Implementing Adaptive Cuckoo Firewall for Programmable Switch

Alex Pedersen, student *Harvard University*
Weiqi Feng, student *Harvard University*
Xiaoqi Chen, student *Princeton University*
Minlan Yu *Harvard University*

Abstract

Stateful firewalls are a critical security function in many networks. To implement a faster stateful firewall using high-speed programmable switch and support a large number of connections, we need to avoid the memory footprint of storing full 5-tuples. We store shorter hash digests in switch memory, while mitigating the effect of false positives by applying *adaptations*. We present a stateful firewall implementation on programmable switches, as well as design choices and benchmarks for adapting to false positives using cuckoo filters.

1 Introduction

Stateful firewalls have been widely deployed to protect enterprises from unsolicited traffic [4]. By tracking ongoing connections, stateful firewalls decide accept or reject each incoming packet. Today's stateful firewalls are usually implemented in a special hardware box to achieve high processing throughput (e.g. Cisco Firewall 9300 [2] provides up to 200Gbps throughput).

In this work, we propose to implement stateful firewalls on a commodity programmable switch. The benefits are that the switch can process Tbps-level traffic and has μs-level packet processing latency. However, the switch's limited memory (e.g. a few MBs on Tofino 1) presents challenges for supporting a large number of connections.

In particular, if we store the full connection 5-tuple for each flow (104 bits for IPv4, 296 bits for IPv6) in switch memory, we can only track a small number of connections. Instead, we could use filter-based approximated data structures (e.g. cuckoo filter [3]) and save only a short digest for each connection to reduce the memory footprint. A filter guarantees no false negative, i.e., we will never erroneously drop packets from an ongoing connection. However, it may suffer from false positives due to hash collisions. With a naive filter, once an attacker discovers any false positive connection ID, it can send an unlimited number of packets under that connection ID to consume network and processing resources and possibly enumerate different kinds of attack.

Adaptive cuckoo filters (ACF) [5, 6] support a new "adapt" operation to mitigate false positives. When the filter is notified of a false positive, it can modify its memory in some way to eliminate this false positive for subsequent queries. The first category of design from [6] adds adaptivity by introducing additional hash functions to rehash entries collided with false positives. The second category of design [5] achieves adaptivity using existing features of the cuckoo filter. In an n-stage cuckoo filter there are n possible locations for each entry. When a false positive collides with a valid entry, that entry is moved to a different stage where there is no collision.

2 Design

In this work we choose to examine the second category [5] of ACF design for several reasons. First, its adaptation process follows the same cuckooing procedure used by a regular cuckoo filter for insertion, which simplifies implementation. Beyond this, Kopelowitz et al [5]. demonstrated that this technique achieves equivalent performance to the first category without increasing memory footprint.

We implement the n-staged cuckoo filter [5] as n register memory arrays in the switch data plane. To support adaptation we introduce a notification scheme: First, we let end hosts send explicit false positive notifications (as outlined in Figure 1), as ICMP port unreachable messages. This notification is forwarded to the switch CPU, which calculates the cuckoo-based adaptation for the filter. The control plane maintains a replica of the filter as well as the complete connection IDs of all ongoing connections, which is necessary for re-calculating alternative indices and digests when cuckooing.

Our ACF implements four fundamental operations:
Lookup: when an ingress packet enters the switch, we calculate n different array indices (one per stage) as well as n different digests, by applying different hash functions over its connection ID 5-tuple. We then read and compare the digest stored at these array indices. If there is any match, we consider the packet a positive and forward it to the end host; otherwise the firewall drops the packet.

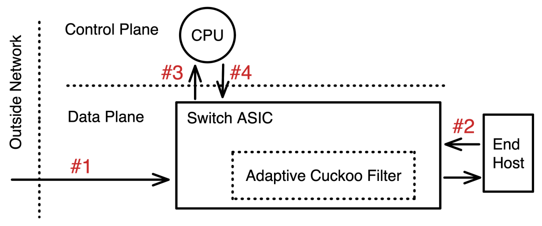

Figure 1: When (#1) a false positive packet reaches the end host, (#2) it replies with a notification. This is (#3) forwarded to the control plane, which then computes the cuckooing adaptations and (#4) propagates the update to the data plane.

Adaptation: when the control plane receives a false positive notification message from an end host, it first uses the connection ID to calculate all the corresponding indices and digests. We then add them to "collision lists" – every bucket in the array maintains a list of digests that will trigger false positives. If an ongoing connection (true positive) has a hash collision at one of the buckets, we mark the placement in this array as disallowed and move this connection to other arrays. To find a way to move the collided connection to other arrays, we consider a graph whose nodes are all the buckets across all arrays, and treat each connection's allowed cuckoo movements as a hyperedge. We then run a breadth-first search from the collided bucket and find a shortest path towards any unoccupied bucket, and apply the corresponding cuckoo movements.
Insertion: when the switch receives an egress packet marking the start of a new connection, the data plane attempts to insert a digest to the corresponding index if it's currently empty, and send a notification to the control plane containing the full connection ID. Otherwise, it notifies the control plane to start a cuckoo-based insertion procedure; this uses the same algorithm as an adaptation.
Deletion: packets marking the end of a connection are forwarded to the control plane, which cleans up the digests.

For the insertion, deletion, and adaptation operations communication with the control plane introduces latency, which could be unacceptable in certain applications. To help mitigate this issue we implement caching and batching of updates. Before notifying the control plane to begin the operation, the data plane first inserts the connection into an overriding block or allow list. This way, if a false positive is discovered, for example, it can be immediately blocked via the block list, even before the control plane is able to apply all updates. An added benefit of this caching mechanism is that it allows the control plane to perform multiple adaptation and insertion requests in batches, further improving performance.

3 Evaluation

Our early evaluation focuses on the number of false positive adaptations our filter design is able to support. As false positives accumulate, a filter eventually cannot adapt to any more false positives, at which point we need to rebuild the filter using different hash function. Supporting more adaptations mean our control plane doesn't need to rebuild as frequently, which also minimizes control plane overhead. Figure 2 shows that as we add more stages to the cuckoo filter, the number of adaptations it can support (after normalizing for the number of total buckets in the filter) also grows.

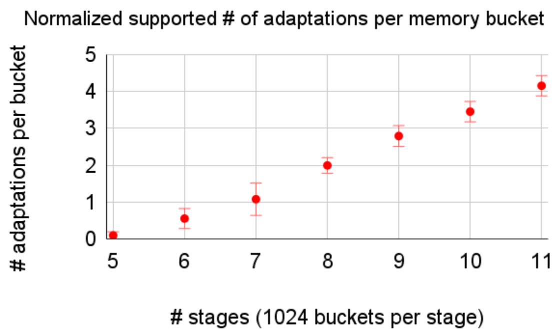

Figure 2: We examine the number of adaptations per bucket a filter with a given number of stages can support. For each trial we fill the filter to 80% occupancy and then begin inducing false positives. As the number of stages increases, we can support more adaptations per bucket.

In our poster, we will present more benchmark results, including comparisons to alternative filter designs on memory efficiency and supported adaptations. We will also use a simulation experiment based on the CAIDA trace [1] to show the false positive rate of approximate cuckoo filter, as in [6], as well as measurements for the control plane overhead and performance penalty of the adaptation algorithm itself.

References

[1] The CAIDA UCSD Anonymized Internet Traces, 2019.

[2] Firepower 9300 series firewall. www.cisco.com/c/en/us/products/collateral/security/firepower-9000-series/datasheet-c78-742471.html, 2023.

[3] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[4] Mohamed G Gouda and Alex X Liu. A model of stateful firewalls and its properties. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 128–137. IEEE, 2005.

[5] Tsvi Kopelowitz, Samuel McCauley, and Ely Porat. Support optimality and adaptive cuckoo filters, 2021.

[6] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters, 2020.