CUCKOOGUARD: A Memory-Efficient SYN Flood Defense Architecture for SmartNICs

[1]Tristan Döring, [2]Kate Ching-Ju Lin, [1]Hung-Min Sun, and [1]Cheng-Hsin Hsu
[1]National Tsing Hua University, Taiwan
[2]National Yang Ming Chiao Tung University, Taiwan

***Abstract*—SYN flood attacks continue to challenge server scalability and availability. Existing defenses either burden the host CPU or exceed the memory limits of programmable hardware. We present CUCKOOGUARD, a memory-efficient SYN flood mitigation architecture for SmartNICs that offloads connection validation to the data plane using a split-proxy design. At its core, CUCKOOGUARD employs Cuckoo filters to enable precise TCP connection tracking with support for the explicit removal of stale connection entries, thereby sustaining high connection churn. Implemented in P4 and benchmarked against the state-of-the-art, CUCKOOGUARD reduces flow filtering false positive rates from 7.66% to 1.56%, yielding a 79% reduction in server CPU overhead. These results demonstrate that accurate, low-overhead SYN flood defense is achievable on resource-constrained programmable platforms.**

I. INTRODUCTION

SYN flood attacks remain a dominant class of Distributed Denial-of-Service (DDoS) attacks, exploiting the TCP handshake to exhaust server CPU and memory resources [1]. Despite widespread deployment of stateless defenses like SYN-Cookies [2], these mechanisms are limited by host CPU capacity and degrade under large-scale attack volumes.

Broader defenses such as SDN-based filtering [3] offer coarse-grained protection, e.g., by blocking spoofed or known-malicious IPs. However, they lack the per-flow granularity needed for effective SYN flood mitigation and are best viewed as complementary solutions.

Recent work has thus focused on in-network defenses implemented entirely in the data plane [4]. One notable example is SMARTCOOKIE [5], which uses a split-proxy design to offload SYN-Cookie validation. While effective, it relies on programmable switch hardware that has since been discontinued by the manufacturer [6], its scalability is limited, and it incurs significant server overhead due to a 7.66% false positive rate in flow filtering.

In this work, we present CUCKOOGUARD, a memory-efficient SYN flood defense architecture designed for deployment on SmartNICs. These widely used devices [7] offer line-rate packet processing near the server edge and support programmable data planes via P4 [8], making them ideal for fine-grained, low-latency traffic handling. We implement a proof-of-concept of CUCKOOGUARD using the P4 language [9] to validate our approach. As a widely adopted domain-specific language (DSL) for programmable data planes, P4 enables hardware-agnostic yet efficient implementations, making it well-suited for deployment across SmartNIC platforms.

Unlike prior approaches such as SMARTCOOKIE [5], which rely on time-decaying Bloom filters for flow filtering, CUCKOOGUARD aims to provide precise and memory-efficient flow filtering. While an array of set-membership data structures were considered, the Cuckoo filter, as introduced by Fan et al. [10], crystallized itself as the right data structure for this role. Therefore, CUCKOOGUARD employs a Cuckoo filter design. This enables precise per-connection tracking with support for dynamic flow entry removal, significantly reducing memory overhead. Our implementation demonstrates that CUCKOOGUARD offers both precision and practicality, particularly in SmartNIC environments where memory resources are limited e.g. due to the need for multiple concurrent network functions [11].

Compared to SMARTCOOKIE, CUCKOOGUARD offers the following contributions:

- **Improved CPU Offload:** Lower false positive rates at the flow filter reduce unnecessary cookie verifications at the server, decreasing server-side CPU overhead by up to 79%.
- **Memory-Efficient Connection Tracking:** Compact and precise connection tracking that supports the deletion of individual flow entries (corresponding to terminated connections) and sustains high connection churn, even when severely memory constrained.

II. RELATED WORK

SYN-Cookies. SYN-Cookies are a lightweight and effective defense against SYN flood attacks, eliminating server-side state during the initial TCP handshake. Instead of allocating memory upon receiving a SYN, the server encodes connection-specific data—such as IP addresses, ports, and timestamps—into the sequence number of the SYN-ACK using a hash. When a valid ACK with the correct cookie is received, the server reconstructs the connection state and allocates resources. Originally proposed by Bernstein [2], this stateless mechanism trades memory usage for additional computational effort (hashing) during connection setup.

ML-based detection. A representative ML-based SYN flood defense [12] uses programmable data planes for feature extraction over aggregated traffic windows, with classification handled in the control plane. Under attack, such designs risk indiscriminately dropping all traffic—including legitimate packets—thus enabling denial-of-service rather than preventing it. While flow-level ML classifiers could offer finer-grained detection, the limited context during the initial handshake makes accurate classification difficult. Moreover, unavoidable false negatives undermine proxy transparency and degrade service quality for legitimate clients.

SYN proxy techniques. SYN proxy approaches validate connections in the data plane using SYN-Cookies, forwarding

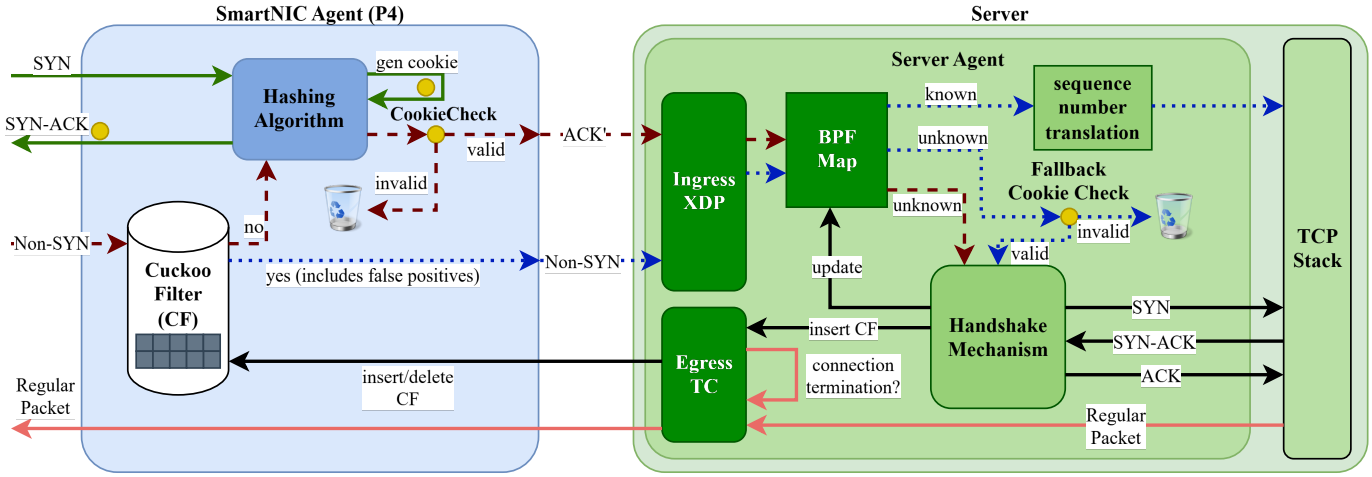

Fig. 1. Deployment of CUCKOOGUARD, illustrating separation of handshake validation and stateful connection management across the SmartNIC and server.

only confirmed flows to the server. Because each participant performs an independent handshake, the proxy must maintain a per-connection state, including the TCP 4-tuple and sequence number offsets. Scholz et al. [4] demonstrate such proxies using P4 across multiple platforms. However, the required fine-grained state tracking incurs significant memory overhead, limiting scalability and deployability on resource-constrained platforms such as FPGA-based SmartNICs and programmable switches.

Split-proxy architectures. SMARTCOOKIE[5] extends the SYN proxy model with a split design for resource-constrained data planes. It offloads SYN-Cookie validation and flow filtering to a programmable switch while delegating sequence number translation and connection state management to a server-side eBPF agent[13]. By avoiding per-connection state on the switch, this approach reduces memory overhead and preserves protocol transparency. However, it introduces trade-offs: coordination with the server adds complexity, and false positives in the flow filter can trigger unnecessary cookie checks during ACK floods, significantly limiting CPU offload. Still, offloading sequence number translation remains a promising direction.

III. CUCKOOGUARD

The architecture accelerates the SYN-Cookie proxy mechanism using a split-proxy design distributed across the SmartNIC and server. The *SmartNIC Agent* and *Server Agent* collaborate to authenticate incoming connections and forward only verified flows. This design eliminates the need for a per-connection state in the data plane, reducing memory consumption while enabling transparency and precise connection tracking. Figure 1 presents an overview of the CUCKOOGUARD architecture, highlighting the roles of key components and the flow of packet processing.

A. Components

SmartNIC Agent. The SmartNIC Agent is fully offloaded to the SmartNIC's data plane, which can sustain line-rate packet processing. The SmartNIC Agent functions as the first line of defense. It intercepts all incoming TCP packets and manages the initial TCP handshake with external clients using a SYN-Cookie-based authentication mechanism.

Upon a successful handshake, the SmartNIC Agent notifies the Server Agent, which returns by inserting the flow entry into the flow filter at the SmartNIC Agent. This filter distinguishes verified (benign) from unverified traffic, ensuring that only known-good flows reach the server. CUCKOOGUARD utilizes a Cuckoo filter [10], which supports the deletion of stale flow entries and offers a superior memory-precision tradeoff (see Section V).

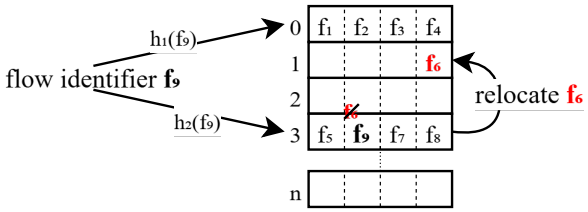

Fig. 2. Insertion into a Cuckoo filter with n buckets using Cuckoo hashing [14] (h_1, h_2)

The Cuckoo filter is realized using a fixed-length register array in the data plane. The structure is organized into buckets, each capable of storing multiple flow identifiers called fingerprints. Each fingerprint is mapped to two candidate buckets via Cuckoo hashing [14], and inserted into one—preferably the one with available space. If both candidate buckets are full, as illustrated in Figure 2, the filter initiates a trial-and-error process in which an existing entry is evicted and relocated to its alternate bucket, potentially triggering a chain of further relocations. While the mechanism supports both deletion and insertion of individual flow entries, the latter requires iterative logic, making it challenging to implement efficiently in data plane environments and necessitating repeated packet recirculation. Nonetheless, as our evaluation shows, this limitation can be effectively overcome.

It is important to note that Cuckoo filters, like Bloom filters, are probabilistic data structures and, therefore, prone to false positives. This means that while the system is *perfectly resilient against SYN flood attacks* (since SYN packets are always intercepted at the SmartNIC), a small fraction of ACK packets from an attacker may still be erroneously forwarded to the server. To mitigate this risk, the Server Agent includes a fallback mechanism to re-verify such packets.

Server Agent. The Server Agent is implemented using two coordinated eBPF programs: one at the ingress (XDP [13]) and one at the egress (TC [15]) of the kernel networking stack. Operating entirely in-kernel, these programs avoid costly user-space context switches and execute with minimal overhead. Both programs share access to a common BPF map [16], which stores the state of currently active connections and the sequence number offsets needed to maintain protocol correctness. The Server Agent fulfills two key responsibilities:

Once a connection request passes cookie verification at the SmartNIC, the SmartNIC Agent adds a setup tag to the final ACK of the TCP handshake. Then, it is forwarded to the Server Agent, initiating a second handshake with the actual TCP server. Due to the server selecting a new initial sequence number independently (as required by the TCP protocol), the Server Agent computes and stores the sequence number differential. This allows future packets of the connection to be correctly translated and relayed. After the handshake is completed, the Server Agent instructs the SmartNIC to register the connection in the Cuckoo filter. When the connection terminates, the Server Agent similarly signals the SmartNIC to remove the corresponding flow entry from the Cuckoo filter.

If an ACK packet that does not correspond to an existing connection and is not tagged as part of a verified handshake reaches the Server Agent, the agent performs a fallback cookie check. If successful, the handshake process is resumed; otherwise, the packet is dropped. Due to cookie-hashing, this fallback path is computationally expensive, and thus, minimizing its invocation is a primary optimization goal. Reducing the false positive rate of the SmartNIC's Cuckoo filter is, therefore, critical and is the central focus of our evaluation.

B. Operations

The operation of the proposed architecture is described with three representative traffic scenarios:

- **Scenario–Benign Client Connection:** A legitimate client sends a SYN, and the SmartNIC replies with a SYN-ACK containing a cookie. Upon receiving a valid ACK, the SmartNIC verifies the cookie and forwards a tagged packet to the Server Agent, which completes the handshake, calculates sequence number offsets, and updates the Cuckoo filter. Subsequent packets are translated and forwarded until teardown when the connection-related flow entry is removed from the Cuckoo filter.
- **Scenario–SYN Flood:** An attacker floods the server with SYNs. The server remains fully protected since the SmartNIC intercepts all of them and discards them unless the cookie is validated.
- **Scenario–ACK Flood:** Forged ACKs attempt to bypass cookie checks. Some may pass the Cuckoo filter due to false positives and reach the Server Agent, which performs a fallback check. Invalid ACKs are dropped, and processing cost remains bounded by the Cuckoo filter's false positive rate.

In summary, CUCKOOGUARD enables transparent, high-performance SYN flood mitigation while minimizing the memory overhead on programmable data plane devices, such as SmartNICs.

IV. IMPLEMENTATIONS

To evaluate our architecture, we implement two representative SYN flood defenses in P4 and run them on the *BMv2* [17] behavioural model. Although P4 is target-independent, BMv2 lacks support for secure hash functions, so all variants use CRC-based hashing [18] for SYN cookie generation. While insecure for deployment, this suffices for evaluating filter behaviour, as cookie security is not the focus. Experiments are conducted on a single commercial off-the-shelf (COTS) machine, using Linux virtual interfaces and namespaces to emulate the split-proxy setup. We denote the number of tracked TCP connections by n and the total filter memory (in bits) by m.

A. SMARTCOOKIE Implementation

Our SMARTCOOKIE baseline is derived from the open-source P4 implementation [19] targeting Intel Tofino, which was ported to BMv2. The implementation uses a split-proxy model: SYN-Cookie generation and flow filtering occurs in the data plane (P4). At the same time, a server-side agent (eBPF) performs sequence number translation and connection state management For flow filtering, we evaluate two Bloom filter variants:

- The *(Ideal) Bloom filter* is a theoretical precision upper bound for standard Bloom filters. The number of hash functions, k, is set as an integer variable, and using linear search, the optimal value minimizing expected false positives can quickly be found (ideal k does not exceed 30 up to filter sizes of up to 1 TB). The expected false positive rate is calculated using the following exact formulation [20]:

$$F_s(n, m, k) = \sum_{i=1}^{m} S(n, m, k, i) \left(\frac{i}{m}\right)^k, \qquad (1)$$

where:

$$S(n, m, k, i) = B(nk, m, i); \qquad (2)$$

$$B(n, m, i) = \binom{m}{i} \cdot i^n \cdot m^{-n}. \qquad (3)$$

To implement k hash functions, we use CRC32 and CRC16 [18] as base functions ($k = 1$ and $k = 2$) and generate further hashes using the method [10]:

$$\text{hash}_k(x) = \text{CRC32}(x) + (k - 2) \cdot \text{CRC16}(x). \qquad (4)$$

This presents an idealized scenario since the number of hash calculations and filter accesses may be limited in real hardware due to resource constraints.

- The *(Ideal) Bloom filter with time-decaying,* enables connection entry expiry by maintaining two (Ideal) Bloom filters in parallel, with alternating resets every interval t. Each uses $m/2$ bit of memory and flows not seen for $2t$ are implicitly removed.

B. CuckooGuard Implementation

This implementation follows the architecture proposed in Section III. The SmartNIC Agent is implemented in P4, and the Server Agent is implemented using eBPF running in the kernel, respectively. The SmartNIC Agent's Cuckoo filter operations require conditional looping for flow insertion, which is implemented via packet recirculation. Packets that trigger insertions are recirculated up to `MaxNumKicks` in the worst case to resolve collisions. The parameters are scenario-dependent, as introduced below.

Cuckoo filter. The Cuckoo filter's constants, such as entries per bucket $b = 4$, load factor $\alpha = 0.95$, and `MaxNumKicks` $= 500$ are chosen according to common recommendations [10]. As a hash function, we use CRC32, which offers good distribution characteristics despite lacking full bijectivity for our input sizes [21]. The fingerprint size parameter f is chosen to minimize the false positive rate under fixed n and m. The number of buckets B can be directly determined from this optimal fingerprint size. Both parameters are determined by:

$$f = \left\lfloor \frac{m \cdot \alpha}{n} \right\rfloor ; \text{and} \tag{5}$$

$$B = \left\lfloor \frac{m}{f \cdot b} \right\rfloor . \tag{6}$$

V. Experiments

A. Setup

All experiments follow a unified setup and parameterization to enable fair comparison of connection-tracking approaches, focusing on memory usage and precision of flow filters rather than absolute throughput or latency. Bloom filter variants are evaluated within the SmartCookie implementation, while Cuckoo filters are tested using CuckooGuard. In addition to the practical, time-decaying Bloom filter, we include an (Ideal) Bloom filter as a non-deployable reference to illustrate the theoretical precision limit. For consistency, decay is disabled in the time-decaying variant (via timeout t) unless stated otherwise.

We fix the number of actively tracked connections at $n = 5000$ and vary the available memory m (in bits), which determines the per-connection memory budget m/n and directly impacts filter precision. Due to BMv2's performance constraints, small values for n and m are used to allow fine-grained variation of m and high-resolution results. While not representative of large-scale deployments, these parameters can be proportionally scaled, preserving filters' relative behavior [10]. To ensure statistical robustness, each measurement was repeated ten times and averaged; false positive rate and recirculation overhead measurements were computed from 100,000 randomly generated spoofed packets per run.

To validate scalability, we adopt a large-scale experimental scenario introduced by Yoo et al.[5] for their SmartCookie architecture. This baseline tracks 579,600 concurrent TCP connections under an ACK flooding attack, using time-decaying, partitioned Bloom filters[22] with $k = 3$ partitions of 2^{20} bits each, totaling approximately 786 KB of memory. We use this setup to quantify the concrete advantages of CuckooGuard under realistic load.

The experimental evaluation focuses on two primary Key Performance Indicators (KPIs):

- **False Positive Rate**, which directly impacts server-side computational load by triggering unnecessary cookie validations during ACK floods.
- **Recirculation Overhead**, which quantifies the number of packet reprocessing cycles required at the SmartNIC (Cuckoo filter Insertion).

B. Results

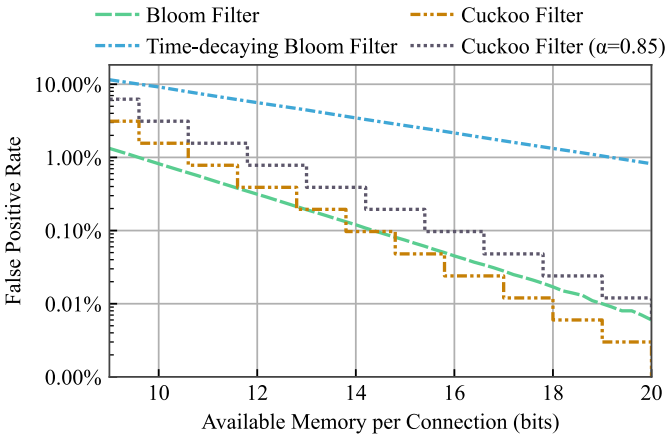

Fig. 3. Memory-precision tradeoff: False positive rate of Cuckoo filters vs. time-decaying Bloom filters across varying memory per connection.

Cuckoo filters offer better precision under low false positive rate requirements. Figure 3 shows the relationship between the false positive rate and the available memory per connection. All Bloom filter variants exhibit gradually improving (decreasing) false positive rates with increasing memory; the Cuckoo filter, on the other hand, displays step-wise improvements. These steps are a direct consequence of fingerprint size adjustments (Section IV): as the available memory increases, the fingerprint length increases discretely, thereby exponentially reducing the probability of false positives due to hash collisions. The Cuckoo filter maintains a significant advantage over all types of Bloom filters up until a false positive rate of $\tilde{0}.1\%$.

It can be observed that SmartCookie's Bloom filter with time-decaying consequently experiences significantly worse precision compared to CuckooGuard's Cuckoo filter. This confirms that CuckooGuard's Cuckoo filter achieves superior precision and remains robust in deployment scenarios where periodic removal of stale connections is essential.

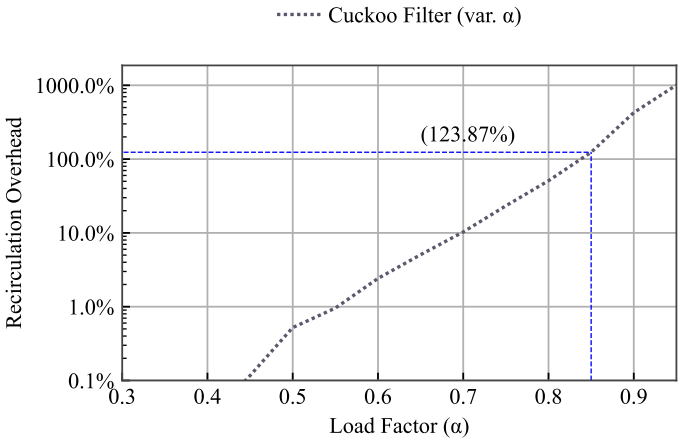

Fig. 4. Insertion overhead of Cuckoo filters increases sharply with higher load factors. Measured as average recirculations per inserted flow entry.

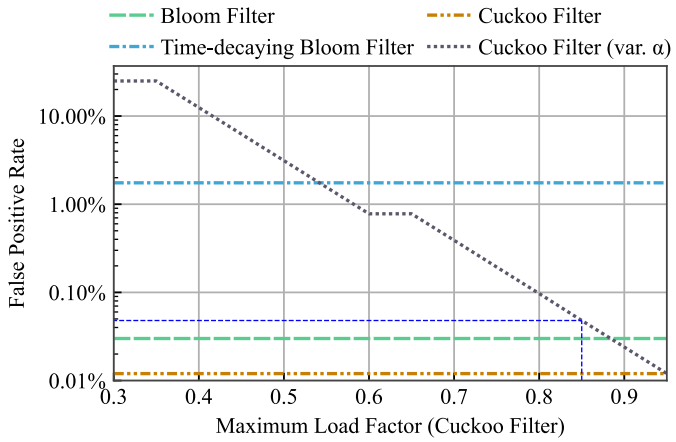

Fig. 5. False positive rate of Cuckoo filters under varying load factors. Higher occupancy improves precision but increases insertion overhead.

Optimized Cuckoo filter insertion overhead has minimal impact on false positive rates. Although prior experiments operated CUCKOOGUARD's Cuckoo filter near its space-optimal capacity ($\alpha = 0.95$) to minimize the false positive rate, this configuration may not be ideal for performance. Therefore, we measure the average recirculation overhead under a fixed memory budget of $m = 84{,}227$ bits while varying the load factor α, which denotes the fraction of occupied entries in the Cuckoo filter. Overhead is quantified as the average number of recirculations per insertion, expressed as a percentage, where 100% indicates one additional pipeline pass per inserted flow entry.

Figure 4 illustrates the recirculation overhead of CUCKOOGUARD's Cuckoo filter across varying load factors. The overhead remains moderate at approximately 124% even at a load factor of $\alpha = 0.85$, but rises sharply beyond this point, exceeding 1000% at $\alpha = 0.95$. This cost must be interpreted in context: flow insertion occurs only once per TCP connection, while most packets are forwarded without recirculation. Consequently, the impact is amortized in long-lived connections but may become significant in high-churn workloads—such as REST APIs or microservice communication—where frequent flow insertions can degrade throughput.

Figure 5 shows the Cuckoo filter's false positive rate under varying load factors compared to the other flow filters. To determine a practical operating point for α, we correlate recirculation cost with an achievable false positive rate. The Cuckoo filter's false positive rate improves monotonically with a higher load, reaching a minimum at $\alpha = 0.95$; even at a conservative 0.85 load factor, it remains just above the idealized Bloom filter and outperforms the time-decaying Bloom filter by a wide margin. This insight suggests that CUCKOOGUARD's Cuckoo filter can be tuned to mitigate its primary disadvantage—recirculation—without significantly compromising precision. A load factor of $\alpha = 0.85$ offers a favorable trade-off, achieving a low false positive rate throughout (see Figure 3) while maintaining reasonable processing overhead. In practice, this corresponds to an average of just over one additional recirculation per TCP connection, making the overhead negligible in most deployment scenarios.

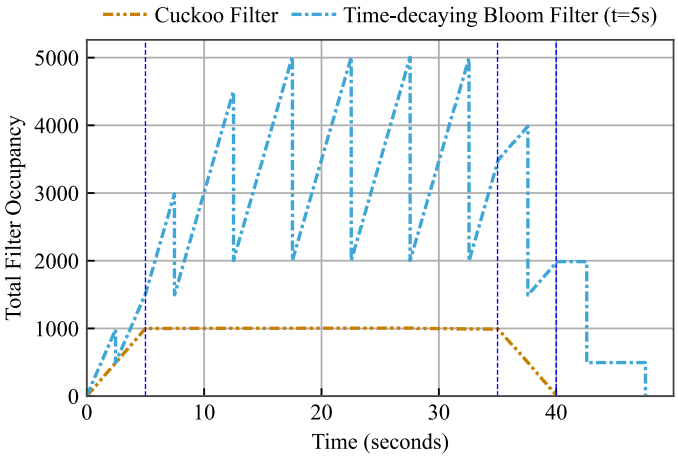

Fig. 6. Temporal behavior of filter occupancy under dynamic connection churn. Cuckoo filters track flows precisely with minimal memory overhead while time-decaying Bloom filters require overprovisioning.

Cuckoo filter ensures minimal overhead and precise connection tracking. Figure 6 shows both designs' flow filter occupancy over time under a workload of 200 new connections per second (*cps*), each lasting exactly 5 seconds. This setup reveals how each filter handles dynamic connection churn across three distinct phases: a *growth phase* (0-5 s), a *steady-state phase* (5-35 s), and a *drain phase* (35-40 s). The Cuckoo filter exhibits linear growth during the initial phase, stabilizing at 1000 flow entries—the number of active connections sustained at 200 *cps* with 5-second lifetimes. During the steady-state phase, flow entry insertions and deletions balance out, keeping occupancy flat. Once new arrivals stop, the filter empties smoothly. This behavior reflects precise connection tracking via explicit deletions triggered by CUCKOOGUARD's server agent.

In contrast, the time-decaying Bloom filter ($t = 5s$) exhibits oscillating occupancy during the steady-state phase, fluctuating between 2000 and 5000 entries with sharp drops every 5 seconds due to instance resets. Each connection's flow entry is inserted into two overlapping filter instances: one is reset every 5 seconds, and both remain active for 10 seconds. The overlap between active and new connections extends the effective tracking window per instance to 15 seconds. At a rate of 200 cps, each instance must support up to 3000 flow entries. During the drain phase, occupancy decreases as existing connections expire. Full clearing occurs once traffic ceases and both instances complete their rotation. Lacking support for flow entry deletion, this Bloom filter must overprovision—each instance holds 3× the active flow entries, and a second instance doubles that requirement, resulting in a 6× flow entry overhead. In contrast, CUCKOOGUARD's Cuckoo filter maintains a tight one-to-one mapping with active connections, offering precise expiration and significantly lower flow entry overhead.

Computational overhead is decreased by 79%. To enable a fair comparison, CUCKOOGUARD is configured to match the memory and connection parameters of the SMARTCOOKIE baseline, as described in Section IV. Both systems are subjected to an ACK flood of 100,000 spoofed, randomized packets across 10 independent runs while tracking 579,600 concurrent TCP connections. In this setting, CUCKOOGUARD achieves an average false positive rate of 1.56%, significantly lower than the 7.66% observed for SMARTCOOKIE, resulting in an estimated CPU overhead reduction of up to **79%**. Notably, the performance gap widens at even lower false positive thresholds (e.g., below 0.1%), highlighting CUCKOOGUARD's scalability.

VI. CONCLUSION

The novel CUCKOOGUARD architecture leverages a dynamic connection-tracking approach centered around Cuckoo filters to achieve superior false positive rates while maintaining minimal overhead for packet processing. Among a range of candidate data structures, the Cuckoo filter was selected for its support of flow entry deletions and strong memory efficiency under constrained environments. Our implementation demonstrated that Cuckoo filters provide substantially better precision than Bloom filter variants. Although insertion requires packet recirculation, this overhead is limited and tunable via the filter's load factor. Experimental results validate the practicality of CUCKOOGUARD, showing up to **79% reduction in server CPU load** compared to the state-of-the-art under equivalent memory constraints.

CUCKOOGUARD can be extended in multiple directions, including: (i) evaluations of packet latency and sustained throughput on real SmartNIC hardware, (ii) integration with other network functions on multi-tenant SmartNIC, and (iii) generalization to semi-sorted Cuckoo filters for trading algorithmic complexity for lower memory usage.

REFERENCES

[1] Cloudflare, "DDoS Threat Report for 2024 Q4," 2024, accessed: 2025-02-04. [Online]. Available: https://blog.cloudflare.com/ddos-threat-report-for-2024-q4/

[2] D. J. Bernstein, "Syn cookies," 1996, accessed: 4 Feb. 2025. [Online]. Available: https://cr.yp.to/syncookies.html

[3] Q. Yan, F. R. Yu, Q. Gong, and J. Li, "Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 602–622, 2016.

[4] D. Scholz, S. Gallenmüller, H. Stubbe, B. Jaber, M. Rouhi, and G. Carle, "Me love (SYN-)Cookies: SYN flood mitigation in programmable data planes," *arXiv preprint arXiv:2003.03221*, 2020. [Online]. Available: https://arxiv.org/abs/2003.03221

[5] S. Yoo, X. Chen, and J. Rexford, "SmartCookie: Blocking Large-Scale SYN floods with a Split-Proxy defense on programmable data planes," in *Proc. of the 33rd USENIX Security Symposium (USENIX Security '24)*, Philadelphia, PA, August 2024, pp. 217–234. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/yoo

[6] Intel Corporation, "Intel® tofino products, pcn 827577-00, product discontinuance, tofino end of life," August 2024, product Change Notification 827577-00. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/827577

[7] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking:{SmartNICs} in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 51–66.

[8] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with P4: Fundamentals, advances, and applied research," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2551–2595, 2021. [Online]. Available: https://arxiv.org/abs/2101.10632

[9] P4 Language Consortium, "P4˜16 language specification, version 1.0.0," https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html, 2020, accessed: 2025-03-28.

[10] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[11] X. Han, Y. Gao, T. Wood, and G. Parmer, "Byways: High-performance, isolated network functions for multi-tenant cloud servers," in *Proc. of the ACM Symposium on Cloud Computing (SoCC'24)*, Redmond, WA, USA, December 2024, pp. 792–810. [Online]. Available: https://faculty.cs.gwu.edu/timwood/papers/24-SOCC-byways.pdf

[12] F. Musumeci, V. Ionata, F. Paolucci, F. Cugini, and M. Tornatore, "Machine-learning-assisted ddos attack detection with p4 language," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 2020, pp. 1–6.

[13] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[14] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *European Symposium on Algorithms*. Springer, 2001, pp. 121–133.

[15] eBPF Documentation Project, "Traffic control (tc) – classifier and action program type," https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_SCHED_CLS/, 2025, accessed: 2025-03-31.

[16] The Linux Kernel Organization, "Bpf maps — linux kernel documentation," https://www.kernel.org/doc/html/latest/bpf/maps.html, 2025, accessed: 2025-03-30.

[17] P4 Language Consortium, "behavioral-model: P4 software switch (bmv2)," https://github.com/p4lang/behavioral-model, 2024, accessed: 2025-03-28.

[18] T. Fujiwara, T. Kasami, and S. Lin, "Error detecting capabilities of the shortened hamming codes adopted for error detection in ieee standard 802.3," *IEEE Transactions on communications*, vol. 37, no. 9, pp. 986–989, 1989.

[19] Princeton Cabernet Group, "Smartcookie," https://github.com/Princeton-Cabernet/p4-projects/tree/master/SmartCookie, 2023, accessed: 2025-03-30.

[20] P. S. Almeida, "A case for partitioned bloom filters," *IEEE Transactions on Computers*, vol. 72, no. 6, pp. 1681–1691, 2022.

[21] P. Reviriego and S. Pontarelli, "Perfect cuckoo filters," in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, 2021, pp. 205–211.

[22] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2011.