HASHMVT: Accelerating Parallel Networking Services with Hardware Offloaded Hash-based Match-Value Tables

Wendi Feng*† *Member, IEEE* and Wei Zhang*‡§ *Member, IEEE*
*College of Computer Science, Beijing Information Science and Technology University (BISTU), China
†State Key Laboratory of Networking and Switching Technology, Beijing University of Posts & Telecommunications, China
‡Advanced Innovation Center for Future Blockchain and Privacy Computing, BISTU, China
§Beijing Laboratory of National Economic Security Early-warning Engineering, BISTU, China

2025 IEEE International Conference on Web Services (ICWS) | 979-8-3315-5563-4/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICWS67624.2025.00033

***Abstract*—Modern network infrastructures rely heavily on *stateful* networking services (NSes), such as firewall and network address translation (NAT). These services utilize state variables (*a.k.a.*, states) to process network flows. However, existing state management mechanisms face significant performance bottlenecks in attaining high-throughput performance, primarily due to the substantial overhead incurred by frequent state *matching* and the inability to *partition states* across instances. To address these challenges, we propose HASHMVT, a novel abstraction optimized for state management in NSes. HASHMVT employs a *Match-Value Table* (MVT) that associates network flows with their corresponding states and incorporates *built-in* partitioning operations that cope with the distribution of states across instances running in parallel. Additionally, it integrates a carefully designed hash-based flow matching mechanism, with hash calculations offloaded to general-purpose platforms using widely available hash instructions. This architecture achieves $O(1)$ time complexity for state lookup operations while supporting concurrent state access via built-in atomic operations, which is particularly crucial for parallel NS deployments. We validate HASHMVT using a representative use case—network address translation. Experimental evaluations demonstrate that HASHMVT exhibits superior scalability compared to existing solutions, delivering over a 60% improvement in throughput compare to the state-of-the-art implementation. Furthermore, comprehensive analyses on adapting HASHMVT to various real-world NSes confirm its generalizability through a systematic examination of state management requirements across different NS categories.**

***Index Terms*—Networking services, state management, scalability, flow matching**

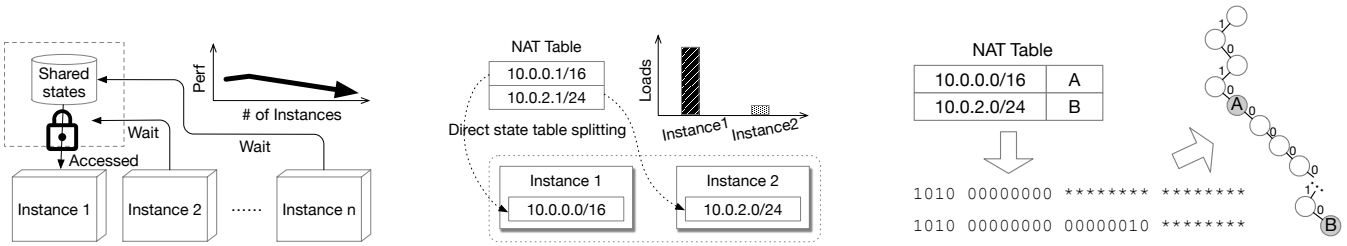
I. INTRODUCTION

Modern network infrastructures increasingly depend on networking services (NSes), such as firewalls and network address translation (NAT), to deliver enhanced service quality. Traditionally, these NSes are implemented as dedicated hardware appliances, commonly referred to as "middleboxes". However, middleboxes suffer from inherent limitations in terms of *functionality customization* and *processing scalability*. Moreover, the static architecture of physical middleboxes poses significant challenges when adapting to rapidly evolving cloud-native network environments, particularly regarding dynamic resource allocation and protocol updates. These constraints have driven an industry-wide transition toward softwarized NS implementations on general-purpose commodity servers. Hence, it has been widely touted as the future of networking. This transformation making *scaling* cost-effective and efficient by running NS instances across additional CPU cores to accommodate growing traffic demands.

These advantages are nevertheless attractive, recent studies have highlighted that the *correctness*, *scalability*, and *robustness* of NSes hinge critically on state management [1]. Take one of the most widely used *stateful* NS, network address translator (NAT), as an example. NAT maintains a table (so as to most NSes) for translating Internet Protocol (IP) addresses and port pairs (*i.e.*, $\langle addr, port \rangle$) of packets. A significant challenge arises from the need to employ parallelism to meet high-throughput requirements [2] while ensuring the correctness of state values without compromising performance. The rationale is, when states are *shared* across parallelized NS instances, the instances contend for access to the same state, necessitating exclusive access mechanisms that incur substantial performance penalties (Sec. II-A). To mitigate this issue, the state table should be partitioned so that each partition's state variables are accessed by only one NS instance. Another challenge is that states exhibit varying *scopes*, which represent the proportion of packets associated with a given state variable. For example, each state variable in a NAT table typically corresponds to a single (5-tuple) flow, referred to as a *per-flow* state, whereas each counter state variable in a sketch-based network measurement service [3] may correspond to multiple flows and is therefore termed a *cross-flow* state. Partitioning a state table without considering the scope of each state may lead to load imbalances across instances, ultimately degrading overall performance (Sec. II-B).

State-of-the-art state management schemes [1], [4]–[9] can be classified into two categories: *shared nothing* and *sharing* approaches. They both fail to efficiently handle states, *e.g.*, in shared-nothing schemes [1], [6]–[9], the state table is partitioned among instances without considering state scopes, and flows are dispatched to the instance that holds the corresponding state. Although this approach avoids state sharing, it often fails to maintain load balance across instances due to the skewed distribution of network traffic, leading to suboptimal performance. In contrast, state-sharing approaches [4], [5] employ shared memory or centralized datastores to simplify

2836-3868/25/$31.00 ©2025 IEEE
DOI 10.1109/ICWS67624.2025.00033



(a) Shared states degrade performance. (b) Uneven loads due to scope unawareness. (c) Matching a state with trie is slow.

Fig. 1: Issues in existing work.

state access across instances; however, these methods suffer from significant performance degradation as a result of the overhead associated with state sharing.

In this paper, we propose a novel abstraction for NSes – the *Hash-based Match-Value Table* (HASHMVT) – designed to tackle the state management challenges discussed above. HASHMVT is an Match-Value Table (MVT) that incorporates comprehensive *scope-aware* partitioning operations alongside a carefully designed hash-based traffic matching mechanism. Akin to the flow table in OpenFlow/SDN [10], HASHMVT captures the relationship between states and flow with a matching field and a value field. The matching field utilizes the hash-based flow matching mechanism to identify the corresponding state, and the value field stores state-related information. Our hash-based mechanism distinguishes HASHMVT from conventional flow tables by leveraging widely available CPU instructions on commodity general-purpose platforms, achieving $O(1)$ time complexity for flow matching. Furthermore, the integrated partitioning operations can detect and subdivide large-scope states, thereby mitigating load imbalance issues.

To demonstrate the benefits of our proposed abstraction, we design and implement HASHMVT as a proof-of-concept. We use network address translation (NAT) as an example because it employs both per-flow states (*e.g.*, the NAT table) and cross-flow states (*e.g.*, the resource pool), thereby illustrating the generality of the HASHMVT abstraction. Additionally, we survey real-world NSes and classify their states into five distinct types, discussing how MVT can effectively represent each type. In summary, our contributions are threefold:

- We propose the HASHMVT, a novel abstraction for state management in NSes, which eliminates shared states across instances and enables $O(1)$ time complexity for flow matching on general-purpose platforms.
- As a proof-of-concept, we design and implement HASHMVT, with experimental evaluations showing that it outperforms state-of-the-art schemes by more than 60%.
- We conduct a survey of real-world NSes, classify their states into five categories, and discuss both the generality and limitations of the HASHMVT abstraction.

II. MOTIVATION AND RELATED WORK

Our work contributes to a theoretical state management scheme, as state management is still an impediment with many untethered problems. This section first motivates the problems with examples and briefs state-of-the-art solutions.

A. Stateful NSes and Curses of State Sharing

Due to its stateful nature, NS processing can be formally represented as $(states_{new}, pkt_{out}) = NS(state_{cur}, pkt_{in})$. In multi-instance deployments, states may be accessed simultaneously across instances. Because correct packet processing critically depends on maintaining state consistency, exclusive access is required. However, as illustrated in Fig. 1a, one instance must wait until another releases the state to ensure exclusivity, resulting in serializing parallel operations and degrading overall performance, which can make NSes fail to comply with the required Service Level Objective (SLO). Hence, sharing must be eliminated.

B. State Scope and Imbalanced Loads across Instances

NSes often process flows at varying granularities. For example, in a NAT table, one state may match source IP addresses within the range `src:10.0.0.0/16`, while another state may match the more specific range `src:10.0.2.0/24` (see Fig. 1b). Consequently, the former encompasses $2^8\times$ times more flows than the latter. If the NAT table consists solely of these two states entries and is deployed across two instances, each managing one state, one instance will handle 256 times more flows than the other, leading to significant load imbalance. These load imbalances can result in overload of some instances leading to packet drops or underuse of other instances, thus, failing to attain the line rate processing speed for the underlying hardware infrastructure.

C. Inefficient State Matching

NSes are required to achieve 100 Gbps throughput nowadays [2]. To meet stringent performance requirements, matching the corresponding state for each flow must be swift. NSes employ flow classification algorithms to locate the appropriate state variable within a state table. Current approaches primarily utilize the Longest Prefix Match (LPM) algorithm, implemented using trie or decision tree structures to represent state matching rules. As illustrated in Fig. 1c, matching state "A" requires traversing 16 trie nodes, whereas matching state "B" requires traversing 24 nodes. Consequently, this decision tree structure necessitates multiple rounds of memory access, incurring remarkable memory access penalty. Another widely adopted approach is exact matching, where the packet fields (used for matching) are used for hashing, and the hash value is then used to index the corresponding state. However, this

approach has to create a state item for each flow. When the total number of flows is gigantic (*e.g.*, millions of flows), it can induce substantial memory usage, leading to frequent cache replacements due to limited cache capacity on a CPU core. Another drawback of this approach is that it cannot provide hints about the overall matched range of states on an instance. Losing this information, the traffic dispatcher may be unable to promptly conduct load balancing, resulting in load imbalances.

D. Related Work

State-of-the-arts lack a concise state abstraction with scope-aware fine-grained table partitioning operations while fail to provide a *fast* state matching mechanism for general-purpose platforms, leading to impaired performance. Some solutions fail to acknowledge the shared state problem. For example, NFOS [1], RSS++ [11] and ScaleFlux [6] only consider per-flow states but ignore cross-flow states that is commonplace in real-world scenarios (*e.g.*, the resource pool in NAT and sketch-based network measurement services [3]). Most solutions lack concise abstractions. For example, LibVNF [12] and SNF [13] contributes to finer-grained state management mechanisms by further categorizing states into various kinds and employing different mechanisms accordingly. However, they require developers to be proficient in parallel programming, raising the bar for scalable NS development. Some solutions avoid complex state management by externalizing the states, making the NS itself "stateless". For example, CHC [4] and SPRIGHT [14] utilize a (conceptually) centralized datastore to save states, and all instances access the same datastore. However, the centralized datastore itself have to maintain exclusive access mechanisms, which is a performance bottleneck while external state accesses negate the goal of achieving desired high throughput (*e.g.*, 100 Gbps or above) [2], [15]; and it can also become a single point of failure. Moreover, other scheme [16] considers splitting state tables with a decision tree-based traffic classification algorithm. However, it still suffers from impaired performance due to its $O(\log N)$ time complexity.

Therefore, we propose a novel abstraction for state management in scalable NSes, which simplifies the scalable NS development with the splitable table design. Its *shared-nothing* design significantly reduces the performance overhead introduced by state-of-the-arts, and the hash-based flow matching mechanism further pushes performance boundary at the constant time complexity level. The remainder of this paper describes the design and evaluation of our proposed abstraction using NAT as an example.

III. HASH-BASED MATCH-VALUE TABLE

This section first presents our new state abstraction, the Match-Value Table (HASHMVT), and its operations. We then detail how the hash-based matching mechanism is designed followed by representing NAT's state using HASHMVT.

A. Match-Value Table Definition

An HASHMVT is an ordered set of match-value pair entries. Given an HASHMVT T, we have $T = \{e_1, e_2, \ldots, e_n\}$.

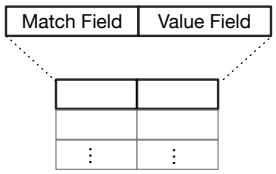


Fig. 2: The Match-Value Table entry.

Each HASHMVT entry $e = \langle M(e), V(e) \rangle$, as depicted in Fig. 2, consists of two fields: a *matching* field $M(e)$ and a *value* field $V(e)$. The matching field matches the packets related to the state. Given a table $T = \{e_1, e_2, \ldots, e_n\}$, $\langle * \rangle = \bigcup_{i=1}^{n} M(e_i)$, where $\langle * \rangle$ denotes all incoming traffic. We call $M(T)$ the mapping range of table T. The value field $V(e)$ stores the state value that comprises five parts: the *state value*, a *timestamp*, a *timeout* value, a *validity flag*, and an *auxiliary* field. The timestamp and timeout parts are set when the entry is created. The validity flag is set when the entry times out, and invalid entries are periodically cleaned. The auxiliary field is a reserved field.

B. Operations

The HASHMVT framework provides a series of operations that can be classified as *table* and *entry* operations.

1) Table Operations: Table operations change the cardinality of an HASHMVT. Basic operations include creating/removing an entry (*i.e.*, "`new`", "`remove`" operations); distributing/combining entries to/from an/many HASHMVT(s) (*i.e.*, "`shard`", "`combine`" operations); and splitting/merging one/two entrie(s) into/as two/one entrie(s), respectively (*i.e.*, "`split`", "`merge`" operations).

Let $\Rightarrow$ denotes the shard operation. Hence, partitioning entries of table T is written as $T \Rightarrow \{T_1, T_2\}$, where $M(T) = M(T_1) \bigcup M(T_2)$. Accordingly, given two HASHMVT tables T_1, T_2, the combine operation is the union of the tables written as $T = T_1 \bigcup T_2$. The "`split`" and "`merge`" operations are much complicated. Let $\rightarrow$ denotes the "`partition`" operation, and suppose $T = \{e\}$. The "`split`" operation first splits e as $e \rightarrow (e'_1, e'_2)$, where $M(e'_1) \bigcap M(e'_2) = \varnothing$. Then, T is split as $\{e\} \Rightarrow \{T'_1, T'_2\} = \{\{e'_1\}, \{e'_2\}\}$. When merging two tables T'_1 and T'_2, the "`combine`" operation is first conducted, and we have $\{e'_1\} \bigcup \{e'_2\} = \{e'_1, e'_2\}$, where $T'_1 = \{e'_1\}$ and $T'_2 = \{e'_2\}$. Let $+$ be the "`merge`" operation, it checks whether $\exists\ e' \in \bigcup_i T_i$ globally, such that $M(e')$ is located between $M(e'_1)$ and $M(e'_2)$. If no, we have $\{e'_1 + e'_2\} = \{e\}$; and otherwise, the "`merge`" operation fails.

2) Entry Operations: Entry operations change $V(e_i)$ of an entry e_i. Different types[1] of HASHMVTs may have some distinct entry operations (*e.g.*, "`set`", "`get`", "`inc`", "`timeout`", and "`renew`" operations) but are mostly identical. For example, "`set`", "`get`", and "`inc`" operations update/read the state value. The "`timeout`" operation invalidates an entry if it times out, and the "`renew`" operation sets a new timeout value to extend its availability. We detail the entry operations of NAT's states in Sec. III-D.

[1]Different NSes have various types of states, which can be represented in different types of the HASHMVT, repectively.

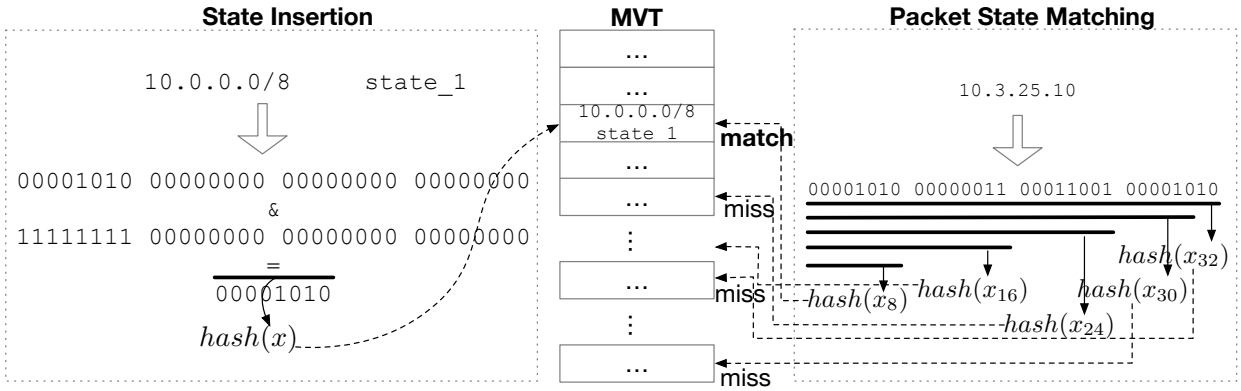


Fig. 3: Design of $M(e)$.

C. Hash-based Matching with Hardware Accelerations

$M(e)$ employs hashing to match a packet to their corresponding states. The core idea involves extracting the relevant packet field and computing multiple hash values in parallel using different bit-lengths of the field. Additionally, $M(e)$ maintains a State Cache to store frequently accessed states, thereby reducing state access latency. As depicted in the right side of Fig. 3, ❶ HASHMVT computes a hash value using the full bit-length of the field (*i.e.*, $hash(x_{32})$), which is used to index the State Cache. If the required state is not cached, ❷ HASHMVT simultaneously calculates multiple hash values with progressively fewer bits, *e.g.*, $hash(x_{30})$, $hash(x_{24})$, $hash(x_{16})$, and $hash(x_8)$. The subscript denotes the number of bits utilized. These hash values are used to index the HASHMVT. Next, ❸ HASHMVT matches the state when the packet's field aligns with the state's rule (*e.g.*, `10.0.0.1/8` in Fig. 3). Finally, ❹ HASHMVT replicates the matched state into the State Cache at the corresponding position, enabling fast state access for subsequent packets within the same flow.

Another use case for $M(e)$ is dynamic state insertion, which occurs when states are generated on-the-fly. To insert a new state into the HASHMVT, HASHMVT first derives the matching bits (*e.g.*, the first 8 bits of the IP address in Fig. 3). These bits serve as input to compute the hash value, which then indexes a specific row in the HASHMVT. Note that multiple rules may map to the same row. For example, state A is `10.0.0.0/8 A`, and state B is `160.0.0.0/4 B`. Both states share identical prefix bits (10) for indexing. To resolve such collisions, we employ a linked list-based collision resolution mechanism. When conflicting rules occupy the same row, the system stores all matched rules in a linked list, enabling subsequent packet processing to validate the correct rule during state matching. Our experiments demonstrate that such collisions are statistically rare in practice (see Sec. VI-B).

Hashing is a critical operation in $M(e)$, and existing work demonstrates that its computational overhead can significantly impact performance [17]. To address this, we benefit from hardware acceleration: instead of relying on software-based algorithms, we offload hash computations to dedicated hardware circuits. This is because modern CPU architectures (*e.g.*, SSE4.2 on Intel chips [18] and CRC Extension on ARM chips [19]) integrate native support for Cyclic Redundancy Check (CRC) operations. By utilizing hardware-accelerated CRC instructions for hash computation, $M(e)$ minimizes software-induced latency. As demonstrated in Sec. VI-C, this approach achieves a $5\times$ acceleration in hash calculation compared to software-based implementations.

D. Abstracting NAT's States

In this subsection, we introduce how the HASHMVT neatly represents NAT's states. We show the HASHMVT can fuse the traffic states[2] with a single table, and the table is partitionable for scalability. Existing work [4], [12], [20] argues that the resource pool is a cross-flow state variable and cannot be partitioned, which should be shared across all instances. We show how the HASHMVT can reconstruct the resource pool to avoid sharing the resource pool, and we demonstrate how the HASHMVT cleverly fuses the NAT table and resource pool as a single HASHMVT table.

1) Representation: NAT consists of two types of states: a *NAT table*, which records the mapping of internal and external IP address pairs, and a *resource pool*, which holds vacant IP addresses and port numbers. It is straightforward to convert the NAT table to an HASHMVT thanks to its structure, where the matching field is an address pair and the value field is the corresponding mapped address pair. However, adapting the resource pool to work with the HASHMVT requires modifications.

The resource pool keeps track of all address resource usage (*i.e.*, available and occupied $\langle IP_E, Port_E \rangle$ pairs), and traditional implementation (*e.g.*, VigNAT [21]) utilizing a range-based variable to represent this pool. When the NAT service is scaled out into multiple instances, the optimal approach for placing states is to partition them and assign each shard to an instance. However, since a single variable cannot be partitioned but instead must be replicated, this limits scalability and performance. As a result, sharing the resource pool among instances becomes necessary but can bottleneck performance.

We take a different approach that enumerates all resources available in the resource pool to the not yet matched HASHMVT entries, which allows to be partitioned. **We argue that**

[2]We use states to represent traffic states in the paper if not pointed out.

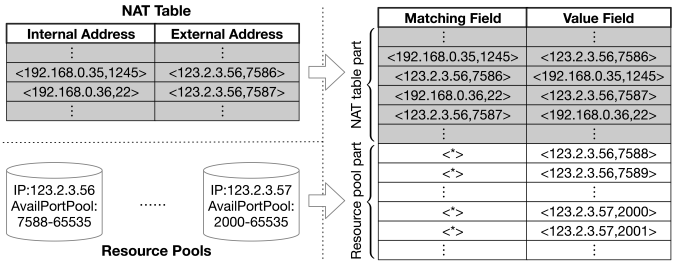


Fig. 4: Representing NAT's states with HASHMVT. HASHMVT consolidates the NAT table and the resource pool, where the gray-shaded part is the NAT table, and the white part is the resource pool.

this approach could exhaust the memory resources of the computer system. for the following three reasons. **i)** The available resource is configured by the network operator based on the number of devices in the network. **ii)** the NAT service should be capable of supporting all the resources configured by the network operator. **iii)** Modern commodity servers are now equipped with DRAMs that can provide ample memory resources, often with capacities of hundreds of gigabytes. This makes partitioning the resource pool feasible. However, another crucial consideration must be resolved to retrofit the resource pool to HASHMVT, which is *determining the matching field for these entries.*

Each available $\langle IP_E, Port_E \rangle$ pair is mapped bidirectionally to an internal $\langle IP_I, Port_I \rangle$ pair whose matching field is $\langle * \rangle$. When a pair is allocated, two steps are conducted. **i)** The available entry's matching field is changed to the new external pair. **ii)** The reverse direction mapping entry is created simultaneously. When all pairs are occupied, the HASHMVT becomes a purely NAT table. As a result, we can conclude that a single HASHMVT can represent both the NAT table and resource pool.

2) NAT's HASHMVT *Entry Operations:* When NAT starts up, the HASHMVT is initialized as a pure resource pool, which is a table of $\langle IP_E, Port_E \rangle$ address pair resources. Let e_i be any entry in the table, and we set $M(e_i) = \langle * \rangle$ for the initialization.

When a new internal host joins the network and begins to transmit traffic, the traffic is matched to the first entry e, whose matching field is represented by $\langle * \rangle$. The NAT changes $V(e)$ to the traffic source IP address and port number using the "`set`" operation. To facilitate the reverse direction of traffic, a corresponding entry e' is "`create`'d", where $M(e') = V(e)$ and $M(e) = V(e')$. Both e and e' have the same timestamp and timeout when e is "`set`", and e' is "`create`'d",

When performing address translation for packets from existing mapped flows, the value field of the matched entry is retrieved from the HASHMVT using the "`get`" operation. During this process, the validity, timestamp, and timeout of the entry are also checked. If the validity flag is invalid, the "`get`" operation returns an error, and a new address resource is used to create a new mapping. If the entry has timed out based on its timestamp and timeout, but the validity flag is valid, the validity flag is set to invalid using the "`timeout`" operation. Once the entry has been read, it is "`renew`'ed", thereby extending its lifetime. If an entry becomes invalid, its matching field is set to $\langle * \rangle$ and the corresponding reverse entry is "`remove`'d".

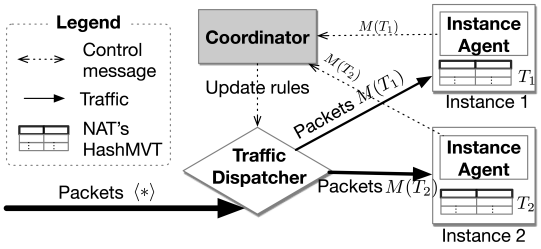


Fig. 5: Overview of the HASHMVT-based networking system.

IV. HASHMVT DESIGN

This section describes the design of our MVT-based NAT. We will begin by providing a brief introduction to HASHMVT, followed by a detailed analysis of each of its components.

A. Design Overview

HASHMVT is comprised of three main components: *NAT instances*, a *coordinator*, and a *traffic dispatcher* (TD). As shown in Fig. 5, incoming traffic is first directed to the TD, which then steers packets to the appropriate instance. Each instance is allocated a shard of the entire HASHMVT, and the matching range of the HASHMVT corresponds to the traffic dispatching rules that are configured on the TD. The coordinator receives "heartbeats" messages from each instance agent, which monitors the number of active entries in each instance's HASHMVT, as well as the instance's load information. The coordinator then determines whether a new instance is necessary. If so, HASHMVT employs the "`shard`" operation to divide the original HASHMVT and update TD's traffic dispatching rules accordingly (detailed in Sec. IV-B).

B. Traffic Dispatcher

The traffic dispatcher (TD) applies traffic dispatching rules to determine the instance of processing a packet. HASHMVT can be deployed either as a single server or as a cluster. In a single server deployment, modern network interface cards (NICs) are capable of implementing the traffic dispatching rules, allowing the NIC to automatically route packets to the corresponding CPU cores (each running an instance). In cluster deployments, software-defined networking (SDN) technology is used, and HASHMVT can update or install forwarding rules (*e.g.*, OpenFlow rules) on the SDN switch for flow steering.

To avoid the expensive state migration process [22], we use a migration-free strategy [23] for traffic dispatching during scaling. The TD uses dispatching rules to steer existing flows to the current instance, while new flows are steered to an available instance that is not overloaded and has sufficient address resources. This approach enables efficient and scalable traffic management without the overhead of migrating states.

C. Instance Agent

To monitor the load on each instance, we use an instance agent that runs alongside each instance and collects load information every second, including *CPU utilization* and *memory usage*. Because many systems rely on the polling model to retrieve packets from the network interface card (NIC), the operating system always shows 100% utilization even when the system is not fully loaded. To address this issue, we use the ratio of CPU cycles of processing packets to the total number of CPU cycles, as described in Equation 1. This approach provides a more accurate measure of system load and helps ensure optimal performance.

$$\eta_{CPU} = \frac{C_{pkt}}{C_{Total}}, \tag{1}$$

where η_{CPU} denotes the CPU utilization, C_{pkt} represents the CPU cycles used for pulling packets from the RX queue, and C_{Total} is the total CPU cycles. Memory usage is calculated by counting all active NAT table entries in the HASHMVT, as the size of each entry is determined after deployment.

The matching range of an instance is defined as the union of the matching fields of every entry in the HASHMVT on the NAT instance, as explained in Sec. III-A. To determine the matching fields for each instance, instance agents collect information about the entries on their respective instances and send it to the coordinator. However, the flow table capacity of both switches and NICs is limited[3], so to optimize performance, the coordinator uses a flow table aggregation mechanism [24] to reduce the number of entries. We describe this aggregation mechanism in Sec. IV-D3.

D. HASHMVT Coordinator

In this subsection, we introduce the coordinator, which plays a crucial role in the system. It acts as the central "brain" that retrieves information from instance agents, judges when and how to scale out an instance, aggregates matching ranges, and updates the forwarding rules accordingly.

1) Communicating between Instance Agents and the Traffic Dispatcher: The coordinator acts as the central point of communication among system components by sending and receiving "heartbeat" messages with instance agents. It makes decisions on when to scale out an instance and submits updated dispatching rules to the TD. The "heartbeat" message contains two types of information: the instance's current *load information* and its *matching range*.

Instance agents collect load information and matching range information from the instance they monitor and send it to the coordinator every second using the UDP `socket`. To mitigate congestion, the message is sent out-of-band. The matching range information is formatted as a list of match fields, and the coordinator aggregates it to reduce the number of entries. If the coordinator fails to receive the "heartbeat" message from an instance for two[4] consecutive periods of four seconds each, it is assumed that a failure has occurred, and the failure recovery process is initiated. The paper does not cover failure recovery, and we rely on existing techniques [4], [25] for failover. When the coordinator decides to scale out an instance, it updates the traffic dispatching rules on the TD to allow the new instance to receive packets.

2) Handling Scaling Out: We employ a threshold-based approach to determine when to scale out. As previously proposed in existing work [26], we profile a single NS instance on a hardware platform (such as a single commodity server or a cluster of servers) to obtain threshold performance metrics. Although the profiling process is currently performed manually, we are working on developing automatic NS profiling techniques as part of our ongoing and future research. If either the CPU load or memory usage metrics reach their respective threshold, the system initiates a scaling-out procedure.

HASHMVT uses a migration-free mechanism for scaling out, which preserves the affinity between old flows and their respective instances, while new flows are dispatched to new instances based on TD's dispatching rules. This approach eliminates the overhead of traffic migration. We allocate 20% of the resource pool to the old instance, as new flows can still be assigned to the existing instances during rule updates. Additionally, the NAT table is completely preserved at the old instance to maintain the affinity between existing flows and their respective instances.

The process of scaling out an instance involves five steps: **i)** starting the new instance; **ii)** redirecting incoming packets to the currently assigned instance; **iii)** transferring the resource pool section of the old instance's HASHMVT to the new instance; **iv)** updating the relevant traffic dispatching rules; and **v)** steering new flows to the new instance.

3) Matching Range Aggregation: We employ an efficient algorithm for aggregating flow tables [24] to reduce the number of entries in the matching range. Unfortunately, the algorithm [24] only supports the aggregation of flow table entries. Therefore, after obtaining all entries' matching ranges, we convert them into a flow table whose action fields are set as destination instance of the flow (*i.e.*, instance ID). We then apply the flow table aggregation algorithm to obtain the optimized table.

V. IMPLEMENTATION

We implement HASHMVT on a commodity server using the Data Plane Development Kit (DPDK) [27], a high-performance kernel-bypass framework optimized for cache-friendly data structures. Each HASHMVT instance and its coordinator operate as a thread pinned to a dedicated physical CPU core[5], with a lightweight instance agent thread co-located on the same core to periodically collect loads. These agents execute at 1-second intervals and forward aggregated metrics to the coordinator for load balancing. Flow dispatching is hardware-accelerated via the NIC: we offload traffic distribution (TD) rules to the NIC using DPDK's Generic Flow API [28], ensuring flows are directly routed to their assigned HASHMVT instance. For the NAT implementation, we adapted the VigNAT [21] codebase due to its compliance with RFC 3022 [29] and rigorously verified correctness.

[3]For example, the NIC used in our testbed supports installing 9325 rules.

[4]Configurable according to different network conditions.

[5]Hyper-threading is disabled.

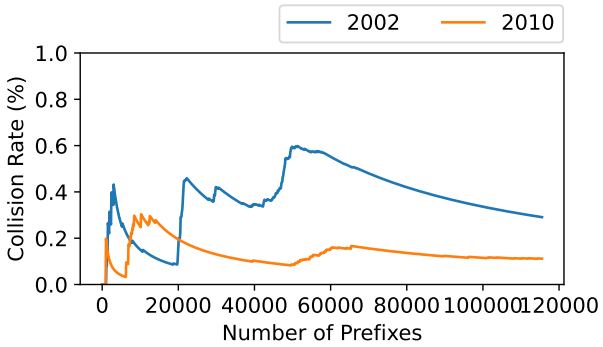


Fig. 6: Collision rate of Matching. Data source from RIPE NCC [31] of years 2002 and 2010.

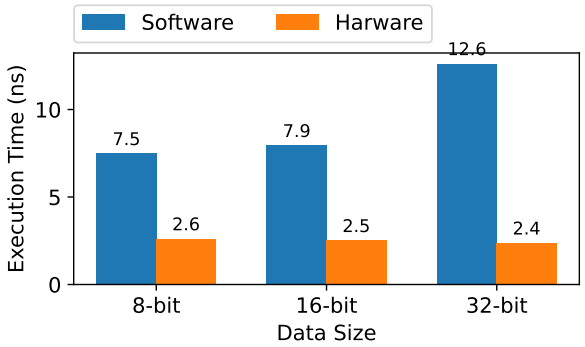


Fig. 7: Performance comparison between software and hardware based hashing.

VI. Experiments

A. Testbed Setup

We use a testbed comprising two NUMA architecture servers for our experiment. One server is used for generating packets, while the other run HashMVT. Both servers are equipped with dual-port Mellanox ConnectX-5 EN 100 Gbps NICs supporting DPDK. They have 384 GB DRAM and one Intel Xeon Platinum 8168 processor with 24 cores, running at 2.7 GHz. We connect the servers directly back-to-back. To isolate the CPU cores, we modify the kernel parameters[6] on both servers, preventing the Linux kernel from using those cores for scheduling. We reserve core 0 for the operating system and disable hyper-threading on both servers. On the device under test (DUT) server, we use Ubuntu 18.04.2 and allocated 128 hugepages, each with a size of 1 GB, for packet processing. To generate traffic for our experiment, we use TRex [30], one of the few software-based traffic generators capable of generating 100 Gbps traffic and can fully utilize our 100 Gbps high-performance NIC.

B. Insertion Collisions of HashMVT

We evaluate collisions during state insertions in the HashMVT, as discussed in Sec. III-C. The experiment leverages multiple datasets of backbone network routing data (*e.g.*, RIPE NCC [31]). Fig. 6 illustrates the collision rate as the number of matching rules increases, with the observed maximum collision rate remaining below 0.6%. The sporadic spikes in collision rates because collisions are confined to specific rules, indicating their statistical rarity. Moreover, the inset boxplot in Fig. 6 depicts the distribution of collision counts, where small circles denote outliers. This visualization further confirms that the number of collisions is extremely low across most rule configurations. Based on these results, we conclude that the collision rate in $M(e)$'s state insertion mechanism is operationally negligible.

[6]Kernel parameters on both servers are set to the following: isolcpus=1–23 intel_idle.max_cstate=0 processor.max_cstate=0 intel_pstate=disable nohz_full=1–23 rcu_nocbs=1–23 rcu_nocb_poll default_hugepagesz=1G hugepagesz=1G hugepages=128 audit=0 nosoftlockup vt.handoff=1

C. Hardware-based Hashing Performance

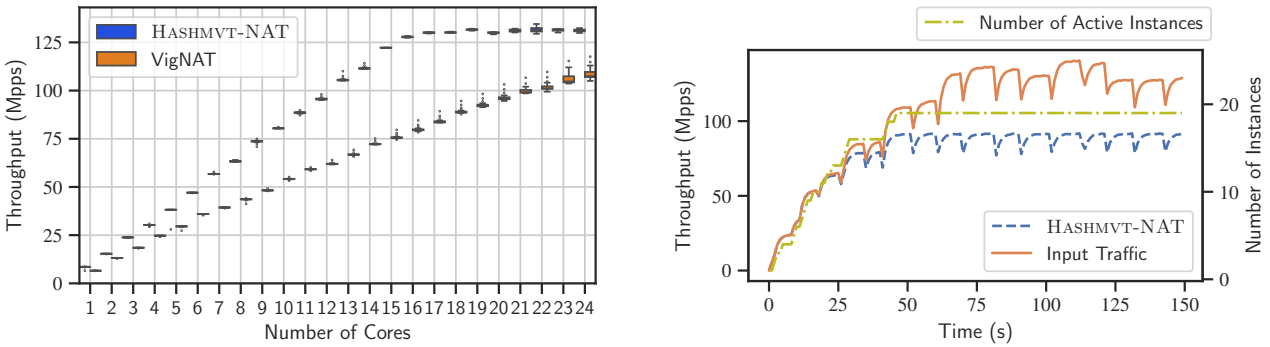
We evaluate the efficiency of $M(e)$ by comparing software-based hashing with hardware-accelerated hashing. For the software approach, we implement a global lookup table-based CRC algorithm [32], which precomputes hash values to eliminate repetitive calculations. For hardware acceleration, we leverage the Intel SSE4.2 instruction set to compute hash values directly via dedicated hardware circuits. As shown in Fig. 7, the hardware-accelerated method achieves a $5\times$ speedup over the software-based approach. This performance gain underscores the efficacy of offloading compute-intensive hash operations to specialized hardware units.

D. Scaling Performance

We evaluate the scalability of our approach by measuring the throughput of HashMVT under different numbers of NAT instances. We conduct 20 tests for each configuration. To compare the performance of HashMVT with existing mechanisms, such as VigorNAT, we measure their throughput. In the results presented in Fig. 8a, VigorNAT's resource pool is shared across instances. Our experiments show that HashMVT outperformed the VigorNAT implementation, achieving up to a 64.2% improvement in performance (using 16 cores). VigorNAT needs to utilize additional cores in order to achieve a performance level that is closer to the line-rate. With 64 B small Ethernet packets and 16 cores, HashMVT achieves a throughput of approximately 130 Million packet per second (Mpps). Besides, HashMVT's throughput increased linearly with an increase in the number of instances, thanks to the elimination of shared states. Hence, HashMVT is proved to be scalable. In contrast, VigorNAT's throughput increase flattened as the number of instances increased. This trend is due to higher overhead resulted from more instances sharing the state, which degrades the overall performance. When using 24 cores, the performance of HashMVT achieves 26% improvement compared with VigorNAT.

E. Real Traffic Performance

We then test the performance of HashMVT under real traffic, as shown in Fig. 8b. For this test, we use a clip of



(a) Throughput using different numbers of NAT instances.

(b) Throughput fluctuation over time under the Caida dataset.

Fig. 8: Throughput performance of HashMVT.

Caida traffic [33] containing 6750 5-tuple flows as the input traffic. However, we encounter two challenges while replaying Caida traffic. The first challenge is that Caida traffic traces only contain raw IP packets unsupported by TRex [34]. To this end, we extract 5-tuple information from each packet and regenerate flow streams with TRex. The second challenge is that sending packets at the rate of the packet timestamp could not achieve the 100 Gbps line rate speed. To simulate the fluctuations of the input traffic, we randomly vary the sending rate from 80% to 100% of the line-rate speed. With this approach, HashMVT achieves a stable throughput of about 90 Mpps. This number is lower than the experiment using arbitrarily synthesized packets, and we find that the performance of 19 instances is the same as using 24 instances. The reason is that memory accesses became DRAM bound as the L1/L2 high-performance per-core caches were exhausted (see [26] for more details).

VII. Discussion

This section discusses the generality and limitations of HashMVT. We begin by demonstrating how HashMVT can enhance the scalability of NSes beyond NAT. We then discuss some scenarios where HashMVT is not efficient.

A. HashMVT's Generality

We surveyed various real-world NSes such as load balancers [35], proxy caches [36], [37], NAT [38], [39], packet encryptions [40], intrusion detection systems (IDS) [41]–[43], deep packet inspection (DPI) [44], WAN optimizers [45], firewalls [38], and network monitors [46]. We analyzed their states and classified the states into five categories: *mapping tables*, *policies*, *counters*, *automata*, and *resource pools*.

Automata are commonly used for state machine-oriented operations such as TCP reassembly, intrusion detection systems (IDS), and deep packet inspection (DPI). These automata are usually related to a specific protocol's "flow" and are thus typically *per-protocol-flow* states that can be easily integrated into the HashMVT. The matching fields for automata are packet fields used by the protocol, and the state value represents the automaton's current state.

Counters and **policies**, on the other hand, could be more complicated because they may relate to uneven granularities of traffic, such as one counter relating to packets of 10.0.1.1/24 and another relating to 10.0.0.1/26. When an instance becomes overloaded, scaling out is conducted, and the "`shard`" operation is used to partition the HashMVT to distribute traffic and loads evenly across instances. For read-only states (*e.g.*, policies in SNORT [41] and PRADS [42]), states may have overlapping matching ranges. We conduct the overlapping matching range decoupling algorithm as detailed in [16]. After that, state entries become independent and will not be shared across instances when scaling out.

The **mapping table** and **resource pool** states are discussed in Sec. III-D.

B. Limitations

Although the HashMVT is highly effective in improving the scalability of most NSes, it is not suitable for use in scenarios where NSes have global states that are frequently updated and read. For example, consider a stateful firewall (SFW) that blocks communication when the total number of received packets exceeds a threshold within a specific time window. This SFW needs to maintain a global counter that records the number of received packets, and the state is read frequently on a per-packet basis to determine whether to throttle the traffic. This global counter must be shared across instances in such a scenario, incurring substantial performance penalty. Fortunately, such scenarios are rare in practice. For example, network operators may not necessarily demand a precise number but rather than range. Hence, a looser consistency may be employed.

VIII. Conclusion

This paper introduces HashMVT, a novel state management abstraction designed to enable scalable NSes. HashMVT improves the scalability of NSes by eliminating shared-state contention through its shared-nothing architecture, providing rich built-in state operations to simplify NS development. It further reduces performance overhead via a carefully designed hash-based matching mechanism that achieves $O(1)$ time complexity. To demonstrate its benefits, we implement

HASHMVT by reconstructing a NAT service, where the HASHMVT unifies the NAT table and resource pool into a single table, assigning each resource unit a dedicated matching field to eliminate cross-instance resource sharing. Experimental results show that HASHMVT scales linearly with the number of instances and delivers over 60% higher throughput compared to the VigNAT implementation. To validate generality, we analyze multiple real-world NSes and demonstrate how their state management can be retrofitted to HASHMVT with minimal effort.

ACKNOWLEDGEMENT

This work is supported in part by National Natural Science Foundation of China under grant 62402049, R&D Program of Beijing Municipal Education Commission under grant KM202311232005, Open Foundation of State key Laboratory of Networking and Switching Technology (Beijing University of Posts and Telecommunications) under grant SKLNST-2023-1-01, National Key R&D Program of China under grant 2022YFC3320903. We acknowledge the funding from Advanced Innovation Center for Future Blockchain and Privacy Computing and Beijing Laboratory of National Economic Security Early-warning Engineering. We thank the anonymous reviewers for their insightful comments, which have significantly improved the quality of the paper.

REFERENCES

[1] L. Yan, Y. Pan, D. Zhou, G. Candea, and S. Kashyap, "Transparent multicore scaling of single-threaded network functions," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 1142–1159.

[2] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM Conference*, 2021, pp. 65–77.

[3] J. Guo, B. Chen, K. Yang, T. Yang, Z. Liu, Q. Yin, S. Wang, Y. Wu, X. Wang, B. Cui *et al.*, "HourGlassSketch: An Efficient and Scalable Framework for Graph Stream Summarization," in *IEEE International Conference on Data Engineering (ICDE)*, 2025, pp. 1–14.

[4] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *NSDI*, Boston, MA, 2019, pp. 501–516.

[5] L. Zeno, D. R. K. Ports, J. Nelson, D. Kim, S. Landau-Feibish, I. Keidar, A. Rinberg, A. Rashelbach, I. De-Paula, and M. Silberstein, "SwiSh: Distributed shared state abstractions for programmable switches," in *Proceedings of NSDI 2022*. Renton, WA: USENIX Association, Apr. 2022, pp. 171–191.

[6] L. Liu, H. Xu, Z. Niu, J. Li, W. Zhang, P. Wang, J. Li, J. C. Xue, and C. Wang, "Scaleflux: Efficient stateful scaling in nfv," *IEEE TPDS*, 2022.

[7] M. Pozza, A. Rao, D. F. Lugones, and S. Tarkoma, "Flexstate: Flexible state management of network functions," *IEEE Access*, vol. 9, pp. 46 837–46 850, 2021.

[8] Z. Wu, T. Cui, A. Narayanan, Y. Zhang, K. Lu, A. Zhai, and Z.-L. Zhang, "Granularnf: Granular decomposition of stateful nfv at 100 gbps line speed and beyond," *ACM SIGMETRICS Performance Evaluation Review*, vol. 50, no. 2, pp. 46–51, 2022.

[9] H. Ghasemirahni, A. Farshin, M. Scazzariello, M. Chiesa, and D. Kostić, "Deploying stateful network functions efficiently using large language models," in *Proceedings of the 4th Workshop on Machine Learning and Systems*, 2024, pp. 28–38.

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[11] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić, "Rss++: Load and state-aware receive side scaling," in *Proceedings of the 15th international conference on emerging networking experiments and technologies*, 2019, pp. 318–333.

[12] P. Naik, A. Kanase, T. Patel, and M. Vutukuru, "Libvnf: Building virtual network functions made easy," in *SoCC*, 2018, pp. 212–224.

[13] A. Singhvi, J. Khalid, A. Akella, and S. Banerjee, "Snf: Serverless network functions," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 296–310.

[14] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. Ramakrishnan, "Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 780–794.

[15] P. Zheng, W. Feng, A. Narayanan, and Z.-L. Zhang, "Nfv performance profiling on multi-core servers," in *Proceedings of 2020 IFIP Networking Conference (Networking)*. IEEE, 2020, pp. 91–99.

[16] Z. Wu, Y. Zhang, W. Feng, and Z.-L. Zhang, "Nflow and mvt abstractions for nfv scaling," in *Proceedings of IEEE INFOCOM 2022*. IEEE, 2022, pp. 180–189.

[17] H. Li, Q. Chen, Y. Zhang, T. Yang, and B. Cui, "Stingy sketch: a sketch framework for accurate and fast frequency estimation," *Proceedings of the VLDB Endowment*, vol. 15, no. 7, pp. 1426–1438, 2022.

[18] Intel. (2007) Intel SSE4 Programming Reference. Accessed: 2025-3-8. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/d9156103-705230.pdf

[19] ARM. (2017) Technical Reference Manual - ARM architecture family. Accessed: 2025-3-8. [Online]. Available: https://documentation-service.arm.com/static/5e7e1430b471823cb9de57cf

[20] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *NSDI*, 2018, pp. 299–312.

[21] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified nat," in *SIGCOMM*, 2017, pp. 141–154.

[22] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling innovation in network function control," in *ACM SIGCOMM CCR*, vol. 44, no. 4. ACM, 2014, pp. 163–174.

[23] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A framework for nfv applications," in *SOSP*, 2015, pp. 121–136.

[24] S. Luo, H. Yu, and L. Li, "Practical flow table aggregation in sdn," *Computer Networks*, vol. 92, pp. 72 – 88, 2015.

[25] J. Xing, A. Chen, and T. E. Ng, "Secure state migration in the data plane," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020, pp. 28–34.

[26] P. Zheng, W. Feng, A. Narayanan, and Z. Zhang, "Nfv performance profiling on multi-core servers," in *IFIP Networking*, 2020.

[27] Intel. (2025) Dpdk boosts packet processing, performance, and throughput.

[28] DPDK Project, "Generic flow API (rte_flow)," *https://doc.dpdk.org/guides/prog_guide/rte_flow.html. Accessed: 2025-1-11*, 2025.

[29] P. Srisurensh and K. Egevang, "Network address translator rfc 3022," *IETF, January*, 2001.

[30] The TRex Comunity, "TRex," 2025, https://trex-tgn.cisco.com. Accessed: 2025-3-10.

[31] R. NCC. (2025) RIPE Database. Accessed: 2025-3-10. [Online]. Available: https://www.ripe.net/manage-ips-and-asns/db/

[32] Y. Sun and M. S. Kim, "A pipelined crc calculation using lookup tables," in *2010 7th IEEE Consumer Communications and Networking Conference*. IEEE, 2010, pp. 1–2.

[33] Center for Applied Internet Data Analysis, "Caida: center for applied internet data analysis," *https://www.caida.org/home/. Accessed: 2025-1-10*, 2025.

[34] Cisco Systems, Inc., "TRex Stateless API Reference," *https://trex-tgn.cisco.com/trex/doc/cp_stl_docs/api/client_code.html. Accessed: 2025-03-10*, 2025.

[35] Inlab Networks, "Overview :: Balance by Inlab Networks," *https://www.inlab.net/balance/*, 2022.

[36] W. Tarreau *et al.*, "Haproxy-the reliable, high-performance tcp/http load balancer," *https://www.haproxy.org. Accessed: 2025-1-11*, 2025.

[37] footer.inc, "squid : Optimising web delivery," *https://www.squid-cache.org/Accessed: 2023-03-02*, 2023.

[38] Linux Community, "Netfilter - Firewalling, NAT, and Packet Mangling for Linux," *https://www.netfilter.org. Accessed: 2025-3-11*, 2025.

[39] Mazu Networks, Inc., "Mazu NAT," *https://github.com/kohler/click/blob/master/conf/mazu-nat.click. Accessed: 2025-3-10*, 2025.

[40] OpenVPN Inc., "VPN Software Solutions & Services For Business — OpenVPN," *https://openvpn.net. Accessed: 2025-3-10*, 2025.

[41] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[42] gamelinux, "PRADS - Passive Real-time Asset Detection System," *http://gamelinux.github.io/prads/. Accessed: 2025-1-10*, 2025.

[43] L. De Carli, R. Sommer, and S. Jha. (2014) Beyond pattern matching: A concurrency model for stateful deep packet inspection. ACM.

[44] X. Yu, W.-c. Feng, D. Yao, and M. Becchi, "O^{3}fa: A scalable finite automata-based pattern-matching engine for out-of-order deep packet inspection," in *ANCS*. IEEE/ACM, 2016, pp. 1–11.

[45] B. Agarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "Endre: An end-system redundancy elimination service for enterprises." in *NSDI*, 2010, pp. 419–432.

[46] W. Feng, C. Liu, and J. Chen, "Batchsketch: a "network-server" aligned solution for efficient mobile edge network sketching," in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, 2022, pp. 811–813.