplore. Restrictions apply.



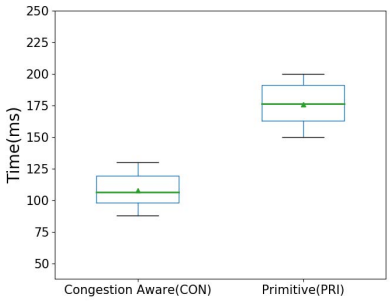

(a) Filter stress test. Both OCF modes manage the filter occupancy (keep Occupancy below 1) well as the number of keys grows, while the traditional Cuckoo filter does not.



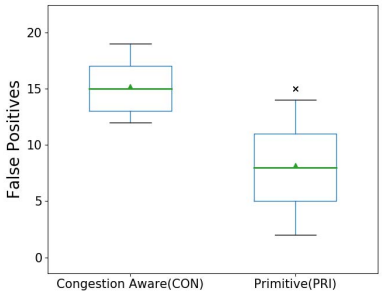

(b) Unused bucket space of CON and PRI modes. PRI gives better performance when number of keys is less, however it starts over-utilizing memory at large scale




(c) Packet state of the OPNFV network, using virtual firewalls implemented using OCF, during a sequence of DDoS attacks.




(d) Time taken to search 5000 names in the Content Centric network by Congestion Aware(CON) and Primitive(PRI) modes, for 100 test runs.




(e) Number of false positive observed over the course of 100 test runs.

Fig. 3: **Dataset and forecasts**: in *(a)*, OCF in PRI mode re-scales the filter fewer times as it doubles in size when the load increases. In *(b)*, CON mode utilizes the available memory more effectively as ratio of the number of elements to total capacity remains close to one. In *(c)*, the number of breaches reduce as packets from unknown sources get blacklisted and dropped over the course of the test. In *(d)*, CON mode outperforms PRI in look-up speed, while having a lower memory footprint. Lastly, in *(e)*, PRI mode gives better false positive rates, while having a higher memory footprint for storing names in the nodes.

its occupancy is below $50\%$. On the other hand, this policy configuration consumes a lot more space, and the filter is mostly underutilized.

In Figure 3a we can see how the original Cuckoo gets filled within the first few trials of the experiment. Both CON and PRI perform well for the first 50 rounds; however, as the number of elements increases, the size of the filter in PRI mode gets exponentially larger, therefore, consuming more space than necessary, whereas CON maintains an optimal size.

In Figure 3b, we show that the unused bucket space for CON and PRI are similar in the initial trials; as the size of the filters increases, CON tends to maintain size optimality while utilizing the maximum possible space, hence doing this is beneficial because memory constraints become more prominent at that scale.

### *B. Content Centric Caching*

To test the performance of OCF for content caching, we created a virtual network testbed using Mininet [23]. We simulated a scenario where a new virtual node queries for cached names with the peers in its network. To simulate this scenario, we created 50 nodes arranged in a fully connected network, communicating with each other using the gossip protocol.

The total number of names or keys in the network is over 100,000 and each peer node has less than 12,000 keys stored in its filter. The newcomer is searching for five thousand keys and starts by asking the 50 peers in the network.

CON mode of OCF has a smaller memory footprint and gives better runtime than PRI mode. As we can see in Figure 3d OCF in CON mode took less time to finish the search over 5000 keys. Both simulations were performed on similar network conditions. Factors such as the memory of the virtual node affect the overall time of a run. As expected, the CON mode is faster than PRI as it takes up less memory on the host node.

However, it is interesting to note that PRI performs better in terms of false positives (Figure 3e). This better performance is due to the better false-positive rates when the filter size is twice as large as the number of elements present in the filter. We ran these experiments on a machine with 8 GB RAM, running Intel's i7-8750H processor with 12 cores.



Authorized licensed use limited to: SAINT LOUIS UNIVERSITY. Downloaded on January 02,2021 at 00:47:54 UTC from IEEE Xplore. Restrictions apply.

### C. Virtual firewalls for DDoS prevention

Network Function chaining is the method of passing end-to-end data streams through a sequence of Network Functions. Although having a service chain of multiple VNFs is harder to manage, modular functions prevent a single point of failure in the tunneling protocol between VNFs and keep logical entities separate. This concept is adopted by modern Firewalls [19] for service providers. Distributing the firewall task to multiple network functions makes them better prepared to defend against security threats to ensure their network is always available, in case of DDoS attacks.

As proof of concept, we implemented a Virtual firewall on OPNFV [20] reference architecture. In our implementation, we analyze the source and destination IPv6 headers of each incoming packet and decide to forward or drop them. VNFs consists of two virtual firewalls that prevent Denial of Service attacks. The first firewall is responsible for detecting packets originating from blacklisted sources. However, this does not check for unknown or whitelisted sources, which are forwarded to the next firewall. The second firewall checks if the packet source and destination headers are present in the whitelisted sources if a packet from an unknown source has made it this far, it is added to the blacklisted sources to prevent further requests from that address. Both firewalls contain hash tables: the first firewall has the fingerprints of blacklisted sources, and the second one has the whitelisted sources. The source and destination, headers of an incoming packet are being searched in the hash table of the Optimized Cuckoo Filter, and the forwarding decision is based on that. The filter has been configured for high throughput and low false positives as it is desirable in this scenario while compromising memory efficiency for better security.

The second firewall is installed with 1,000,000 trusted sources. The first firewall initially contains 500,000 blacklisted sources, which will increase throughout the experiment. To simulate a DDoS attack on our network, we create modified packets using Scapy [22]. The packets originate mostly from blacklisted and unknown sources to simulate a DDoS attack and a few regular network packets. If a packet from an unknown or blacklisted source makes it to the second firewall, we consider it a "breach" as this effects the regular packets passing through the function chain. As seen in Figure 3c, the number of breaches initially is high, as the network is attacked by 3,000,000 unknown sources and 500,000 blacklisted sources. Once a packet reaches the second firewall, it is dropped, and the filter in the first firewall is updated. Subsequent packets from that source are then dropped. Hence the number of breaches reaches almost zero.

## VI. Related Work

Bloom Filters and its variants have been examined thoroughly over the years; they are still the industry standard for commercial and open source applications. The literature reveals that membership testing has novel applications in software-defined networking [21].

Du and Wang [9] propose two-stage adaptive Bloom filters that work collaboratively across the span of the network in SDN. They address the issue of per-flow monitoring in applications with varied monitoring requirements that cannot be offered with generic traffic monitoring methods for coarse-grained visibility. They provide this flexibility by adjusting the number of filters in their action Bloom Filter, used for flow classification. However, the individual filters are not dynamic, and the number of filters is adjusted by the controller, which has a global view of the network.

Bhattacharya et al. [25] improve upon the model for Learned Bloom Filters (LBF) [26], [27] by offering improvements to the existing learned model by supplementing it with an adaptive bloom filter. The authors present a solution that deals with the adaptability of LBF for incremental workloads. They address the challenge of accommodating adaptability in the static version of the learned set membership problem. The tradeoff is the size of the filter and false-positive rates.

Ozisik et al. [28] propose Graphene to provide a protocol for interactive set reconciliation among peers in distributed systems. Their novel protocol uses 12% of the bandwidth of existing deployed systems to provide synchronization among replicas in a distributed system. They do so by exchanging false-positive rates of the filters among the peers in a network so the internal memtable can be corrected. They present a novel combination of Bloom Filters and Invertible Bloom Lookup Table which provides a solution to the case, where one party is missing some or all of the subset.

Kalghoum et al. [29] propose Novel Forwarding Strategy for Content-Centric Networking [30] based on SDN and Bloom Filter. They use FIB bloom filter to check whether a key exists to make decisions on how to how a packet would be routed and in case of a miss, the request is redirected to a controller to initiate a network-wide search.

Jinyuan, et al. [31] Propose a packet classification algorithm in OpenFlow switching to address the problem of bottlenecks in that area. They use counting bloom filters to predict the failures of mask probing without searching the flow table. They show that their method maintains steady flow table lookups as compared to the classical algorithm in Open vSwitch which goes up rapidly with increasing flow table length.

Graphlene is optimal when the block size grows, but is not space optimized for a small number of elements. Du and Wang predetermined false-positive rates. Size of filters is defined with a fixed false-positive rate, however, the number of false positives can get high if the traffic exceeds beyond what the filter was optimized for. The iterative hash function approach used by Kalghoum et al. for Content-Centric Networking gives better runtime at the cost of speed. There is also a memory-efficient approach for content-centric networking for SDN proposed by Berto et al. [32] by using Spatial Bloom Filter.

The prime objective of these studies was not to optimize the internal membership testing data structure to be customizable based on the application. And all of them suffer from the intrinsic limitations that affect Bloom Filter and its variants by design.



Authorized licensed use limited to: SAINT LOUIS UNIVERSITY. Downloaded on January 02,2021 at 00:47:54 UTC from IEEE Xplore. Restrictions apply.

## VII. Conclusion

In this paper, we presented an optimization of Cuckoo Filters — a data structure for membership testing — and its performance evaluation when applied to Distributed SDN and NFV applications, highlighting the advantages of using this data structure over alternative membership testing algorithms. We discussed some practical use-cases for this data structure in NFV and SDN and showed that the dynamic nature of our *Optimized Cuckoo Filter* has the potential to improve on a variety of networking applications. The adaptability of OCF is primarily credited to its tune-able policies, which can be adjusted depending on the application requirements. Our results show how this data structure can be integrated with several NFV applications, and we have discussed the trade-offs associated with several filter policy configurations.

## VIII. Acknowledgement

We are grateful to Dr. Erin Chambers for her useful feedback on an initial version of this paper. This work has been supported by the National Science Foundation, under Award Numbers CNS1647084, CNS1836906, and CNS1908574.

## References

[1] Broder, Andrei, and Michael Mitzenmacher. "Network applications of bloom filters: A survey." Internet mathematics 1.4 (2004): 485-509.

[2] Bin Fan, David G. Andersen, Michael Kaminsky , Michael D. Mitzenmacher. "Cuckoo filter: Practically better than bloom." Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. 2014.

[3] Pagh Rasmus, and Flemming Friche Rodler. "Cuckoo hashing." In European Symposium on Algorithms, pp. 121-133. Springer, Berlin, Heidelberg, 2001.

[4] Ivan Sičić, Karlo Slovenec, Lucija Petricioli, Miljenko Mikuc "Comparison of Cuckoo Hash Table and Bloom Filter for Fast Packet Filtering Using Data Plane Development Kit." 2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM). IEEE, 2019.

[5] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian "Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters." Proceedings of the VLDB Endowment 13.2 (2019): 197-210.

[6] Almeida, P.S., Baquero, C., Preguiça, N.M.,& Hutchison, D. , "Scalable Bloom Filters," Inf. Process. Lett., pp. 255–261, 2007.

[7] Khandelwal, Anurag, Rachit Agarwal, and Ion Stoica. "Confluo: Distributed monitoring and diagnosis stack for high-speed networks." In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pp. 421-436. 2019.

[8] Vatche Ishakian, Joseph Akinwumi, Flavio Esposito, Ibrahim Matta "On Supporting Mobility and Multihoming in Recursive Internet Architectures." In: Journal of Computer Communication, Vol. 35, Issue 13, pages 1561-1573, July 2012.

[9] Du, Yan, and Sheng Wang. "Two-stage adaptive bloom filters for per-flow monitoring in software defined networks." 2018 IEEE International Conference on Communications (ICC), 2018.

[10] Islam, Salekul, Nasif Muslim, and J. William Atwood. "A survey on multicasting in software-defined networking." IEEE Communications Surveys & Tutorials 20.1 (2017): 355-387.

[11] Jacobson V, Smetters DK, Thornton JD, Plass MF, Briggs NH, Braynard RL, "Networking named content." Proceedings of the 5th international conference on Emerging networking experiments and technologies. 2009.

[12] Tarkoma, Sasu, Christian Esteve Rothenberg, and Eemil Lagerspetz. "Theory and practice of bloom filters for distributed systems." IEEE Communications Surveys & Tutorials 14.1 (2011): 131-155.

[13] Graf, Thomas Mueller, and Daniel Lemire. "Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters." Journal of Experimental Algorithmics (JEA) 25.1 (2020): 1-16.

[14] Eppstein, David. "Cuckoo filter: Simplification and analysis." arXiv preprint arXiv:1604.06067 (2016).

[15] Fleming, Noah. "Cuckoo Hashing and Cuckoo Filters." (2018).

[16] Almeida, Paulo Sérgio, Carlos Baquero, Nuno Preguiça, and David Hutchison "Scalable bloom filters." Information Processing Letters 101.6 (2007): 255-261.

[17] Lang, Harald, Thomas Neumann, Alfons Kemper, and Peter Boncz. "Performance-optimal filtering: Bloom overtakes cuckoo at high throughput." Proceedings of the VLDB Endowment 12.5 (2019): 502-515.

[18] Lu, Yi, and Prabhakar, Balaji and Bonomi, Flavio, Bloom filters: Design innovations and novel applications, January 2005.

[19] "S/Gi Firewall for Service Providers" F5 Incorporated Blog (2020) https://www.f5.com/services/resources/use-cases/s-gi-firewall-for-service-providers

[20] "Technical Overview of OPNFV platform" OPNFV Project a Series of LF Projects, LLC (2018) https://www.opnfv.org/software/technical-overview

[21] F. Esposito and W. Cerroni. "Integrating Peer and Piece Selection in Content Distribution Networks." In: Proc. of IEEE Global Comm. Conf. (GLOBECOM '15), San Diego, CA, 6-10 Dec 2015.

[22] P. Biondi and the Scapy community, "Scapy: Packet crafting for Python2 and Python3" (2020) https://scapy.net/

[23] Bob Lantz, Brandon Heller, and Nick McKeown (2010) "A network in a laptop: rapid prototyping for software-defined networks", In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX), Association for Computing Machinery, New York, NY, USA, Article 19, 1–6.

[24] Kao, Ming-Yang, "Encyclopedia of Algorithm"s, pp 212–215, Germany, Springer, 2008.

[25] Bhattacharya, Arindam, Srikanta Bedathur, and Amitabha Bagchi. "Adaptive Learned Bloom Filters under Incremental Workloads." In Proc. of the 7th ACM IKDD CoDS and 25th COMAD. 2020.

[26] Mitzenmacher, Michael. "A model for learned bloom filters and optimizing by sandwiching." Advances in Neural Information Processing Systems. 2018.

[27] Kraska, Tim, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. "The case for learned index structures." Proc. of the 2018 International Conference on Management of Data. 2018.

[28] Ozisik, A. Pinar, Gavin Andresen, Brian N. Levine, Darren Tapp, George Bissias, and Sunny Katkuri. "Graphene: efficient interactive set reconciliation applied to blockchain propagation." Proc. of ACM SIGCOMM. 2019.

[29] Kalghoum, Anwar, Sonia Mettali Gammar, and Leila Azouz Saidane. "Towards a novel forwarding strategy for named data networking based on SDN and bloom filter." 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA).

[30] Amadeo, Marica, Claudia Campolo, Antonella Molinaro, and Giuseppe Ruggeri "Content-centric wireless networking: A survey." Computer Networks 72 (2014): 1-13.

[31] Zhao, Jinyuan, Zhigang Hu, Bing Xiong, and Keqin Li. "Accelerating packet classification with counting bloom filters for virtual openflow switching." China Communications 15.10 (2018): 117-128.

[32] Berto, Filippo, Luca Calderoni, Mauro Conti, and Eleonora Losiouk. "Spatial bloom filter in named data networking: a memory efficient solution." Proc. of the 35th Annual ACM Symposium on Applied Computing. 2020.



Authorized licensed use limited to: SAINT LOUIS UNIVERSITY. Downloaded on January 02,2021 at 00:47:54 UTC from IEEE Xplore. Restrictions apply.