A Scalable and Dynamic ACL System for In-Network Defense

Changhun Jung[†]
Ewha Womans University
mizno@ewha.ac.kr

Sian Kim[†]
Ewha Womans University
ksy60a@ewha.ac.kr

Rhongho Jang
Wayne State University
r.jang@wayne.edu

David Mohaisen
University of Central Florida
David.Mohaisen@ucf.edu

DaeHun Nyang
Ewha Womans University
nyang@ewha.ac.kr

ABSTRACT

In-network/in-switch Access Control List (ACL) is an essential security component of modern networks. In high-speed networks, ACL rules are often placed in a switch's Ternary Content-Addressable Memory (TCAM) for timely ACL match-action and management (e.g., insertion and deletion). However, TCAM-based ACL systems are encountering an scalability issue owing to increasing demand on AI-powered autonomous defenses that detect and block attacks online, which inevitably derives finer-grained ACL rules. Existing solutions minimize the TCAM usage by partially offloading ACL matching into larger Static Random-Access Memory (SRAM) or customized hardware. Nevertheless, current SRAM-based solutions induce high management costs, especially a high rule-deployment latency, which delays time-sensitive defense actions. Also, the customized hardware approaches have its own scalability issue. To support autonomous defenses at a scale, in this paper, we propose an in-switch ACL system called PortCatcher, which breaks the trade-off between scalability and rule management latency. System-wise, we detach layer-4 port matching from TCAM for improving its memory efficiency. Algorithm-wise, we introduce a novel port (range) rule representation concept, called linear range map (LRM), which enables port (range) matching in SRAM-based hash tables. LRM guarantees not only fast and scalable port matching but also low-latency ACL management for timely defenses. With static ACL datasets, we show that PortCatcher saves 74%∼90% TCAM space compared to state-of-the-art approaches by adding a small overhead to SRAM (0.49 SRAM entry per ACL rule). Also, we deploy PortCatcher on a programmable switch to demonstrate that PortCatcher can serve the 5-tuple rule matching at a line rate, where port rules are completely matched in SRAM. With an attack traffic-driven dynamic ACL dataset, our use case study shows that PortCatcher's rule deployment is 168x faster than the state-of-the-art approach, allowing our in-network defense system to block 92.09% (55.45% more) malicious packets and all flows in an attack trace.

[†]These two authors contributed equally.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560606

CCS CONCEPTS

• **Networks** → **Bridges and switches**; • **Security and privacy** → **Access control**.

KEYWORDS

In-network ACL, dynamic management, low-latency defense, scalable port (range) matching

ACM Reference Format:
Changhun Jung[†], Sian Kim[†], Rhongho Jang, David Mohaisen, and DaeHun Nyang. 2022. A Scalable and Dynamic ACL System for In-Network Defense. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3548606.3560606

1 INTRODUCTION

Access Control List (ACL) is an essential security component of modern networks, and plays an important role in blocking unwanted and malicious network traffic. To date, in-network/switch ACL is wildly used in various network contexts, such as backbone [82], core [73], data center [66, 78], edge [33], and enterprise [51] networks, for data plane defenses and access control alike.

In general, ACL rules are created by network administrators based on network policies for basic and static network control purposes (i.e., static ACL), including network flow control, quality of service (QoS), unused port blocking, where the size of ACLs can be 100K entries or even larger [21]. Traditional ACL systems match the source and destination IP addresses to perform a layer-3 network control. However, due to the need for finer-grained layer-4 network control and defenses [40, 55, 76, 80], the support of layer-4 ACL (L4-ACL) becomes essential. Nevertheless, the L4-ACL's additional matching of the protocol and source-destination ports fields inevitably increases the memory overhead. Even worse, a special property of the port rule, namely range matching, triggers a memory inefficiency issue to meet the fast matching requirements, which aggravates the problem of scalability.

Meanwhile, there has been a surge in demand for online/in-network attack detection and mitigation, i.e., autonomous defense [14, 50, 87], thanks to advances in deep and machine learning technologies [11, 28, 57, 65, 85] and the programmability of modern network switches [58]. Such a trend, in turn, requires in-switch/in-network ACL systems that are capable of rapidly and scalably deploying dynamically-generated ACL rules (i.e., dynamic ACL). Recent advances in in-network autonomous defense systems [14, 50, 80, 87] enabled onsite attack defenses by allowing flow-level investigation.

Therefore, unlike static ACL rules with diverse matching types (i.e., exact, range, wildcard), dynamic ACL mainly performs exact per-flow matching. Considering that the L4-ACL has a large flow ID (i.e., 13-Byte ID per layer-4 flow) and the massive number of machine-generated exact matching rules under a large-scale attack, *scalability* is also a crucial factor in dynamic ACL.

We stress that the scalability issue of the legacy static ACL is as urgent as the dynamic ACL's for in-network autonomous defense systems for the following reasons. First, the static ACL, as a preliminarily filter, can prevent out-of-interest network traffic from unnecessarily triggering the attack defense loop (i.e., detection and action), which helps reduce the burdens of the entire system. Second, the static and dynamic ACLs that coexist in a switch's data plane have to share and compete for fixed and scarce memory resources. Nevertheless, with the ever-increasing network traffic volume, ACL's scalability is often sacrificed to guarantee high-speed and non-delayed packet processing. In particular, ACLs today utilize advanced and high-speed memory, such as Ternary Content-Addressable Memory (TCAM), which supports a hundred million longest prefix matching (LPM) per second by exploiting hardware-backed parallel processing. Although TCAM meets the functional needs of L4-ACLs, namely port range matching of static ACL and exact matching of dynamic ACL, it is extremely constrained in recent switches (i.e., up to 8.2 MB [13]) and is preferentially assigned to more essential network functions, such as routing. To date, several works have been proposed to improve TCAM's memory efficiency [16, 17, 41, 44, 47, 52, 53, 61, 62, 68, 70, 71, 90]. Unfortunately, their memory-saving effects are limited to scaling out neither the static nor the dynamic ACL (see section 2.2 for details).

A better approach to resolve the scalability issue stemming from *(1) the port range rules of the static ACL* and *(2) volumetric exact matching rules of the dynamic ACL* is to offload partial ACL matching from TCAM to SRAM [19, 25, 74, 75, 84] assisted by an external rule separation function (i.e., external encoding). However, we observe that the existing approaches induce high rule deployment latency due to the complex external functions, which is unacceptable for an in-network autonomous defense system to defend against attack flows in real-time. As such, we argue that *a complete and desirable in-network ACL system must: (1) support both scalable static and dynamic ACLs, and (2) guarantee low-latency rule deployment for dynamic ACL rules, to allow viable in-network defenses.*

In this paper, we design, implement, and evaluate a new L4-ACL architecture that breaks the trade-off between the scalability and latency of ACL systems. Our ACL system, called PortCatcher, separates layer-4 port matching from the IPs and protocol matching, and performs a two-stage matching for efficiency. In the first stage, IPs and protocol matching is performed in TCAM. This separation allows us to save a large amount of TCAM by eliminating the duplicate IP and protocol definitions having different port range rules. In the second stage, which is a key contribution of this paper, the port (range) matching is executed in SRAM using a novel rule matching primitive, called the Linear Range Map (LRM). LRM is a port rule representation where the rule-transformed entries are stored efficiently and matched quickly in SRAM-based simple hash table. At the core of LRM, port rules, regardless of the matching types (i.e., range matching in static ACL and exact matching in dynamic ACL), are all projected into bitmaps, where one bit stands for a 16-bit layer-4 port (i.e., for scalability). Then, multiple optimization functions are designed to further improve memory efficiency, particularly by handling duplicated, long-range, and wildcard port rules. Therefore, using LRM, PortCatcher supports scalable port matching in both static and dynamic ACL management scenarios. Especially, for autonomous defense systems that dynamically generate per-flow ACL rules [14, 50, 80, 87], PortCatcher uses only 72-bit TCAM and 102-bit SRAM memory per flow in the worst case, which means the programmable switch used in this work [12] can host millions of ACL rules with $O(1)$ MB TCAM and $O(10)$ SRAM memory. It is worth mentioning that the latest switch processor has an increased memory space (i.e., 8.2 MB TCAM and 160 MB SRAM [13]). However, the TCAM is still not large enough to scale out ACL management, but PortCatcher can take advantage of the larger SRAM to scale out the port matching. Moreover, with straightforward-yet-efficient ACL rule processing functions in the control plane, PortCatcher supports low-latency ACL deployment that meets the needs of autonomous defenses. While the TCAM-SRAM hybrid concept is not new, PortCatcher is the first in-switch ACL system that supports scalable and low-latency ACL matching and management in SRAM without a specialized hardware.

Contributions. Our key contributions in this paper are as follows: (1) We design PortCatcher, a novel L4-ACL system to break the trade-off between the scalability and latency of ACLs without requiring any specialized hardware. (2) We introduce a novel port range rule representation, LRM, to enable scalable and low-latency ACL match-action and management with a simple SRAM-based hash table. (3) We optimize PortCatcher with four design ideas to minimize PortCatcher's memory consumption without sacrificing ACL management latency. (4) We deploy PortCatcher on a programmable switch to verify PortCatcher's real-time matching performance. Our evaluation results based on static ACL datasets show that PortCatcher saves 74%-90% of TCAM compared to ACL_{TCAM} and 36%-61% compare to ALPM. (5) With an attack traffic-driven dynamic ACL dataset, our use case study shows that PortCatcher's rule deployment is 168x faster than a state-of-the-art approach, allowing our in-network defense system to block 92.09% (55.45% more) malicious packets and all flows in an attack trace.

2 BACKGROUND AND MOTIVATION

The reported global distributed denial-of-service (DDoS) attacks have rapidly increased in volume over the past decade [1, 20]. The targets of DDoS attacks are not limited to endpoints, but also single and multiple networks' transit-links for indirectly making endpoint servers unreachable [37, 67]. In practice, specialized hardware components, e.g., hardware firewalls, are used to address such security issues. However, firewalls are expensive. Thus, they are placed at the border of an autonomous system (AS), with limited power in defeating attacks that target internal AS links [37, 67]. To prevent such threats, any single node of the network should be able to perform the defense once malicious flows are detected, thus an in-network or in-switch ACL has become a crucial component that is available in most network devices. Over the last few decades, network monitoring and attack detection functions have been greatly improved through advanced network flow measurement data structures [23, 29, 30, 32, 34, 42, 43, 46, 48, 49, 59, 69, 77, 83] and machine

learning techniques [11, 28, 57, 64, 65, 85, 86]. Moreover, the centralized view of software-defined network (SDN) enabled global detection of attack flows and action against them. With these efforts, we are inching closer to autonomous in-switch defenses, which will automatically detect and block malicious flows based on pre-defined rule sets or using learning modules within a network switch.

FlowLens [14] is recently proposed to measure per-flow packet size distribution in a switch's data plane and detect botnet traffic and covert channels using machine learning in the switch's control plane. Similarly, Jaqen [50] and Poseidon [87] proposed to detect attack flows in the data plane with a compact data plane data structure. Ripple [80] focused on defending against link-flooding attacks using a set of decentralized switches across the network. These systems concern a control delay of centralized frameworks and suggest a new paradigm, namely flow-level detection in the data plane. Such an approach must be supported by a scalable and low-latency ACL system, for defending against large-scale attacks without delay. However, as the key component that is responsible for performing eventual defense actions (e.g., packet drop), the current in-switch ACLs hardly meet the demands of modern networks because of two drawbacks: scalability and rule management latency.

2.1 Challenges of in-network ACL

Memory constraint of network devices (*scalability*). Core network switches must contain high-speed memory (e.g., TCAM) to execute network functions (i.e., match-action) under a large volume of packets. However, TCAM is a scarce resource in network switches with a few megabytes storage capacity [13], restricted by its large physical size, high power consumption, and high price tag. Therefore, as shown in Figure 1 ①, the network functions, especially in-switch ACLs, face a serious scalability issue due to the memory resource constraint in the data plane. First, a network policy-based static ACL is essential for routine network control and defense. Since an IP-based coarse network control can no longer meet the network operators' needs, given the coexistence of virtual machines and various applications behind an IP, the L4-ACL has been widely adopted in practice, although it requires more memory since additional fields, i.e., protocol and source-destination ports, should be matched. Second, recent in-network autonomous defense systems rely on either machine learning [14] or a predefined policy [50, 80, 87] to generate ACL rules dynamically for blocking malicious flows, which will add extra burdens to resource-constrained switches, as shown in Figure 1. The fundamental difference between static and dynamic ACL rules is that the static ACL rules involve various matching types, such as exact, range, and wildcard matching [72], whereas the dynamic ACL has exact matching rules only, as most in-network autonomous defense systems perform flow-level detection and action [14, 50, 80, 87]. We stress that both static and dynamic ACLs are essential building blocks of in-network defense systems. Due to the resource constraints, a complete and desirable in-network defense system must support memory-efficient static and dynamic ACLs simultaneously.

High management cost (*latency*). Static ACLs are usually managed offline, which means the ACL rules have a relatively loose time constraint for rule population. However, an autonomous defense system requires a tight ACL deployment deadline to block attack traffic at low latency, as shown in Figure 1②. Unfortunately, the existing solutions address the scalability by designing a complex rule encoding function, which results in a high management cost when deploying newly generated rules (e.g., [17, 70, 75]). It is worth noting that the ACL management overhead in this case may be negligible for a single tuple-based network application, such as destination IP-based routing. However, the complexity is significant when handling 5-tuple-based ACL rules (§5). To this end, we conclude that the current ACL systems have become a bottleneck in autonomous defense systems due to the rule deployment latency.

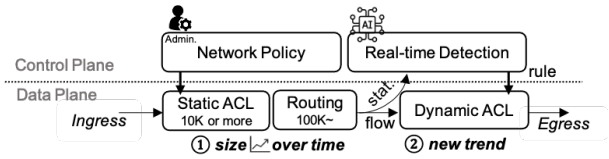


Figure 1: The in-switch ACL's scalability issue caused by a memory resource competition in the data plane.

2.2 Trade-off: Scalability vs. Latency

The existing L4-ACL systems focused only on improving the static ACL's scalability [3, 9, 17, 70, 75], ignoring the emerging needs of the dynamic ACL for a low-latency attack blocking. However, as we discussed previously, the scalability of the static ACL is as crucial as the dynamic ACL since it reduces the burden of the autonomous defense system. Although these solutions do not directly benefit the dynamic ACL with exact matching rules, their designs, namely port matching isolation [3, 9] and SRAM offloading [75] inspired our system framework design, as described in the followings.

L4-ACL can be expressed as $L = \{R_1, R_2, ...R_n\}, R_i = (M_i, A_i)$, where M_i is a matching field that consists of source-destination IPs, source-destination ports, and the protocol of a flow (5-tuple), and A_i is the corresponding action to be taken, when matched.

❶ **TCAM-only L4-ACL (*baseline*).** It is commonly accepted that TCAM is essential for in-network functions (i.e., match-action functions), including ACL, as it supports hundreds of millions of parallel LPM. *While TCAM fits perfectly for IP matching, it exposes a critical memory shortcoming in 5-tuple matching (i.e., ACL_{TCAM}).* An IP definition belongs to a prefix rule that works with an IP address and mask to match a particular IP address ($IP_i/32 = IP_i$) or a range of IP addresses ($IP_i/24 \in [IP_j, IP_k]$). Either case can be expressed and stored as a single TCAM entry and be parallelly searched among other entries. A combination of source IP, destination IP, and protocol can also be expressed by a single TCAM entry, thanks to the ternary matching feature. Unlike the IP addresses, however, the port rule may define *an arbitrary range*, which should be expressed with one or more prefix rules (or TCAM entries), triggering the well-known range-to-prefix expansion issue (i.e., memory inefficiency). For instance, to perform port range matching, a 16-bit range port rule, e.g., $range < 1024$, can be expressed with one prefix $000000*$. However, $range > 1023$ should be expanded with six prefixes: $000001*$, $00001*$, $0001*$, $001*$, $01*$, $1*$. As a result, a single range rule could generate multiple TCAM entries with duplicated IPs and protocol definitions, which significantly reduces the TCAM's storage efficiency section 5.2). A continuous range should be expressed

with at most $2w-2$ entries with an internal Binary-Prefix approach, where w is the range bit length [70]. A $\text{Port}_{\texttt{src}}$ ($w = 16$) range rule can be expanded up to 30 TCAM entries. Even worse, a combination of $\text{Port}_{\texttt{src}}$ and $\text{Port}_{\texttt{dst}}$ may result in a single range matching rule to be expanded up to 900 TCAM entries. Even though the upper bound can be lowered to $2w-4$ with a complex Gray code-based encoding scheme [17], the increased encoding complexity makes dynamic management of ACL rules more challenging, especially when ACL rules must be deployed instantly to block malicious flows (i.e., rule management latency).

❷ **TCAM-LOU-based L4-ACL (*port isolation*).** The logical operation unit (LOU) is specialized hardware for port range rules. Each LOU stores only start and end ports of a range rule, and is associated with an IP and protocol definition for 5-tuple matching. Lastly, the LOU checks if a port falls into the range using operator-operand pairs (e.g., > 128 and < 1024). By isolating the port matching from TCAM (IP and protocol matching), ACL saves TCAM and avoids the range-to-prefix expansion issue. Unfortunately, there are only a limited number of LOUs in commercial switches, which is a major drawback of this approach. For instance, modern switches have only 24∼104 LOUs [2, 3, 8, 9], allowing them to support up to 104 different port range rules. Although ACL_{LOU}'s operation is straightforward and fast, it limits the diversity of port rules of switch ACL. *Despite of its scalability issue, ACL_{LOU} clearly shows that the port matching isolation is a promising direction for saving TCAM memory.*

Instead of LOU, one may suggest that range matching can be easily performed by setting up a hash table in SRAM, where IP and protocol are a key and the corresponding range list is a value. However, we note that an IP and protocol rule may be associated with an indefinite number of port (range) rules, which should be matched sequentially. Unfortunately, this design is infeasible for pipeline-based switches due to limited programmability (i.e., prohibition of loop operations and memory double access [4, 87]).

❸ **TCAM-SRAM-based L4-ACL (*SRAM offloading*).** *An alternative for saving TCAM is to offload a portion of matching tasks to SRAM.* ALPM [75], a state-of-the-art approach, divides a matching field (i.e., M_i) into two bit-wise portions, and stores the most significant bits (`msb`) in TCAM and less significant bits (`lsb`) in SRAM. In the encoding phase, ALPM first constructs a binary trie (i.e., prefix tree) using all entries, then trims and stores its subtrees, where the total number of nodes is smaller than a threshold (s), in pre-allocated and equal-sized (s) SRAM partitions. Finally, the root node (e.g., $101*$) of each subtree among the SRAM partition indices is stored in TCAM. We note that ALPM is originally designed for a single field LPM (i.e., IP), but one can extend the same idea to match multiple fields (e.g., 5-tuple). However, this extension faces three issues. First, ALPM still requires port range-to-prefix expansion before constructing the trie (i.e., encoding). Second, since SRAM does not support ternary matching, the offloaded portion (i.e., $\text{5-tuple}_{\texttt{lsb}}$) will produce a massive amount of duplicated leaf nodes (SRAM entries). Assuming that the offloaded portions are source and destination ports, a port rule (src=ANY, dst=17) will produce 2^{16} SRAM entries to match all port combinations (i.e., SRAM memory efficiency). One can further offload fewer bits to the SRAM, but the TCAM's saving effect diminishes. Last but not least, a simple ACL management operation (e.g., rule insertion and deletion) triggers complex operations, including trie re-construction, TCAM update, and multiple SRAM partition updates, which delay a time-sensitive defense action to be taken in the data plane (section 5.3).

Table 1: Scalability vs. Latency

		ACL_{TCAM}	ACL_{LOU}	ALPM	**PortCatcher**
TCAM		5-tuple	IP, Proto	$\text{5-tuple}_{\texttt{msb}}$	IP, Proto
SRAM		-	-	$\text{5-tuple}_{\texttt{lsb}}$	Port
LOU [9]		-	Port	-	-
Data Plane	Scalability	Poor	Poor	Poor	Good
	Latency	Line rate	Line rate	Line rate	Line rate
Control Plane	Scalability	Low	Low	Very High	Low
	Latency	High	Low	Very High	Low

2.3 Our Approach

Inspired by ACL_{LOU} and ALPM, our L4-ACL system, called PortCatcher, is designed to isolate the port matching from 5-tuple matching and completely offload it in SRAM, as shown in Table 1. Our design choice not only avoids TCAM's inefficiency (i.e., range-to-prefix) but also saves TCAM memory by excluding port definitions (i.e., saves 32-bit per TCAM entry). Uniquely, we propose a novel Linear Range Map (LRM) concept to enable hash table-based scalable and non-delayed port match-action without specialized hardware. Essentially, LRM is a bitmap representation of port rules designed for expressing both exact and range rules in a memory-efficient way. Thanks to the design, PortCatcher supports scalable static and dynamic ACL management simultaneously. Moreover, the low management cost of LPM allows PortCatcher to guarantee low latency in the dynamic ACL rule deployment to block malicious flow immediately (section 5.4).

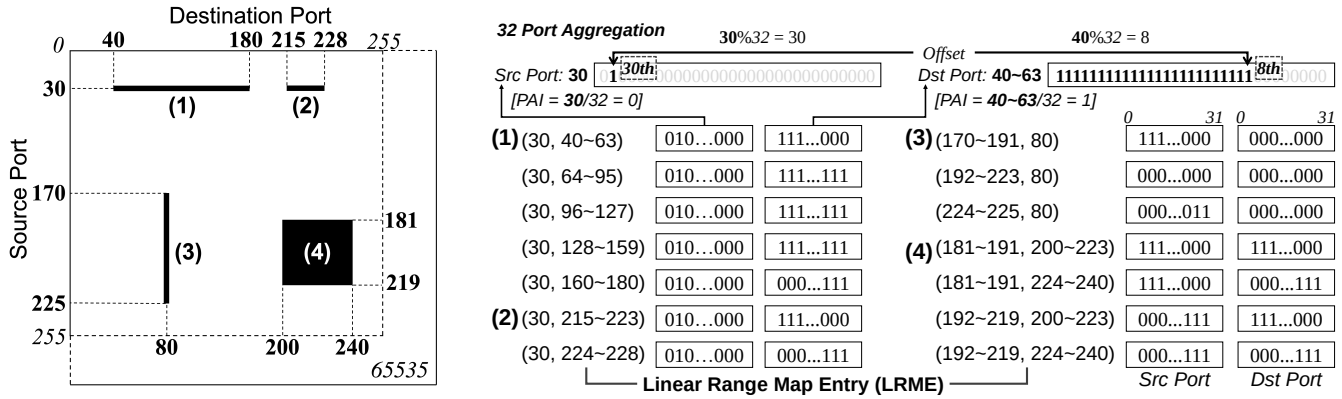
3 PORT RANGE MATCHING IN HASH TABLE

Before delving into our design, we describe LRM, which enables a hash table-based port range matching in SRAM.

3.1 Bitmap Representation of Port Ranges

A layer-4 port rule is given as a source-destination port range pair. For example, assume a port range rule consisting of four port rule pairs: (30, 40-180), (30, 215-228), (170-225, 80), and (181-219, 200-240). For a hash table-based range matching, we break down these port (range) rules into 1,810 (=141 + 14 + 56 + 39 × 41) individual source-destination pairs, i.e., all the points in the black area in Figure 2(a) but not the range, and then store each pair as a 32-bit table entry (16 bits per source and destination ports). By doing so, a packet with layer-4 port fields can be matched using the hash table (i.e., hash table-based exact port matching). However, this approach suffers from scalability and latency issues. As discussed earlier, port rules with port range settings can be extended to a massive number of 32-bit table entries, which degrades the memory efficiency significantly. In addition, the port rule management (e.g., insertion and deletion) is extremely costly since the extended port entries are stored in the hash table separately and independently, which causes a high latency in the dynamic control scenario (i.e., autonomous defense).

A naïve approach to implementing our idea is to express the port rules in a two-dimension bitmap for port matching. However, the bitmap approach is still costly (i.e., 2^{32} bits). Moreover, the port rules are not matched independently, but with the IP and protocol rules



(a) Port range rules' logical map (src, dst): **(1)** (30, 40-180), **(2)** (30, 215-228), **(3)** (170-225, 80), **(4)** (181-219, 200-240).

(b) Linear Range Map (LRM) with 32 port aggregation for the logical map in (a). Each entry has source-destination port bit map representing exact port ranges. All the LRMEs will be stored in the hash table.

Figure 2: Linear Range Map (LRM) with illustration.

(i.e., 5-tuple matching). As such, the bitmap should be created for r distinct IP and protocol rule pairs (i.e., $r \times 2^{32}$), which is unacceptably large. Our observation is that the bitmap wastes memory because it reserves the space for undefined ports. Therefore, we propose the LRM Table to store only the meaningful portion of the bitmap (i.e., black area in Figure 2(a)) using a simple hash table. To do so, we suggest *LRM port aggregation* to represent port range rules in a compact bitmap format. We further propose three optimization designs, namely *range reversing*, *ANY port handling*, and *Universal LRM table with deduplication* to scale up the table capacity.

3.2 LRM: Range Representation for Hash Table

To enable the LRM-based port matching with a hash table, we introduce a compact representation of port rules, called *LRM with port aggregation*, which internally aggregates the continuous ports into a source-destination port bitmap, called LRM entries (LRME). As shown in Figure 2(b), we break down the port rule (30, 40-180) into five continuous subsets: (30, 40-63), (30, 64-95), (30, 96-127), (30, 128-159), and (30, 160-180), where each subset includes up to 32 continuous ports for both source and destination, in what we call 32-port aggregation. Then, we express each subset using two 32-bit bitmaps, where each bit stands for a single individual port value, as shown in Figure 2(b). By repeating this process for every port rule, we eventually obtain 28 32-bit bitmaps, which saves us 98.5% of the memory compared to the naïve hash table-based approach (i.e., 1,810 32-bit entries). Eventually, we store these bitmaps (LRME) into a hash table for port matching purposes.

An LRME can be promptly located in a hash table by its starting port number, and a specific port number of interest can be located within an LRME by the offset. Thus, LRME is in substance a position code of port ranges. To locate LRMEs in the hash table, we use the first source and destination ports of LRMEs divided by the port aggregation size (e.g., 32) to calculate a port aggregation index (PAI). For instance, the 32-PAI of a port pair (30, 40-63) is <0, 1>, as shown in Figure 2(b). Therefore, every aggregated LRME will have a unique PAI to be used as the item key in the hash table. Moreover, the port matching can be done by calculating the offset (see Figure 5 for the runtime port matching logic).

❶ **Optimization 1 port range reversing.** Even with the port aggregation, some port range rules generate a large number of LRMEs. For instance, the port range $\{\geq 1024\}$ generates 2,016 LRMEs, even with a 32 port aggregation. To tackle this issue, we can reverse the range definition for a port rule with a large coverage (i.e., range covers more than $2^{16}/2$ ports) with a reversed action (e.g., drop to permit). Therefore, we can reduce the number to as low as 32 LRMEs by reversing the range as $\{< 1024\}$. We note that the reversed action is handled by a reverse flag in our system.

❷ **Optimization 2 ANY port handling.** The port rules can have four different "ANY" settings varying source and destination ports, namely (src=ANY,dst=ANY), (ANY,-), (-,ANY), and (-,-), where "-" refers to a concrete port or port range definition. "ANY" rules are a major challenge for LRM, since a port rule (30, ANY) with 32-port aggregation generates $2^{16}/32$ LRMEs and (ANY, ANY) creates $2^{32}/32/32$ LRMEs. While we can resolve the issue using the range reversing, the solution also wastes memory with "ANY" rules generated LRMEs. To further understand this issue, assuming that we use an invalid port number (i.e., 0) to express the "ANY" in the port rule (i.e., range reversing). As such, the port rule (30, ANY) becomes (30, 0), and the problem is that the destination port portion of its LRME is less meaningful since no packets use port "0". Therefore, we can save the memory for "ANY" rules by maintaining these rules in separated tables with half-sized LRMEs. Moreover, an (ANY, ANY) does not generate LRMEs in our system design (section 4).

❸ **Optimization 3 Universal table with port rule deduplication.** LRM can work as a port blacklist to block the traffic from or to disallowed ports. However, in a general L4-ACL, port rules are combined with IP and protocol definitions for 5-tuple-based access control (i.e., rule integrality). Therefore, every IP and protocol rule should have a unique LRM that represents a set of port rules. For memory efficiency, PortCatcher is designed to maintain a universal hash table for multiple LRMs. To distinguish LRMs in the table, we assign a unique ID, called LRM-ID, to every LRM. Eventually, the LRM-ID combined with PAI will be used as an item key to locate a unique LRME (see Figure 5). For multiple IP and protocol rule pairs with the same port rule set, a single LRM with the same LRM-ID is shared to avoid LRME duplication in the table (i.e., memory saving).

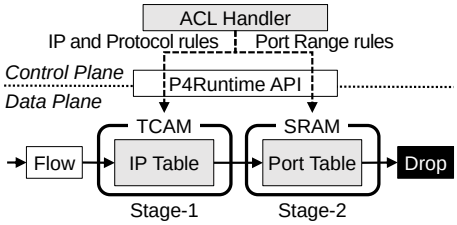


Figure 3: System Architecture. The ACL handler separates L4-ACL rules in two parts, namely IP and port, and then installs each part into the corresponding data plane modules. Subsequently, the data plane modules run independently to filter out unwanted/malicious network traffics based on installed rules (i.e., black-list ACL).

4 PORTCATCHER: SYSTEM DESIGN

In this section, we introduce our L4-ACL matching system, namely PortCatcher, and its data plane and control plane designs.

4.1 System Overview

Our L4-ACL consists of three components: IP module, port module, and ACL handler. As shown in Figure 3, the IP and port modules are implemented in the switch data plane (i.e., ASIC) to perform a two-stage matching, where the source IP, destination IP, and protocol are matched in the IP module with TCAM (stage-1), and then the source and destination ports are matched in the port module with SRAM (stage-2). The ACL handler resides in the switch control plane (i.e., general CPU board running a network OS), which is responsible for managing ACL rules based on the data plane design. The communication between the data and control planes is realized with runtime API over PCIe, which is generally supported by commercial switches.

4.2 Data Plane Modules

As shown in Figure 4, we design two match-action tables in the data plane to isolate the port matching from IP and protocol matching. Therefore, ACL rules should be stored separately but without losing the integrality. In other words, a packet must match two tables' entries that belong to an identical ACL rule. To resolve this issue, we designed a coupled matching mechanism and implemented it in the IP table's action and port table's matching logic.

(1) IP table (stage-1): The IP table resides in TCAM and the match-action logic is conducted as follows:

- **Matching field.** IP table's matching fields include source IP, destination IP, and protocol (① in Figure 4). When defining an L4-ACL rule, each IP also defines a mask to match a subnet. The protocol also requires a mask to match the exact number or *any*. We note that TCAM's mask has a "don't care" bit for generic wildcard matching.
- **Action code.** Once matched with a table entry, the corresponding action data (i.e.,① in Figure 4) is passed to the next stage using a pre-defined action code (**get_coupling_info()**) for port table matching at stage-2. For L4-ACL rules with a (src=ANY, dst=ANY) port rule, PortCatcher sets the rule action as "Drop" and skips the rest of the matching.
- **Action data.** The action data is the information required for the port table matching. As shown as ① in Figure 4 , PortCatcher stores port rules separately in three tables: `Src ANY`, `Dst ANY`, and `No ANY`. Therefore, the action data reserves three independent sub-fields accordingly. In each sub-field, LRM-ID is a unique ID to lookup a group of port rules associated with the matched TCAM entry (i.e., IP and protocol rule pair or its LRM), as shown in Figure 2. A REV flag to indicate whether the port range rules are stored in a reversed manner.

(2) User metadata (coupling stage-1 and stage-2): User metadata is a user-defined data structure for data communication within a switch's data plane. Any function in the packet processing pipeline can read and write the user metadata in its own stage. In our system, the user metadata is mainly responsible for delivering IP table's action data (i.e., coupling information) to the stage-2 port table (② in Figure 4). Our user metadata has six variables, 51 bits in total, including three 16-bit LRM-IDs, three 1-bit REV flags.

(3) Port table (stage-2): The port matching is activated only if a packet matches one of the entries in the IP table and its action is not "Drop". As shown by ③ in Figure 4, the port table consists of three independent hash tables, each of which is associated with a sub-field of IP table's action data. At stage-2, PortCatcher retrieves the action data from the user metadata and sequentially lookup `Src ANY`, `Dst ANY`, and `No ANY` tables with the corresponding LRM-ID in sub-fields. The port tables are as follows:

- **Matching field.** The port table performs exact matching with a key that consists of LRM-ID and PAI (driven from the packet's port information). LRM-ID is used for locating a unique LRM (representing a set of port rules, per Figure 2) associated with the matched IP and protocol entry and the PAI is for searching an LRME that covers the ports of the packet.
- **Action code and logic.** Once the key is found in the table, the port table will simply copy the action data (i.e., LRMEs) into the user metadata using the predefined action code (**get_LRME()**) for port matching and REV flag handling.
- **Action data (LRMEs).** The action data field stores LRMEs. As shown in Figure 4, only the `No ANY` has both source and destination LRMEs (64 bits). The other two tables store either source or destination LRMEs for the non-ANY port rule, thus each LRME requires only 32 bits memory (i.e., SRAM saving).

(4) Port matching and REV flag handling: The eventual port rule matching is performed after acquiring an LRME in port table. As shown in Figure 5, once an LRME is found in the hash table, PortCatcher checks whether the bits that represent the input values (i.e., packet's ports) are "1" or not by executing `AND` with the LRME and the bitmap of a port (pair). For example, port 183's bitmap is $1 \ll 23$, where 23 is an offset calculated by `offset` = 183 mod 32 = 23 in the case of 32 port aggregation. For the `No Any` table, the matching is independently performed for source and destination ports. However, only the both-matched case (neither is zero) is considered the eventual matching (④ in Figure 4). However, we note that the eventual matching of a port rule does not necessarily mean the packet should be dropped, since port range rules can be stored in a reversed manner. Therefore, after lookup, PortCatcher

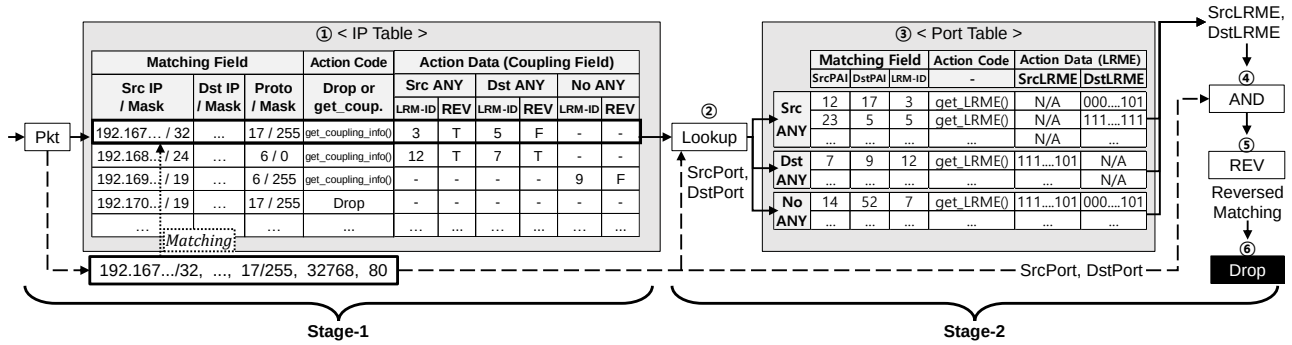


Figure 4: Workflow of PortCatcher with IP table and port table in the data plane. IP and protocol parsed from a packet header of a flow is matched with IP table in TCAM. If matched, the port range matching is performed in the port table of SRAM. Two tables implemented in match-action unit (MAU) isolate the IP and port matching.

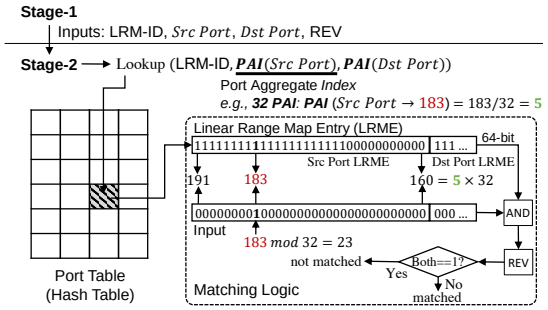


Figure 5: LRME searching and port matching logic.

check REV flag (⑤ in Figure 4) to determine whether the packet should be dropped or sent to the next stage (e.g., routing). If the port numbers are matched, REV "false" triggers the "drop" action; otherwise, PortCatcher moves to the next table. If not matched, REV "true" drops the packet (⑥ in Figure 4); otherwise, moves to the next table. The packet is permitted only if it passes all port tables. This procedure is illustrated in the dotted box of Figure 5.

4.3 Control Plane Module: ACL Handler

The ACL handler resides in a switch's control plane and communicates with the data plane table with PCIe-based runtime APIs. The primary responsibility of the ACL handler is twofold. One is to convert 5-tuple ACL rules in a form that is required by the data plane tables (i.e., IP and port). The ACL rule processing consists of four steps: IP and protocol rule deduplication and port ANY handling (①), port rule deduplication (②), port rule reversing (③), and LRME generation (④), as depicted in Figure 6. The other one is to store ACL rules and keep track of the data plane tables using the *shadow ACL* for quick ACL management, as shown in Figure 6. In the following, we describe the process of static ACL deployment and dynamic ACL rule management (i.e., insertion and deletion).

4.3.1 Static ACL deployment. A static ACL is a fixed set of rules driven by the network policies and is deployed when initializing switches. Therefore, it requires relatively a loose deadline. Algorithm 1 in Appendix A.1 illustrates the ACL rule handling process.

IP and protocol rule deduplication and port ANY handling. Given a list of 5-tuple ACL rules, the ACL handler first aggregates ACL rules based on IP and protocol fields using a linked hashmap with IP and protocol pair as the key and port rules as the values (lines 8-9). Therefore, duplicated IP and protocol definitions will be eliminated and each distinct IP and protocol rule pair is followed by a set of port (range) definitions as a linked list. During this process, the ACL rules are sorted into four groups: bothANY, srcANY, dstANY, and noANY (lines 10-13), based on the "ANY" setting in source and destination ports. Rules in the bothANY group are inserted into an IP (hashmap) table with the coupling field as "Drop" (line 9), so that the port matching will be skipped. The rule in the other groups are put into separate linked hashmaps (lines 10-13). Next, the ACL handler repeats port rule deduplication, port rule reversing, and LRME generation for $\text{Port}_{\texttt{srcANY}}$, $\text{Port}_{\texttt{dstANY}}$, and $\text{Port}_{\texttt{noANY}}$, independently (lines 15-18).

Port rule deduplication. Port rule deduplication is done by swapping the key/value of the port tables, i.e., <K, V> → <V, K> (line 18), so that the duplicated port rule sets are eliminated and each distinct port rule set is followed by a list of IP and protocol definitions. Note that the deduplication functions will group a number of rules and process them at once, in a way similar to a batch mode operation.

Port rule reversing. After obtaining port rule tables, the ACL handler reverses the port rule sets that cover more than 2^{15} ports to reduce the amount of LRMEs to be generated (line 17). For example, for a port rule set, (ANY, 1-512), (ANY, 1024-2048), and (ANY, 4096-65535), of an IP and protocol rule pair, its total coverage is 62,977 ports, larger than 2^{15}. Therefore, the REV_flag is "true". The flag is used in LRME generation function to generate LRMEs in a reversed manner. General in-network ACLs block malicious flows with "permit/deny" actions. For non-reversible actions (e.g., priority queue), PortCatcher has an option to skip the reversing.

LRME generation. LRME generation is a process of converting port rules into hash table entries (i.e., 32-bit or 64-bit bitmap). Figure 2 explains the LRME generation logic, and Algorithm 2 in Appendix explains in more details the used functions. Given a port (range) rule set (V), it generates a list of LRMEs paired with their PAIs (i.e., <PAI, LRME>), which are IDs or location information of the LRMEs. Eventually, these LRMEs are stored in an LRME linked hashmap (i.e., $\text{LRME}_{\texttt{srcANY}}$, $\text{LRME}_{\texttt{dstANY}}$, $\text{LRME}_{\texttt{noANY}}$) with V as their key. The uniqueness of LRM-ID is guaranteed by assigning the V's (LRM's) unique index in LRME tables (line 20). Moreover,

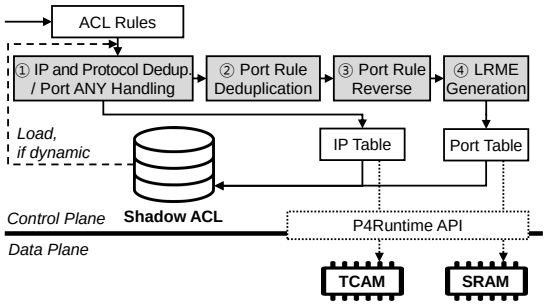


Figure 6: The rule encoding process of ACL handler in the control plane. Four functions help to synchronize ACL rules between control plane and data plane modules.

LRM-ID and REV_flag, as the information needed for coupling IP and port tables, are stored in the IP table combined with the IP and protocol definitions (lines 19-21).

Data plane installation. As shown in Algorithm 1 (lines 22-23), the ACL handler populates the entire IP table into TCAM in the data plane and installs LRMEs into the data plane's SRAM. Three SRAM-based hash tables are reserved in the data plane to store $\text{LRME}_{\texttt{srcANY}}$, $\text{LRME}_{\texttt{dstANY}}$, $\text{LRME}_{\texttt{noANY}}$ tables, separately. Moreover, when installing an LRME, its LRM-ID and PAI are used as the hash table key to ensure LRME can be uniquely located and identified.

4.3.2 Shadow ACL for dynamic rule management. As shown in Algorithm 1 (lines 22-24), the ACL handler also stores IP, port, and LRME tables in the control plane database, called `Shadow ACL`. Shadow ACL is responsible for synchronizing ACL rules installed in the data plane for fast and dynamic ACL management. We note that the data plane does not support the dynamic ACL management feature due to its limited programmability. Moreover, PortCatcher's data plane tables have only highly-compressed port rule representations (i.e., LRMEs) but not the raw port rules. Therefore, the shadow ACL is crucial for a seamless ACL management. The essential idea of our dynamic ACL management is to reuse IP, port, and LRME tables for low-computation and non-delayed rule insertion and deletion. Algorithm 3 in Appendix A.3 shows the detailed dynamic insertion and deletion operations.

Rule priority. We note that a rule priority notion is essential for any ACL systems, since the continuous rule deployment will overflow the finite data plane resources and new rules may conflict with previous rules. There has been a body of work [24, 73, 88, 89] focused on the rule priority issue, which can be leveraged by PortCatcher to evict rules on-demand.

5 EVALUATION

In this section, we first evaluate PortCatcher's memory efficiency and computational overhead using a real-world dataset-driven ACL benchmark. Next, we compare PortCatcher with the standard ACL_{TCAM} and the state-of-the-art ALPM [75] and discuss the trade-off between scalability and rule management latency. Then, we use an attack traffic-driven ACL dataset to show that PortCatcher can deploy dynamically-generated ACL rules with a low latency to block malicious traffic rapidly. Finally, we discuss the resource consumption and rule matching speed of PortCatcher in data plane.

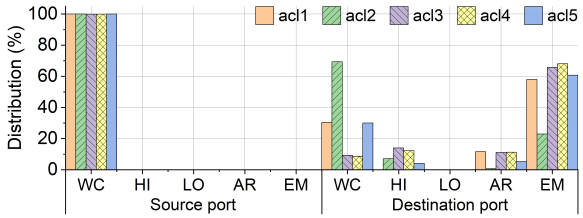


Figure 7: The distribution of datasets with five ACL seeds generated by ClassBench.

5.1 Testbed and Dataset

To show the feasibility of PortCatcher, we implemented our system in a programmable switch [58] with a low cost (section 5.5 for resource consumption and rule matching speed). The programmable switch is equipped with a Tofino2 data-plane processor [12], providing $O(1)$ MB TCAM and $O(10)$ SRAM capacity, and 1.6 GHz Pentium D-1517 control-plane CPU [58]. To evaluate our system, we interconnected the switch with a traffic generator equipped with AMD Ryzen$^{\text{TM}}$ 5 2400 G 8-core CPU, 16 GB of DRAM, and Intel XL710 40 Gbps network adapter. We used two datasets, namely ClassBench [72] and attack trace-driven [35] datasets.

Static ACL rules. ClassBench is used to evaluate PortCatcher's scalability in a static ACL scenario (see section 5.3). The dataset was created by analyzing 12 real-world datasets from Internet Service Providers (ISPs), network equipment vendors, and research institutes. Also, it covers all possible matching types (i.e., exact, range, and wildcard matching) with various distributions, as shown in Figure 7. Thus, it is widely used in network applications [10, 26, 39, 45, 54, 60]. As shown in Figure 7, five ACLs with 10K rules were created, varying the port rule definitions. We note that the ClassBench generates source IP, destination IP, and protocol with a uniform distribution. However, the distribution of the source and destination port definitions is controlled by pre-defined seeds, which define the generation probability among different rule types: wildcard (WC), high range (HI), low range (LO), arbitrary range (AR), and exact matching (EM). WC matches every port number (0-65535). HI and LO mean ephemeral user port range (1024-65535) and well-known system port range (0-1023), respectively. AR means arbitrary port range rule and EM rule matches one exact port. We can observe that WC dominates source port rules. On the contrary, the destination port rules show diverse distributions among seeds. Therefore, the destination port distribution will have major effects on the estimated results.

Dynamic ACL rules. In our use-case study on *attack traffic blocking performance with dynamic ACL*, the ACL dataset is generated using the CIC attack trace [35] to analyze PortCatcher's dynamicity (see section 5.4). The derived ACL dataset has only the exact matching rules transformed from all individual L4-flows (i.e., 306,355 flows in total), since the existing in-network autonomous detection system performs flow-level actions [14, 50, 80, 87].

5.2 Analysis of PortCatcher's Efficiency

5.2.1 Analysis of TCAM saving effects (IP table). Constrained by the data plane's TCAM, PortCatcher offloads port matching completely to SRAM, which results in shortened TCAM entry definition using only IP and protocol parts and the following TCAM saving.

Range-to-prefix expansion prevention. As discussed in the motivation, the ACL_{TCAM} suffers from a TCAM waste due to the range-to-prefix expansion for port range rules. Here, we assume ACL_{TCAM} is based on the widely used internal binary-prefix approach [70], which expands a port range rule to up to 30 prefix rules (i.e., TCAM entries) in the worst case. Since PortCatcher prevents the range-to-prefix expansion by performing only IP and protocol matching in TCAM, the number of TCAM entries required by PortCatcher is the same as the number of ACL rules in the worst case (i.e., black reference line in Figure 8). Figure 8 shows the number of TCAM entries required in ACL_{TCAM} when varying ACL dataset. As shown, `acl3` presents the highest expansion rate of 230%, since the rule set contains a large number of HI and AR port rules (i.e., 13% and 11%). The `acl4` has a similar amount of HI and AR rules, thus 10K ACL rules are expanded to 21K TCAM entries. `acl1` and `acl5` fall into a similar case, although major expansions are triggered by AR rules. Among the five rule set, `acl2` has the lowest expansion of 130%, since the rules mainly consisted of WC (69%) and EM (22%) rules.

TCAM entries deduplication. Due to the isolation design, PortCatcher can further reduce the number of entries by eliminating duplicate IP and protocol entries since it is common for ACL rules to have the same source IP, destination IP, and protocol definitions, but different port settings. The TCAM saving effect is demonstrated in Figure 8. As shown, PortCatcher can save 0.07~3.53K TCAM entries by deduplicating the TCAM entries. Compared with ACL_{TCAM}, PortCatcher can save 25%~72% of TCAM entries.

5.2.2 Analysis of SRAM saving effects (port table). Although SRAM is not as constrained as TCAM in a switch, it is still limited to tens of megabytes [13, 50]. Therefore, several optimization efforts have been made for PortCatcher, including port aggregation, port range reversing, ANY port handling, and port rule deduplication, to reduce the number of SRAM entries. Table 2 breaks down the SRAM saving effects in the execution order.

Baseline (LRM concept). In the baseline approach, a concrete port rule of an LRM is converted to an LRME and consumes one hash table entry. For example, the port range rule (src=30, dst=ANY) generates 65,536 LRMEs. As shown in Table 2-Baseline, ACL rule sets are converted to 0.26, 2.73, 3.63, 3.06 and 0.29 trillion LRMEs, respectively. Obviously, the amount of LRMEs is unacceptably large.

Port ANY rule handling. The memory efficiency can be improved notably by handling port ANY rules system-wise with separated hash tables and port table action logic. Therefore, the number of LRMEs is reduced by 2^{16} times compared with the baseline and saves 1.9 trillion LRMEs on average, as shown in Table 2-ANY.

Port rule deduplication. Since the IP and protocol entries, and the port entries are separately handled, we also remove the duplicated port rules, as we have done for the TCAM table. Intuitively, multiple IP and protocol rule pairs may have the same port rule sets producing the exact same LRM and LRMEs. To avoid such LRME duplication, we let different IP and protocol entries share the same LRM and LRMEs by assigning the same port table lookup key (i.e., LRM-ID). Table 2-Dedup. shows the number of LRMEs after deduplication. As shown, `acl2` shows the most significant LRME reducing effect (i.e., 166 times), which infers a large amount of IP and protocol rules have the same port rule settings. While `acl5` has the least amount of LRMEs, the number is further reduced by 2.59 times. Other ACL sets also saved SRAM memory significantly; by 5.48 times for `acl1`, 4.09 times for `acl3`, and 7.34 times for `acl4`.

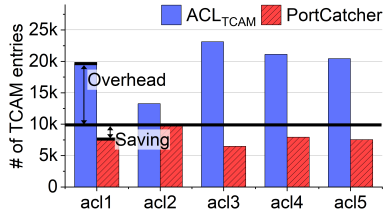


Figure 8: Analysis of TCAM saving effects. The number of TCAM entries with five 10K ACL rule datasets.

Table 2: Analysis of SRAM saving effects. The number of SRAM table entries with a function-wise breakdown: the baseline (i.e., LRM), port ANY handling (ANY), port rule deduplication (Dedup.), range reversing (REV), port aggregation (Aggr.). The values in parentheses show the entry reduction ratio compared with the previous column.

	LRM	ANY	Dedup.	REV	Aggr.
acl1	0.26T	4.11M (-99%)	0.75M (-81%)	61.08K (-91%)	3.59K (-94%)
acl2	2.73T	41.67M (-99%)	0.25M (-99%)	4.14K (-98%)	0.17K (-95%)
acl3	3.63T	55.42M (-99%)	13.53M (-75%)	272.64K (-97%)	10.74K (-96%)
acl4	3.06T	46.80M (-99%)	6.37M (-86%)	160.88K (-97%)	6.92K (-95%)
acl5	0.29T	4.52M (-99%)	1.74M (-61%)	30.29K (-98%)	2.87K (-90%)

Port range reversing. The next optimization is done for the port range rules with a broad coverage, which generates a long list of LRMEs (e.g., HI and AR). For such rules, we reverse a port range definition, e.g., from (1024-65535) to (0-1023), with a reversed action to reduce the amount of LRMEs. As shown in Table 2-REV, the number of LRMEs is reduced notably to 105.80K, on average, with the port range reversing. Noticeably, more significant LRME reducing effects can be observed for `acl2`~`acl4` (i.e., reduced 41 times on average), which have more HI rules.

LRM port aggregation. For LRM, we introduced a concept called port aggregation, which converts continuous ports into a port n-sized bitmap, namely n-port aggression. With $n = 32$ (i.e., 32-port aggregation), we expect that a 128 continuous port range to be expressed by only 4 LRMEs with port aggression. However, it may not be efficient for discrete port rule sets, as a 32-bit LRME may store only one port (i.e., EM). We observe that, although EMs are the most common port rules in the ACL rule sets, the port aggregation design is still able to reduce the number of LRMEs by around 32 times, as shown in Table 2-Aggr. It infers that the LRME saving by the port aggregation is more significant than the memory waste caused by discrete EM rules. As results, the number of LRMEs is reduced to 0.13, 1.30, 1.73, 1.46, and 0.14 million, respectively.

To conclude, our optimization designs show positive and significant effects in reducing the number of LRMEs (i.e., SRAM table entries) and saving SRAM memory. We note that the reduction of the table entries not only addresses the SRAM's scalability issue but also speeds up the rule deployment, since the table entries (i.e., LRMEs) are managed independently in the port table.

5.2.3 Analysis of ACL initialization latency (control plane overhead). The SRAM saving designs are integrated in our control plane module, called ACL handler, which is executed to manage ACL rules

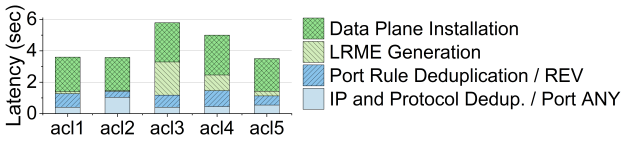


Figure 9: PortCatcher's ACL initialization latency breakdown: (1) IP and protocol rule deduplication and port ANY handling, (2) port rule deduplication and range reversing, (3) LRME generation, and (4) TCAM and SRAM table insertion.

(e.g., insertion and deletion). In the following, we analyze the ACL handler's performance by breaking down the time consumption (i.e., latency) of its core functions. This part of experiments simulates the ACL's initialization stage, where a network administrator configures an ACL with a long list of static ACL rules (i.e., 10K rules). After initialization, an autonomous intrusion detection module may dynamically insert or delete ACL rules online. Unlike the initialization stage, the dynamic ACL management handles less ACL rules but requires a much tighter deadline for the rule deployment. The performance of PortCatcher as a dynamic ACL will be further discussed in section 5.4.

Figure 9 shows the time consumption of each function in an execution order from bottom to up. **(1)** The first function to be executed by the ACL handler is the ANY port handling. As shown, `acl2` contributes the highest latency of 1.04 seconds among five datasets, since the dataset has the largest amount of port ANY rules (see Figure 7 for the WC rule's proportion). The other ACL rule datasets consume less time (i.e., 0.40~0.54 seconds), of which the trend follows the WC rule proportions of the five datasets. **(2)** The ACL handler creates LRMs after reversing some long-range port rules to shorter ones. `acl2` consumes the least amount of time since 69.34% of rules are WC and have been processed at the port ANY handling stage. **(3)** The ACL handler converts LRMs to SRAM table entries (i.e., LRMEs). The LRME generation is the most time-consuming task, thus the amount of LRMEs that need to be generated must be reduced. As shown in Table 2, the optimization design integrated in the ACL handler functions can dramatically reduce the number of LRMEs that have to be generated. Therefore, `acl1`, `acl2`, and `acl5` consume negligible time to generate LRMEs, since it requires 3.59K, 0.17K, and 2.87K LRMEs, respectively. `acl3` contributed the highest latency among the five datasets since the number of the generated LRMEs is the largest. **(4)** PortCatcher deploys the processed IP and protocol rules and LRMEs into the data plane tables. We can see that the time consumption of the five datasets is similar even though the number of LRMEs varies. This result shows that the TCAM entry (i.e., IP and protocol rules) deployment is the major factor that delays the ACL rule deployment in the data plane. Moreover, the result confirms that the TCAM entry reduction (i.e., prevention of range-to-prefix expansion) is important for not only memory efficiency but also the rule management latency.

5.3 System Performance with Static ACL

In the followings, we show the overall system performance of PortCatcher by comparing it with the legacy ACL_{TCAM} and state-of-the-art approach, ALPM, with a focus on the static ACL management. We implement ACL_{TCAM} with a P4 switch's built-in TCAM table and ALPM based on the disclosed patent [75]. For ALPM, we offloaded only the destination port to SRAM (i.e., 16-bit SRAM partitioning) since it is not designed for multi-tuple LPM and most of the source port rules are WC, as shown in Figure 7. With PortCatcher, we (1) offload both the source and destination port matching to SRAM using similar memory space needed by ALPM, and (2) save more TCAM space. In this experiment, we measure and analyze the memory consumption of both the data plane and control plane. Then, we measure the time consumption of the rule insertion and deletion under a static ACL management scenario. Moreover, we use an attack traffic-driven ACL dataset to simulate the autonomous attack defense. Finally, we discuss the overhead of PortCatcher added to the switch's data plane.

5.3.1 Overall memory consumption. Table 3 summarizes the data plane and control plane memory consumption of the three systems.

Data plane. Given 10K ACL rules, ACL_{TCAM} occupies more than 10K TCAM entries due to the range-to-prefix expansion. Since 104-bit five tuples and the associated 104-bit masks are all stored in TCAM, ACL_{TCAM} consumes 336~586 KB TCAM memory, which is the highest among the three ACL systems, as shown in Table 3. ALPM requires fewer TCAM entries than ACL_{TCAM}, since the destination port matching is offloaded to SRAM. Therefore, it only expands the source port range rules to multiple prefix rules (i.e., TCAM entries). We observe that even though all source port rules are WC, ALPM still requires more than 10K TCAM entries (i.e., 11.04K). This is due to a unique design of ALPM that groups eight SRAM entries into a partition by default, which means if a destination port rule is expanded to more than eight SRAM entries, more than one SRAM partition will be created, and the same number of TCAM entries will be generated to locate these partitions.

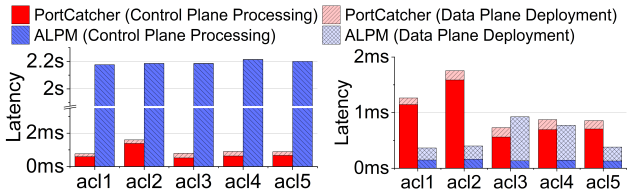
For ALPM's TCAM table, the per-entry size is reduced to 88 bits, since only the destination port matching is excluded. Overall, 135~146 KB of TCAM space is consumed, per Table 3. Among the three schemes, PortCatcher requires the least TCAM space, due to our protocol and port isolation (i.e., range-to-prefix expansion prevention). Per Table 3, PortCatcher consumes 6.47K~9.92K TCAM entries, for five 10K ACL rule sets. In other words, PortCatcher requires less than one TCAM entry per ACL rule, since PortCatcher stores only distinct IP and protocol entries in TCAM (i.e., TCAM entry deduplication). Moreover, since both the source and destination ports are matched in SRAM, the per-entry size of TCAM can be reduced to 72 bits and the overall TCAM consumption is reduced to 57~87 KB. To conclude, PortCatcher saves 74%-90% TCAM space compared to ACL_{TCAM} and 36%-61% compared to ALPM.

Unlike ACL_{TCAM}, PortCatcher and ALPM additionally consume SRAM memory. As shown in Table 3, although PortCatcher and ALPM consume similar amount of SRAM memory, the number of table entries of PortCatcher is significantly smaller than ALPM (4.8K vs. 19.2K on average), thanks to the LRM concept and optimization designs (see Table 2). The difference in numbers affects the ACL management–i.e., rule insertion and deletion–performance since the SRAM entries in both PortCatcher and ALPM are stored and operated independently (section 5.3.2 for more details).

Control plane. For continuous ACL management, ACL systems have to store and track the deployed ACL rules in the control plane. Especially for SRAM offloading, ACL system should cache intermediate data (e.g., extended prefixes for ACL_{TCAM}, LRM for

Table 3: Comparison for memory requirement in the data plane (TCAM and SRAM) and the control plane among PortCatcher, ALPM, and ACL_{TCAM} with five 10K ACL rule datasets.

	Data Plane										Control Plane		
	TCAM						SRAM				DRAM (MB)		
	Entry (#)			Memory (KB)			Entry (#)		Memory (KB)				
	ACL	ALPM	Ours	ACL	ALPM	Ours	ALPM	Ours	ALPM	Ours	ACL	ALPM	Ours
acl1	19,627	10,756	**7,642**	498	137	**67**	19,627	**3,590**	77	**73**	0.24	985	**0.22**
acl2	13,265	10,653	**9,928**	336	135	**87**	13,265	**176**	**52**	63	0.16	1,279	**0.13**
acl3	23,115	11,531	**6,476**	586	146	**57**	23,115	**10,744**	**90**	118	0.29	833	**0.21**
acl4	21,118	11,267	**7,927**	536	143	**70**	21,118	**6,928**	**82**	99	0.26	1,021	**0.22**
acl5	20,441	10,997	**7,525**	520	140	**66**	20,441	**2,874**	80	**68**	0.25	970	**0.20**



(a) Insertion (b) Deletion

Figure 10: Static ACL management: the average per-rule latency for PortCatcher and ALPM.

PortCatcher, and Trie for ALPM) generated during the rule encoding process to reduce the redundant computations and to guarantee the real-time performance. However, a naïve encoding approach may generate a massive amount of intermediate data or require heavy computations, which is undesirable for a dynamic ACL management scenario (e.g., online autonomous defense).

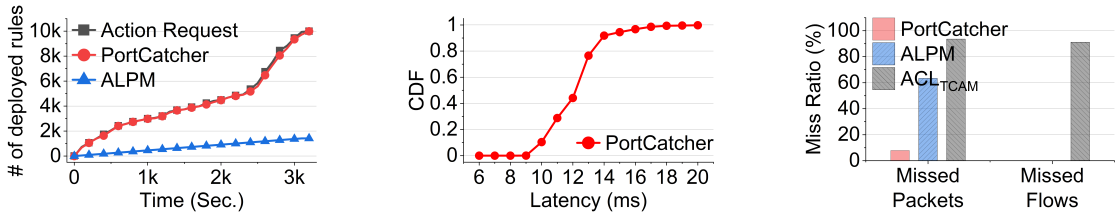
Table 3 shows the control plane memory consumption of three schemes for storing intermediate data. For ACL_{TCAM}, the required memory is doubled compared to the TCAM consumption, since the masks ("don't care" bits or bit-wise ternary matching information) have to be stored separately. We note that ACL_{TCAM} generates table entries directly with the range-to-expansion logic, which consumes more TCAM entries (i.e., scalability issue) and delays the rule deployment (i.e., latency issue). We observed that ALPM presented the highest overhead, since it has to maintain both the encoded table entries and 16-depth trie data structures for destination port rules. Such a trie is needed for every distinct IP, protocol, and source port rule (i.e., TCAM entry in ALPM). As such, ALPM consumes 833~1,279 MB DRAM memory in the control plane, per Table 3. PortCatcher's control plane memory consumption is lower than ALPM, since PortCatcher uses a compact bitmap representation for ACL port rules (i.e., LRM). Moreover, our optimization designs eliminate duplicated expressions. While PortCatcher's requires slightly more memory than ACL_{TCAM} in the data plane, the LRM-based SRAM offloading helps save TCAM space significantly (i.e., 74%-90% of TCAM space saving), as shown in Table 3.

5.3.2 ACL management latency. Next, we use additional 2K ACL rules to simulate static ACL management and measure the per rule management latency. We exclude ACL_{TCAM} in this experiment due to its unsatisfiable scalability in the data plane.

Insertion. We deploy 10K static ACL rules into the switch's data plane, and 2K additional rules are independently processed in the control plane and deployed into the data plane. Subsequently, we measure the per-rule time consumption (i.e., latency) for the 2K rules. The rule insertion is different from ACL initialization (§5.2.3) in that it needs to revisit and update the intermediate data (e.g., LRM or trie) recorded for existing rules. Such changes may lead to deletion/update of the TCAM and SRAM table entries. Figure 10(a) shows the average per-rule insertion latency of PortCatcher and ALPM. The data plane and control plane latency are shown separately. With PortCatcher, the latency for the five ACL datasets is 0.76 ms~1.59 ms , which is much smaller than ALPM's latency at 2.19 seconds on average. During the insertion process, PortCatcher triggered only 0.49~1.38 SRAM entry insertions per rule on average. We note that ALPM presents a very high overhead when constructing a trie for new port (range) rules, which in turn delays the overall rule deployment. PortCatcher's time consumption mainly depends on the number of LRMEs to be generated, as shown in Algorithm 2 in Appendix A.2. As can be seen in Table 2, PortCatcher's optimization designs reduce this number significantly.

Deletion. In some cases, inactive rules need to be removed from the ACL in order to release memory space for new rules. Both PortCatcher and ALPM support an ACL rule deletion function. Therefore, we measure and compare the latency in removing inserted ACL rules based on the previous experiments. Figure 10(b) shows the per-rule deletion latency over five ACL rule datasets. PortCatcher provides 0.72 ms to 1.75 ms for five ACL datasets, with 0.30~1.12 SRAM entry deletions per rule. On the one hand, with ALPM, the deletion latency is 0.36 ms to 0.92 ms, depending on the dataset. PortCatcher shows better performance than ALPM for `acl3`, although it does not show a good performance for other datasets. Moreover, ALPM's deletion is faster since the simple trie-based deletion can be done in $O(M)$ time, where M is the key size (32 bits for ALPM). However, the simple logic presents a huge memory overhead (~1 GB of DRAM for only offloading destination ports), per Table 3. However, PortCatcher is slower than ALPM with some datasets since it has to reconstruct LRMEs for memory efficiency. Instead, PortCatcher requires only a few hundred KBs of memory in the control plane. Although the results vary depending on the dataset, the overall result show that both are comparable for deletion time. Moreover, considering the huge insertion delay with ALPM, PortCatcher is more reliable in static ACL management.

Remarks. PortCatcher and ALPM share the same data plane setting (hardware), thus the comparison is fair. The fundamental difference between the two schemes is the control plane ACL rule processing logic (software). The results suggest that ALPM's software processing dominates the overhead in terms of the overall latency, whereas our LRM-based processing is lightweight.



(a) # of deployed rules over time (b) Latency distribution of deployed rules (c) # of missed packets and flows

Figure 11: Attack traffic blocking with dynamic ACL: Rule deployment latency of ALPM and PortCatcher. (a) shows the number of deployed rules over time compared with the requested actions. (b) shows the distribution of per-rule latency of PortCatcher, (c) shows the number of missed packets and missed flows varying ACL systems

5.4 Attack Traffic Blocking with Dynamic ACL

For an end-to-end system evaluation, we use a general framework proposed by a recent autonomous defense system [14], where a per-flow attack detection module resides in the switch's control plane and generates dynamic ACL rules for suspicious flows. Meanwhile, the control plane module of the ACL system will be responsible for deploying the ACL rules into data plane module. Since our focus is only the ACL performance, we assume an error-free and zero-delay detection module. Then, we measure rule deployment latency, which starts from a rule installation request and ends upon completion of deployment. We also introduce two metrics, the missed flows and packets, to show the explicit dynamicity requirements.

To measure the rule deployment latency, we first deployed 3K static ACL rules using ClassBench (`acl1`). Then, we replayed 10K flows of CIC 2017 attack trace [35] to trigger per-flow dynamic rule installations. Since we assume the detection module is error-free with zero latency, it generates a dynamic ACL rule immediately whenever a new attack flow is observed. As a result, 10K L4-ACL rules will be generated and installed in the switch's data plane. To measure the number of missed packets and flows, we conducted a simulation with the full trace of the CIC dataset, which contains 6,378,442 packets and 306,355 flows.

5.4.1 Rule deployment latency. Figure 11(a) shows the number of deployed rules over time for PortCatcher and ALPM. The black line with square dots shows the accumulated rule installation requests (i.e., action requests) over time. The closer a line is, the faster the requested rules are installed in the data plane. As shown, PortCatcher can install dynamic ACL rules without delay, whereas ALPM fails in handling the dynamic rule deployment due to the huge rule installation latency. This result matches the previous analysis, as shown in Figure 10(a). Figure 11(b) shows the per-rule latency distribution of PortCatcher. Among the 10K rules, 44.12% of the rules are deployed within 12 ms. Moreover, 76.52% of the rules can be installed within 13 ms. As a result, PortCatcher consumes 13.10 ms on average to insert a single rule, with our switch's control-plane CPU (i.e., Intel Pentium 1.6 GHz), which is 168 times lower than ALPM's 2.19 seconds.

Remarks. The latency mainly depends on the CPU performance in the switch's control plane. Since LRM-based rule processing is lightweight, the latency can be tightened further with a better CPU. To verify our assumption, we emulated the same experiment with a better CPU (i.e., Intel Xeon 6230R 2.10 GHz). Per the result, the average per-rule latency dropped rapidly to 5.93 ms. Considering that our attack trace's average inter-packet arrival time (IAT) is 13.54 ms, PortCatcher should be capable of blocking most attack traffic immediately. Last but not least, we note that the control-plane CPU is an upgradable option for switch vendors.

5.4.2 Analysis of missed packets and flows. Figure 11(c) shows the number of missed packets and missed flows when considering three schemes: PortCatcher, ALPM, and ACL_{TCAM}. As shown, PortCatcher missed only 7.91% of the attack packets, out of a total of 6,378,442 packets, with 13.10 ms rule deployment latency, which is much lower than ALPM's 63.36% with 2.19 seconds latency. Moreover, ACL_{TCAM} missed 93.54% of the packets, since it missed 91.18% malicious flows out of 306,355 flows due to memory constraints.

Remarks. Per the results, ACL_{TCAM} is not a viable option for dynamic ACL due to its poor scalability. Since PortCatcher and ALPM did not miss any flows, it is clear that ALPM's high latency in rule deployment caused the massive packet missing. Also, we conclude that PortCatcher's rule deployment latency meets the requirement for the CIC dataset, since it successfully filtered out 92% of the attack traffic. PortCatcher consumes 9-Byte TCAM and 13-Byte SRAM memory per L4-ACL rule. As such, the overall data plane memory consumption for the 306,355 flows is 2.62 MB and 3.79 MB for TCAM and SRAM, respectively, which means that PortCatcher can work under a higher traffic volume. Appendix A.4 describes the scalability analysis in more detail.

5.5 Used Resource and Matching Speed

Resource consumption. PortCatcher's matching logic in P4 successfully fits into the data plane of the resource-constrained Tofino programmable switch with 338 lines of P4 code: 70 lines for matching logic and 268 lines for data structure and function definition. We present the implementation details in Appendix A.5. We sequentially break down the data plane codes of PortCatcher into four sequential functions: IP table matching, LRME lookup, port table matching, and reversed matching. Then, we measured the hardware resources consumed by the four sequential functions of PortCatcher, where the results are normalized by the clean switch implementation (i.e., switch.p4 [5]).

Table 4 shows the normalized resource usage of the switch. As shown, an average of 4.17% of TCAM, 2.47% of SRAM, 1.25% of Hash Bit, and 2.60% of Arithmetic Logic Unit (ALU) are used for the four functions over all stages. The other resources (gateway, match search bus, etc.) were automatically allocated for the data-control plane communication, parser, deparser, etc. Overall, our data

Table 4: Function-wise resource usage for PortCatcher.

	Optimization Functions				
Resource Type	IP & Protocol Match.	LRME Lookup	Port Match.	Rev. Match.	AVG.
TCAM	16.67%	0.00%	0.00%	0.00%	4.17%
SRAM	0.42%	7.71%	0.42%	1.35%	2.47%
Hash Bit	0.00%	2.80%	1.60%	0.60%	1.25%
ALU	2.08%	4.17%	2.08%	2.08%	2.60%

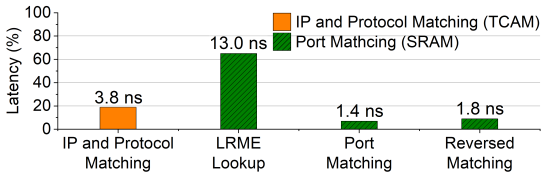


Figure 12: Function-wise packet processing latency of PortCatcher analyzed by P4 compiler's log.

plane codes added 6.72% overhead. We note that Tofino's compiler strictly limits the computation of the data plane algorithms, and PortCatcher's logic was shown in Table 4 to fit into the switch's pipeline resource constraints. This means that adding PortCatcher still guarantees the line-rate packet processing [36, 56, 83].

Rule matching speed. PortCatcher's data plane logic easily fits the resource-constrained packet processing pipeline of the Tofino switch, which means PortCatcher supports the full line rate of packet processing [56]. Moreover, it adds only several dozens of nanosecond in terms of latency to the pipeline (see Figure 12), and a large room remains for other tasks. We note that the data-plane clock-level latency is retrieved from the vendor compiler-generated logs. ACL_{TCAM}, ALPM, and PortCatcher add 3 ns (5 clocks with 1.6 GHz control-plane CPU), 8 ns (14 clocks), and 20 ns (32 clocks) data-plane latency, respectively, without affecting the line-rate processing. Even though PortCatcher does not affect the throughput, it is factorized to see which part of its logic contributes most to the overall latency. Function-wise, PortCatcher code was broken down into the IP table matching, LRME lookup, port table matching, and reversed matching, following the running logic. The packet processing latency was then analyzed from the P4 compiler's log. Figure 12 shows the function-wise data plane latency when adding and compiling the functions sequentially. As shown, LRME lookup in SRAM contributes the most to PortCatcher's total latency (65.62%), followed by the IP table matching of 18.75%. The port table matching and reversed matching add latencies of 6.25% and 9.38%, respectively. Overall, stage-1 with TCAM occupies 18.75%, and stage-2 with SRAM occupies 81.25%. We note that the latency given by the compiler log is based on the worst-case analysis.

6 RELATED WORKS

Static and dynamic ACLs are essential security functions of in-network defense systems, for both routine and adaptive defenses. Benefiting from recent data structures [23, 29, 30, 32, 34, 42, 43, 46, 48, 49, 59, 69, 77, 83] and machine learning techniques [11, 28, 57, 64, 65, 85, 86], a body of in-network defense systems has been proposed [14, 18, 27, 38, 50, 80, 87]. Once detected as malicious, the in-network/switch ACL will be responsible for dropping the flow's packets. For instance, *Poise* [38] relies on a deep packet inspection server to identify malicious flows. Once detected, a new ACL rule of the corresponding flow should be installed into the data plane ACL for mitigation actions. Such a dynamicity of recent defense frameworks necessitates scalable and low-latency ACL systems (i.e., dynamic ACL).

Existing ACL systems improve TCAM's memory efficiency by exploring compact conversions of range rules into prefix matching rules [16, 17, 41, 44, 47, 52, 53, 61, 62, 68, 70, 71, 90], where the optimization effects are negligible for the dynamic ACL rules dominated by the exact matching and improvement is limited for static ACL rules with the scarcity of TCAM memory. The other direction is to offload the partial or entire matching from TCAM to SRAM [19, 25, 74, 84]. While the scalability can be addressed for static and dynamic ACLs, the complex operations of these solutions result in high time complexity, thus cannot meet the latency requirement of the in-network autonomous defense systems.

PortCatcher is a standalone in-network/switch ACL system that supports in-network autonomous defense systems with scalable and low-latency ACL rule management. To scale up the defense, these autonomous defense systems [14, 50, 80, 87] can benefit from distributed measurement systems [15, 63, 81]. Recent promising works, such as *Ripple* [80], have opened a possibility to defend against link-flooding attacks using a decentralized detection and mitigation strategies. Moreover, *Bedrock* [79] enriches in-network ACL's security application by showing a way to secure a remote memory direct access (RMDA) system with address range matching.

7 CONCLUSION

Today's paradigm for port matching leveraging hardware acceleration suffers from a trade-off between the scalability, due to an ever-growing number of ACL rules, and the high rule deployment latency, due to the high complexity of the encoding mechanisms. To break this trade-off, we proposed a new direction for performing ACL port range matching in SRAM, while strictly separating IP/protocol matching in TCAM. To enable a fast and scalable port management in SRAM, a novel range representation method, called "Linear Range Map", was proposed. Experiments on a programmable switch showed impressive results, where one ACL rule was encoded into 0.79 TCAM entry and 0.49 SRAM entry on average, and 0.35x-0.61x TCAM space was saved in comparison to state-of-the-art approaches. PortCatcher also has a negligible rule management latency, which is necessary to block attack flows in real-time. Our experiment showed that PortCatcher's rule deployment is 168x faster than the state-of-the-art approach, allowing our in-network defense system to block 92.09% (55.45% more) malicious packets and all flows in an attack trace.

ACKNOWLEDGMENT

This research was supported by the Global Research Laboratory (GRL) Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2016K1A1A2912757) and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2020R1A2C2009372). DaeHun Nyang and Rhongho Jang are the corresponding authors. The authors would like to thank the anonymous reviewers of CCS'22 for the thorough feedback that helped improve this paper.

REFERENCES

[1] 2018. GitHub Survived the Biggest DDoS Attack Ever Recorded. https://bit.ly/3ByHVj8.
[2] 2019. ACL TCAM and LOUs in Catalyst 6500. https://bit.ly/3bgRQyF
[3] 2019. Catalyst 7000 Troubleshooting TechNotes. https://bit.ly/2RGC49z
[4] 2020. *10k Device Family - Switch Architecture Specification.* Barefoot Networks.
[5] 2020. The P4 Language Consortium. https://github.com/p4lang/switch
[6] 2021. NetFPGA. https://netfpga.org/
[7] 2021. P4-NetFPGA-public. https://github.com/NetFPGA/P4-NetFPGA-public/wiki
[8] 2022. Cisco Nexus 3000 Series NX-OS Security Configuration Guide, Release 7.x. https://bit.ly/2RJPNfS
[9] 2022. Cisco Nexus 7000 Series NX-OS Security Configuration Guide. https://bit.ly/3KqeTpG
[10] Mahdi Abbasi, Hajar Rezaei, Varun G Menon, Lianyong Qi, and Mohammad R Khosravi. 2020. Enhancing the performance of flow classification in SDN-based intelligent vehicular networks. *IEEE Transactions on Intelligent Transportation Systems* 22, 7 (2020), 4141–4150.
[11] Ahmed Abusnaina, Aminollah Khormali, DaeHun Nyang, Murat Yuksel, and Aziz Mohaisen. 2019. Examining the robustness of learning-based ddos detection in software defined networks. In *2019 IEEE Conference on Dependable and Secure Computing (DSC)*. IEEE, 1–8.
[12] Intel Barefoot. Accessed August 8, 2022. *Intel Intelligent Fabric Processors.* https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html
[13] Intel Barefoot. Accessed August 8, 2022. *Intel® Tofino™ 3 Intelligent Fabric Processor.* https://www.intel.com/content/dam/www/central-libraries/us/en/documents/product-brief-final-version-pdf.pdf
[14] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando M. V. Ramos, and André Madeira. 2021. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society.
[15] Ran Ben Basat, Gil Einziger, and Bilal Tayh. 2020. Cooperative Network-wide Flow Selection. In *28th IEEE International Conference on Network Protocols, ICNP 2020, Madrid, Spain, October 13-16, 2020*. IEEE, 1–11.
[16] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Hel-Or. 2018. Encoding Short Ranges in TCAM Without Expansion: Efficient Algorithm and Applications. *IEEE/ACM Transactions on Networking* 26, 2 (2018), 835–850.
[17] Anat Bremler-Barr and Danny Hendler. 2012. Space-Efficient TCAM-Based Classification Using Gray Coding. *IEEE Trans. Computers* 61, 1 (2012), 18–30.
[18] Xiaoqi Chen, Shir Landau Feibish, Mark Braverman, and Jennifer Rexford. 2020. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, Henning Schulzrinne and Vishal Misra (Eds.). ACM, 226–239.
[19] Y. Cheng and P. Wang. 2016. Scalable Multi-Match Packet Classification Using TCAM and SRAM. *IEEE Trans. Comput.* 65, 7 (2016), 2257–2269.
[20] Catalin Cimpanu. 2020. AWS said it mitigated a 2.3 Tbps DDoS attack, the largest ever. https://zd.net/2YGSMJh.
[21] Cisco. [n.d.]. *Software-Defined Networking and Network Programmability: Use Cases for Defense and Intelligence Communities.* https://bit.ly/33SBtnK
[22] The P4 Language Consortium. 2018. $P4_{16}$ Language Specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html.
[23] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.
[24] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Kőrösi, Balázs Sonkoly, Dávid Haja, Dimitrios P Pezaros, Stefan Schmid, and Gábor Rétvári. 2019. Tuple space explosion: A denial-of-service attack against a software packet classifier. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 292–304.
[25] Q. Dai and H. Li. 2018. An Advanced TCAM-SRAM Architecture for Ranges Towards Minimizing Packet Classifiers. In *Proceedings of the 20th IEEE International Conference on High Performance Computing and Communications*. 158–163.
[26] James Daly, Valerio Bruschi, Leonardo Linguaglossa, Salvatore Pontarelli, Dario Rossi, Jerome Tollet, Eric Torng, and Andrew Yourtchenko. 2019. Tuplemerge: Fast software packet processing for online packet classification. *IEEE/ACM transactions on networking* 27, 4 (2019), 1417–1431.
[27] Dinhnguyen Dao, Rhongho Jang, Changhun Jung, David Mohaisen, and DaeHun Nyang. 2022. Minimizing Noise in HyperLogLog-Based Spread Estimation of Multiple Flows. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Baltimore, MD, USA, June 27-30, 2022*. IEEE, 331–342.
[28] Rohan Doshi, Noah Apthorpe, and Nick Feamster. 2018. Machine learning ddos detection for consumer internet of things devices. In *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 29–35.
[29] Cristian Estan and George Varghese. 2002. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 323–336.
[30] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.
[31] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2021. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. *arXiv preprint arXiv:2101.10632* (2021).
[32] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *ACM SIGCOMM 2017*. ACM, 113–126.
[33] Muhammad Ibrar, Lei Wang, Gabriel-Miro Muntean, Aamir Akbar, Nadir Shah, and Kaleem Razzaq Malik. 2021. PrePass-Flow: A Machine Learning based technique to minimize ACL policy violation due to links failure in hybrid SDN. *Computer Networks* 184 (2021), 107706.
[34] Rhongho Jang, DaeHong Min, Seongkwang Moon, David Mohaisen, and DaeHun Nyang. 2020. SketchFlow: Per-Flow Systematic Sampling Using Sketch Saturation Event. In *IEEE INFOCOM 2020*. IEEE, 1339–1348.
[35] Hossein Hadian Jazi, Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. 2017. Detecting HTTP-based application layer DoS attacks on web servers in the presence of sampling. *Computer Networks* 121 (2017), 25–36.
[36] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 35–49.
[37] Min Suk Kang, Soo Bum Lee, and Virgil D. Gligor. 2013. The Crossfire Attack. In *Proceedings of the IEEE Symposium on Security and Privacy, SP*. IEEE Computer Society, 127–141.
[38] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. 2020. Programmable In-Network Security for Context-aware BYOD Policies. In *29th USENIX Security Symposium (USENIX Security 20)*. 595–612.
[39] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. 2018. Metron:{NFV} service chains at the true speed of the underlying hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 171–186.
[40] Naga Praveen Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research, SOSR 2016, Santa Clara, CA, USA, March 14 - 15, 2016*, Brighten Godfrey and Martín Casado (Eds.). ACM, 10.
[41] Y. Kim, H. Ahn, S. Kim, and D. Jeong. 2009. A High-Speed Range-Matching TCAM for Storage-Efficient Packet Classification. *IEEE Transactions on Circuits and Systems I: Regular Papers* 56, 6 (2009), 1221–1230.
[42] Abhishek Kumar, Minho Sung, Jun Xu, and Jia Wang. 2004. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS PER* 32, 1 (2004), 177–188.
[43] Abhishek Kumar and Jun (Jim) Xu. 2006. Sketch Guided Sampling - Using On-Line Estimates of Flow Size for Adaptive Data Collection. In *INFOCOM 2006*. IEEE.
[44] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. 2005. Algorithms for advanced packet classification with ternary CAMs. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, Roch Guérin, Ramesh Govindan, and Greg Minshall (Eds.). ACM, 193–204.
[45] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. 2018. Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2645–2653.
[46] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *NSDI 2016*. USENIX Association, 311–324.
[47] A. X. Liu, C. R. Meiners, and E. Torng. 2010. TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs. *IEEE/ACM Transactions on Networking* 18, 2 (2010), 490–500.
[48] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM 2019*. 334–350.
[49] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM 2016*. 101–114.
[50] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*.
[51] Soumya Maity, Padmalochan Bera, and SK Ghosh. 2012. Policy based acl configuration synthesis in enterprise networks: A formal approach. In *2012 International*

Symposium on Electronic System Design (ISED). IEEE, 314–318.
[52] C. R. Meiners, A. X. Liu, and E. Torng. 2011. Topological Transformation Approaches to TCAM-Based Packet Classification. *IEEE/ACM Transactions on Networking* 19, 1 (2011), 237–250.
[53] C. R. Meiners, A. X. Liu, and E. Torng. 2012. Bit Weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs. *IEEE/ACM Transactions on Networking* 20, 2 (2012), 488–500.
[54] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. 2021. A framework for eBPF-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 133–151.
[55] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*. ACM, 15–28.
[56] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 15–28.
[57] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. 2018. Kitsune: an ensemble of autoencoders for online network intrusion detection. *NDSS* (2018).
[58] Edgecore Networks. 2020. WEDGE 100BF-32X. https://bit.ly/2YDeyv2.
[59] D. Nyang and D. Shin. 2016. Recyclable Counter With Confinement for Real-Time Per-Flow Measurement. *IEEE/ACM Trans. Netw.* 24, 5 (2016), 3191–3203.
[60] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2020. A computational approach to packet classification. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 542–556.
[61] Ori Rottenstreich and Isaac Keslassy. 2010. Worst-Case TCAM Rule Expansion. In *INFOCOM 2010. 29th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 15-19 March 2010, San Diego, CA, USA*. IEEE, 456–460.
[62] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat. 2016. Optimal In/Out TCAM Encodings of Ranges. *IEEE/ACM Transactions on Networking* 24, 1 (2016), 555–568.
[63] Vyas Sekar, Michael K Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G Andersen. 2008. cSamp: A system for network-wide flow monitoring. (2008).
[64] Stefan Seufert and Darragh O'Brien. 2007. Machine Learning for Automatic Defence Against Distributed Denial of Service Attacks. In *Proceedings of IEEE International Conference on Communications, ICC*. IEEE, 1217–1222.
[65] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. 2018. Toward generating a new intrusion detection dataset and intrusion traffic characterization.. In *ICISSp*. 108–116.
[66] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. 2015. Dc. p4: Programming the forwarding plane of a datacenter switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 1–8.
[67] Jared M. Smith and Max Schuchard. 2018. Routing Around Congestion: Defeating DDoS Attacks and Adverse Network Conditions via Reactive BGP Routing. In *Proceedings of the IEEE Symposium on Security and Privacy, SP*. IEEE Computer Society, 599–617.
[68] E. Spitznagel, D. Taylor, and J. Turner. 2003. Packet classification using extended TCAMs. In *11th IEEE International Conference on Network Protocols, 2003. Proceedings.* 120–131.
[69] Robert H. Morris Sr. 1978. Counting Large Numbers of Events in Small Registers. *Commun. ACM* 21, 10 (1978), 840–842.
[70] Venkatachary Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. 1998. Fast and Scalable Layer Four Switching. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Gerald Neufeld, Gary S. Delp, Jonathan Smith, and Martha Steenstrup (Eds.). ACM, 191–202.
[71] Vegesna SM Srinivasavarma and Shiv Vidhyut. 2020. A TCAM-based caching architecture framework for packet classification. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 1 (2020), 1–19.
[72] DE Taylor and JS Turner. 2004. *ClassBench: a packet classification benchmark, WUCSE-2004-28.* Technical Report. Saint Louis: Department of Computer Science Engineering, Washington University.
[73] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. 2019. Safely and automatically updating in-network ACL configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*. 214–226.
[74] Anees Ullah, Pedro Reviriego, et al. 2020. FlexTCAM: Beyond Memory Based TCAM Emulation on FPGAs. In *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 110–113.
[75] Henry Wang. 2019. Algorithmic Longest Prefix Matching in Programmable Switch. https://patents.google.com/patent/US10511532B2/en
[76] Shie-Yuan Wang, Hsien-Wen Hu, and Yi-Bing Lin. 2020. Design and Implementation of TCP-Friendly Meters in P4 Switches. *IEEE/ACM Trans. Netw.* 28, 4 (2020), 1885–1898.
[77] Kyu-Young Whang, Brad T. Vander Zanden, and Howard M. Taylor. 1990. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Trans. Database Syst.* 15, 2 (1990), 208–229.
[78] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. 2012. NetPilot: Automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 419–430.
[79] Jiarong Xing, Kuo-Feng Hsu, Yiming Qiu, H Yang, Hongyi Liu, and Ang Chen. 2021. Bedrock: Programmable Network Support for Secure RDMA Systems. In *Proceedings of the 31th USENIX Security Symposium (USENIX Security'22). USENIX Association*.
[80] Jiarong Xing, Wenqing Wu, and Ang Chen. [n.d.]. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.).
[81] Hongli Xu, Shigang Chen, Qianpiao Ma, and Liusheng Huang. 2019. Lightweight Flow Distribution for Collaborative Traffic Measurement in Software Defined Networks. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*. IEEE, 1108–1116.
[82] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. 2005. Reducing Unwanted Traffic in a Backbone Network. *SRUTI* 5 (2005), 9–15.
[83] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 561–575.
[84] Weiwen Yu, Srinivas Sivakumar, and Derek Pao. 2019. Pseudo-TCAM: SRAM-based architecture for packet classification in one memory access. *IEEE Networking Letters* 1, 2 (2019), 89–92.
[85] Xiaoyong Yuan, Chuanhuang Li, and Xiaolin Li. 2017. DeepDefense: identifying DDoS attack via deep learning. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 1–8.
[86] Xiaoyong Yuan, Chuanhuang Li, and Xiaolin Li. 2017. DeepDefense: Identifying DDoS Attack via Deep Learning. In *Proceedings of the 2017 IEEE International Conference on Smart Computing*. IEEE Computer Society, 1–8.
[87] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. 2020. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*.
[88] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 241–255.
[89] Shuyuan Zhang, Franjo Ivancic, Cristian Lumezanu, Yifei Yuan, Aarti Gupta, and Sharad Malik. 2014. An adaptable rule placement for software-defined networks. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 88–99.
[90] Jincheng Zhong and Shuhui Chen. 2021. Efficient multi-category packet classification using TCAM. *Computer Communications* 169 (2021), 1–10.

A APPENDIX

A.1 Static ACL Initialization

Algorithm 1 illustrates the ACL handler's rule handling process for initializing static ACL rules. Detailed descriptions can be found in section 4.3.

A.2 Utility Functions of the ACL Handler

In Algorithm 2 provides the ACL handler's utility functions, namely **Port_Reverse()** and **LRME_Generation()**, as follows:

❶ **Port_Reverse().** Port_Reverse() implements a utility function that calculates the entire range and sets of REV flag per IP and protocol rule. When a port rule set V is considered, this function first calculates the coverage (COV) of the range. For instance, if V is <ANY,[0-20, 100-1024]>, COV of V is $20-0+1+1024-100+1 = 946$, and if V is <[0-9],[0-20, 100-1024]>, COV is $946*10 = 9460$. Then, the function checks if the port rule contains ANY case, and compares

Algorithm 1: Static ACL Initialization

```
1  Inputs: L = {R_1, ..., R_n}, R_i = (src_IP, dst_IP, protocol, src_Port, dst_Port).
2  Outputs: Shadow ACL, Data Plane_TCAM, Data Plane_SRAM
3  Initialize HashMap IP <K, C>; /* K is IP and protocol rule pair and C is coupling
     field between IP and port tables */
4  Initialize LinkedHashMap Port_srcANY<K, V>, Port_dstANY<K, V>, Port_noANY<K,
     V>; // V is a set of port rules as a linked list
5  Initialize LinkedHashMap LRME_srcANY<V, L>, LRME_dstANY<V, L>,
     LRME_noANY<V, L>; // L is a list of <PAI, LRME> pairs
6  /*IP and Protocol Rule Deduplication and Port ANY Handling*/
7  for each R in L do
8      Key ← <R.src_IP, R.dst_IP, R.Protocol>
9      Value ← <R.src_Port, R.dst_Port>
10     if Value is bothANY then IP.add(Key, "Drop");
11     else if R.src_Port is ANY then Port_srcANY.add(Key, Value);
12     else if R.dst_Port is ANY then Port_dstANY.add(Key, Value);
13     else Port_noANY.add(Key, Value);
14 /*Repeat lines 15-21 for Port_dstANY and Port_noANY*/
15 Port_srcANY<V, K> ← Port_Rule_Deduplication(Port_srcANY<K, V>)
16 for each V in Port_srcANY do
17     REV_flag ← Port_Reverse(V); /*Port Rule Reversing*/
18     LRME_srcANY[V] ← LRME_generation(V, REV_flag); /* see Algorithm 2 in
         Appendix for details. */
19     for each K in Port_srcANY[V] do
20         LRM-ID ← Port_srcANY.indexOf(V);
21         IP.add(K, LRM-ID, REV_flag);
22 Shadow ACL, Data Plane_TCAM ← IP
23 Shadow ACL, Data Plane_SRAM ← LRME_srcANY, LRME_dstANY, LRME_noANY
24 Shadow ACL ← Port_srcANY, Port_dstANY, Port_noANY
```

Algorithm 2: ACL Handler: Utility Functions

```
1  Inputs: V ← <R.src_Port, R.dst_Port>
2  /*Set Rev flag*/
3  Def Port_Reverse(V):
4      COV = calculate_coverage(V);
5      if R.src_Port is ANY or R.dst_Port is ANY then
6          if COV > 2^15 then REV_flag = true; else REV_flag = false;
7      else
8          if COV > 2^31 then REV_flag = true; else REV_flag = false;
9      return REV_flag;
10 /*LRM(E) generation*/
11 Initialize List LRME_List <LRME, PAI>;
12 Def LRME_Generation(V, REV_flag):
13     if REV_flag = true then V ← Subtract([0,2^16-1], V); /* Subtract(A,B)
         returns A-B for set A, B. */
14     for each R in V do
15         for each PAI in range(R.start/32, R.end/32) do
16             LRME ← 2^32-1;
17             if PAI == (R.start/32) then
18                 LRME ← LRME & ((2^32-1) << R.start % 32);
19             if PAI == (R.end/32) then
20                 LRME ← LRME & ((2^32-1) >> (31 - R.end % 32));
21             LRME_List.append(PAI, LRME);
22     return LRME_List;
```

COV to the constant 2^{15} or 2^{31} in each case. If COV is greater than the constant, the REV_flag is true. Otherwise, REV_flag is false and the function returns the value.

❷ **LRME_Generation().** LRME_Generation() implements a utility function that generates the corresponding LRMEs depending on the port range and REV flag. If REV flag is "true", the range is set in a reversed manner (line 13). Then, PAIs of each port range rule in V are calculated by 32 port aggregation (lines 14∼15). If the range rule is (ANY, 3-94), the LRME for range (ANY, 3-31), (ANY, 32-63), (ANY, 64-94) need to be generated. To generate LRME for (ANY, 32-63), the function simply sets LRME to $2^{32} - 1$, which is $1111 \cdots 111$ in binary representation. When PAI is the start or the end of the PAI of the ranges, the function shifts $2^{32} - 1$ to generate LRMEs (lines 17∼20). For example, $1111 \cdots 1000$ and $0111 \cdots 1111$ will be generated for ranges (ANY, 3-31) and (ANY, 64-94), respectively. At the end of the process, the function adds PAI and LRME pair to the LRME list and returns the list.

A.3 Details of Dynamic ACL Management

In the following, we describe the details of the dynamic ACL management functionality, which consists of the dynamic insertion and dynamic deletion, respectively.

❶ **Dynamic insertion.** As described in section 4.3, PortCatcher's control plane module, Shadow ACL, stores the intermediate data (i.e., IP, port, and LRME entries) while processing and deploying static ACL rules. To insert a new ACL rule on the fly (i.e., dynamic insertion), the ACL handler first loads the shadow ACL to retrieve the information of the deployed IP and port entries, as shown in Figure 6. Then, suppose that the new rule matches an IP and protocol entry where the action is "Drop". In that case, the function skips the rule insertion (line 8 in Algorithm 3) since the new rule has been covered by an existing rule, which discards all packets regardless of port rules. Otherwise, the ACL handler updates the shadow ACL with the new rule, which can be divided into four cases according to the port ANY settings in the source and destination port rule pair.

(1) if the new ACL rule has both layer-4 ports as ANY (i.e., bothANY), the ACL handler inserts the IP and protocol rule into the IP table with the action as "Drop" (line 15). Subsequently, all the new IP and protocol rule-related port table data and corresponding LRME table data will be eliminated from Shadow ACL (lines 10∼14).
(2) if the source port of the new rule is ANY, it further checks if any range rules exist for the same IP and protocol rule and deletes matched intermediate data (line 18∼20). The ACL handler then processes and insert the new rule into port and LRME tables for data plane deployment (line 21∼23)
(3) and (4) the last two cases are similar to the second case but deal with different ANY settings in the port rule (line 24).

❷ **Dynamic deletion.** This is a process to remove the expired or redundant ACL rules in the data plane. As in the dynamic insertion, the ACL handler first loads the shadow ACL and then deletes the rule from the IP table if its port rule is bothANY (line 26). Otherwise, if the deletion request is for a rule with source port ANY, it also checks whether the IP and protocol have bothANY port rules in the IP table (line 28). If matched, the existing IP and protocol entry will be updated (i.e., deleted and reinserted) after re-processing the IP and protocol entry's coupling field and the port rule related intermediate data in port and LRME tables (lines 29-34). We note that this process is essential since an IP and protocol entry may have one more port rule. Therefore, the requested port rule should be excluded from the previous port rule set associated with the IP and protocol entry (line 40). Particularly, to exclude the requested

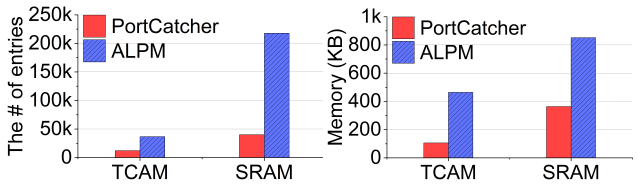
Algorithm 3: Dynamic ACL Management

1 **Inputs**: $R = (src_{IP}, dst_{IP}, protocol, src_{Port}, dst_{Port})$, Shadow ACL
2 HashMap IP <K, C> ← Shadow ACL
3 LinkedHashMap $Port_{srcANY}$<V, K>, $Port_{dstANY}$<V, K>, $Port_{noANY}$<V, K> ← Shadow ACL
4 LinkedHashMap $LRME_{srcANY}$<V, L>, $LRME_{dstANY}$<V, L>, $LRME_{noANY}$<V, L> ← Shadow ACL
5 Key ← $< R.src_{IP}, R.dst_{IP}, R.Protocol >$
6 Value ← $< R.src_{Port}, R.dst_{Port} >$
7 /*Dynamic insertion*/
8 **if** *IP[Key] == "Drop"* **then** continue;
9 **else if** *Value is bothANY* **then**
10 **Repeat lines 11-14 for Port_dstANY and Port_noANY**
11 **if** *Key* ∈ $Port_{srcANY}$ **then**
12 V ← $Port_{srcANY}$.getKeyOf(Key);
13 $Port_{srcANY}$.delete(V, Key);
14 $LRME_{srcANY}$.detete(V);
15 IP.add(Key, "Drop");
16 **else if** $R.src_{Port}$ *is ANY* **then**
17 V ← $Port_{srcANY}$.getKeyOf(Key);
18 **if** *V.size() > 0* **then**
19 $Port_{srcANY}$.delete(V, Key);
20 $LRME_{srcANY}$.detete(V);
21 V ← V ∪ Value;
22 $Port_{srcANY}$.add(V, Key);
23 $LRME_{srcANY}$[V] ← LRME_generation(V);
24 **Repeat lines 16-23 for dstANY and noANY**
25 /*Dynamic deletion*/
26 **if** *Value is bothANY* **then** IP.del(Key);
27 **else if** $R.src_{Port}$ *is ANY* **then**
28 **if** *IP[Key] == "Drop"* **then**
29 IP.del(Key);
30 V ← Subtract([0,2^{16}-1], Value);
31 $Port_{srcANY}$.add(V, Key);
32 REV_flag ← Port_Reverse(V);
33 $LRME_{srcANY}$[V] ← LRME_generation(V, REV_flag);
34 IP.add(K, LRM-ID, REV_flag);
35 **else if** *Key* ∈ $Port_{srcANY}$ **then**
36 IP.del(Key);
37 V ← $Port_{srcANY}$.getKeyOf(Key);
38 $Port_{srcANY}$.delete(V, Key);
39 LRME_deletion(V);
40 V ← Subtract(V, Value);
41 $Port_{srcANY}$.add(V, Key);
42 REV_flag ← Port_Reverse(V);
43 $LRME_{srcANY}$[V] ← LRME_generation(V, REV_flag);
44 IP.add(K, LRM-ID, REV_flag);
45 **Repeat lines 27-44 for dstANY and noANY**

port rule, ACL handler subtracts the requested port from the previous port rule set associated with the IP and protocol rule (line 30) and updates the port and LRME tables accordingly (line 31~33). A similar process will be repeated for the case of srcANY, dstANY, and noANY port rule deletion requests (lines 27-44).

A.4 Scalability with Attack Traffic-driven ACL

Figure 13(a) and (b) illustrate the number of the required entries and the consumed memory (KB) for the CIC DoS 2017 dataset [35]. In (a), and since PortCatcher stores only distinct IP and protocol entries in TCAM, it (at 12K) achieves better performance than ALPM (at 36K). In terms of the number of SRAM entries, PortCatcher requires 40K entries with our optimized design, while ALPM requires 217K entries using the range-to-prefix approach. Accordingly, PortCatcher



(a) The # of entries (b) Memory (KB)

Figure 13: The performance of PortCatcher and ALPM in terms of the scalability for the realistic CIC DoS 2017 dataset. (a) and (b) depict the number of required entries and the consumed memory respectively.

consumes 106 KB and 363 KB in TCAM and SRAM, respectively, whereas ALPM consumes 464 KB and 851 KB, as illustrated in (b). In the control plane, PortCatcher requires 470 KB and ALPM requires 4.71 GB for the trie data structure. Consequently, we confirmed that PortCatcher could mitigate the malicious traffic with less TCAM and SRAM space than ALPM for the same dataset.

A.5 Data Plane Implementation

We built a prototype of PortCatcher in a programmable switch [58]. The data plane modules are implemented in P4 language [22] using Barefoot SDE 9.0.0 [58]. In total, 338 lines of code (LoC) are added to the baseline switch, where 268 LoC are added for defining the match-action functions and 70 LoC are added for the function execution. The control plane module was implemented in Bfrt Python [31]. Port aggregation indexing is realized using bit slicing. The data communication between stages was implemented using user-defined metadata.

The code snippet below shows the data plane implementation details, including the match-action table definitions and the algorithm. The IP table performs a ternary matching of three tuples of a packet, namely *src-IP*, *dst-IP*, and *Proto* (lines 3~7). If matched, two predefined actions may be performed: **drop** and **get_coupling_info()** (line 8). The action is assigned for each TCAM entry in the ACL rule encoding stage. If a packet hits an entry that has `Both ANY` ports, the action field is **drop** and the action is performed immediately. The drop action is realized by setting `ig_dprsr_md.drop_ctl` as 1. Otherwise, if the packet hits non-`bothANY` rules, the entry's action is **get_coupling_info()**, thus the coupling field, user-defined data LRM-ID[] and REV[], are retrieved from the SRAM (line 12). These user-defined data is pre-set in reserved SRAM while setting the IP table entries. To read these data in packet processing pipeline, all data is recorded into the user metadata of the data plane (line 13~18).

In SRAM, three hash tables are reserved to perform the port matching for each of the non-`Both ANY` rules (lines 22~31). PortCatcher uses LRM-ID and PAI pair as a table lookup key, where the LRM-ID is retrieved from the user metadata and the two PAIs for port pair are the 11-bit MSB of the packet's port numbers (lines 23~27). If the ports of the packet hit a table entry, the **get_LRME()** action is performed to store the corresponding LRMEs into user metadata (line 34~35). The complete ACL matching flow is demonstrated in lines 39~56.

PortCatcher 's logic can be implemented in network hardware that is equipped with TCAM and SRAM. FPGA-based network

cards (NetFPGA) [6, 7] are the most relevant hardware in this case. PortCatcher 's major computation (LRM generation) occurs in the switch's control plane with a general-purpose CPU, which is available in NetFPGA. PortCatcher's data plane logic is also implemented under strict constraints to fit a switch ASIC's pipeline design, whereas NetFPGA's logic design is more flexible.

```
//STAGE-1: TCAM IP-PROTO-MATCHING & ACTIONS
table ip_proto {
    key = {
        hdr.ipv4.src_addr : ternary;
        hdr.ipv4.dst_addr : ternary;
        hdr.ipv4.protocol : ternary;
    }
    actions = { //Drop or get_coupling_info()}
}

//Actions
get_coupling_info(bit<16> LRM-ID[], bit<1> REV[]){
    meta.lrm_id_src_any = LRM-ID[0];
    meta.lrm_id_dst_any = LRM-ID[1];
    meta.lrm_id_no_any = LRM-ID[2];
    meta.rev_src_any = REV[0];
    meta.rev_dst_any = REV[1];
    meta.rev_no_any = REV[2];
}

//STAGE-2: SRAM PORT-MATCHING & ACTIONS
table port_no_any {
    key = {
        meta.lrm_id_no_any :  exact;
        meta.l4_lookup.word_1[15:5] : exact;
        meta.l4_lookup.word_2[15:5] : exact;
    }
    actions = { get_LRME();}
}
table port_src_any {...}
table port_dst_any {...}
//Action
get_LRME(bit<32> src_LRME, bit<32> dst_LRME) {
    meta.srcLRME = src_LRME;
    meta.dstLRME = dst_LRME;
}

//PortCatcher: Complete ACL matching flow
if(ip_proto.apply().hit){
    if(ig_dprsr_md.drop_ctl != 1)
    //not a Both Any port rule
        if(port_no_any.apply().hit){
            srcLRME_port_matching.apply();
            dstLRME_port_matching.apply();
            if(meta.srcLRME_port_matched
                && meta.dstLRME_port_matched
                && meta.rev_src_any == 1){
                drop();
            }else if(meta.rev_src_any == 0){
                drop();
            }
        }
        if(port_src_any.apply().hit){...}
        if(port_dst_any.apply().hit){...}
    }
}
```

Code Snippet : Match-action tables and logic.