HyperClassifier: Accurate, Extensible and Scalable Traffic Classification with Programmable Switches

Yichi Xu*, Guanyu Li*, Jiamin Cao*, Menghao Zhang*‡, Ying Liu*†, Mingwei Xu*†
*Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China
†Zhongguancun Laboratory, Beijing, China ‡Kuaishou Technology

ICC 2023 - IEEE International Conference on Communications | 978-1-5386-7462-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICC45041.2023.10279686

***Abstract*—Traffic classification provides substantial benefits for service differentiation, security policy enforcement, and traffic engineering. However, accurately classifying large volumes of network traffic using existing solutions is pretty challenging, as they are typically implemented on commodity servers with slow CPUs for packet processing. To address this, we leverage the opportunity provided by emerging programmable switches and propose HyperClassifier as a solution to achieve accurate, extensible, and scalable traffic classification. HyperClassifier designs an efficient classifying table with an effective flow expiration mechanism that enables lightweight packet inspection on resource-limited switches. We implement an open-source prototype of HyperClassifier on a hardware Tofino switch and conduct extensive evaluations. The results of our evaluation demonstrate that, compared to existing solutions, HyperClassifier can provide orders of magnitude higher classification throughput with comparable classification accuracy.**

I. Introduction

Traffic classification plays a crucial role in a variety of networking applications, such as network intrusion detection systems (NIDS) [1], quality of service (QoS) [2] and network management [3]. The goal of traffic classification is to identify the category (e.g., application protocol) that each flow belongs to, according to which operators are able to manage the network more efficiently and appropriately. For instance, administrators can selectively install only necessary rules for Snort [1] based on the results of traffic classification to improve the efficiency of NIDS.

Currently, increasingly rich network applications are putting higher demands on traffic classification systems (TCS). First of all, TCS should be *accurate* enough to identify the application protocol for each flow. Otherwise, the incorrect identification information may lead to unexpected results in subsequent applications, e.g., attacks bypassing NIDS. Second, TCS should be *extensible* to accommodate diverse emerging application protocols. So users can easily extend TCS on-demand to support and recognize application protocols under different scenarios. Finally, as the bandwidth and size of the network increase rapidly, TCS, which is usually deployed at the gateway of the target network, should also be *scalable* to process the massive amount of network traffic efficiently.

There have been many traffic classification methods, all of which fall into one of four categories, as shown in Table I. First, traditional port-based classification methods [4] classify traffic into different protocols by the port number. They are easy to be integrated in hardware and thus have high scalability. However, since they rely on the port numbers specified by Internet Assigned Numbers Authority (IANA) [5], they can only identify well-known applications or protocols, and the accuracy drops drastically for applications (especially P2P applications) using dynamic ports [6], which are becoming increasingly popular. Second, Machine learning (ML)-based methods [7]–[10] achieve high accuracy by extracting flow or packet features from raw network traffic and using well-trained ML model to classify flows. However, the classification results derived from ML-based methods have poor interpretability, which prevents us from improving the model and extending the well-trained model to other application protocols. Third, DPI-based methods [11]–[13] further resolve the extensibility issue in ML-based methods by inspecting packet payload and comparing it with application signatures in the database. However, they have significant performance issues because each byte of the payload has to be inspected and compared with a large set of application patterns. Finally, although some efforts [14]–[16] propose lightweight packet inspection (LPI) mechanisms that inspect limited portion of packet payload to improve the performance of DPI-based methods, such software-based solutions still have inherent limitations on packet processing capability and cannot sustain large-volume network bandwidth nowadays. In summary, there remains a gap between the requirements for traffic classification and existing methods.

TABLE I: Comparison of existing methods and our goal for HyperClassifier.

Methods	Accuracy	Extensibility	Scalability
Port-based [4]	○	●	●
Machine learning [7]–[10]	●	○	○
DPI [11]–[13]	●	●	○
LPI [14]–[16]	●	●	◐
HyperClassifier	●	●	●

The emergence of programmable switches provides an unprecedented opportunity to bridge this gap. On one hand, one single programmable switch can easily process multi-Tbps traffic at line rate, which has several orders of magnitude higher throughput than highly-optimized servers. On the other hand, programmable switches allow customizing packet processing with domain-specific languages (e.g., P4 [17]), which enables us to implement user-defined traffic classification logic. The high throughput and high flexibility of programmable switches make it possible to design an accurate, extensible and scalable traffic classifier.

However, implementing traffic classification logic in the programmable switch is a non-trivial effort. A naive method is to maintain a large set (e.g., *O(1K)*) of classification rules and then translate them into match-action table entries, which can easily exhaust the scarce memory resources in programmable switches. Besides, since the flow number may be quite large and it is too expensive to maintain per-flow states, we should design appropriate mechanisms to identify uncorrelated packets and clear inactive flows to avoid occupying too much switch memory. However, such mechanisms are challenging to be implemented in the constrained programming model of programmable switches.

To address these problems, we propose HyperClassifier, an accurate, extensible and scalable traffic classification system driven by programmable switches. First, to handle the limited memory on programmable switches, HyperClassifier proposes a memory-efficient table structure to accommodate a large number of classification rules. Second, to make our system scalable to massive flows, HyperClassifier overcomes the limited programmability of programmable switches and designs an efficient flow expiration mechanism. Third, to avoid potential flow table flooding attacks, HyperClassifier designs a self-defense strategy by monitoring half-open connections to adapt to adversarial scenarios. We implement a prototype of HyperClassifier on an Intel Tofino switch [18]. Extensive evaluations show HyperClassifier (1) provides orders of magnitude improvement in throughput with equally detection accuracy; (2) presents high scalability with increasing classification rules and workloads; and (3) robustly filters out the malicious traffic with low overheads.

II. Background & Related Work

A. Libprotoident

Libprotoident [14] is a representative LPI-based traffic classification library, supporting over 200 application protocols. To alleviate both the privacy and performance concerns associated with DPI-based methods, libprotoident is designed no longer to inspect each byte in the payload of every packet. Instead, libprotoident only captures the first payload-bearing packet observed in each direction (i.e., key packets) for every flow and checks whether their port number, packet length and the first four bytes of payload match the rule of an application protocol. Compared with DPI-based traffic classification methods, libprotoident provides comparable classification accuracy with a much smaller memory footprint [14].

Though libprotoident is more lightweight and performant than DPI-based methods, its scalability is still limited by CPU-based implementation [14], [15], since CPUs are not specialized for high-speed packet processing. But considering that libprotoident only needs to extract necessary information from key packets and check whether it matches the rule of an application protocol, both of these tasks can be supported by the programmable parser and programmable match-action pipelines of programmable switches separately. Therefore, we seek to improve the scalability of libprotoident with the opportunities brought by programmable switches.

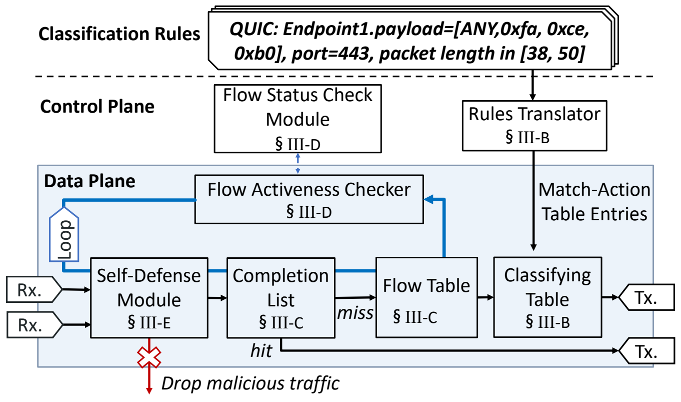

Classification Rules
QUIC: Endpoint1.payload=[ANY,0xfa, 0xce, 0xb0], port=443, packet length in [38, 50]
Control Plane
Flow Status Check Module § III-D
Rules Translator § III-B
Data Plane
Flow Activeness Checker § III-D
Match-Action Table Entries
Loop
Rx.
Rx.
Self-Defense Module § III-E
Completion List § III-C
miss
Flow Table § III-C
Classifying Table § III-B
Tx.
hit
Tx.
Drop malicious traffic

Fig. 1: HyperClassifier overview and workflow.

B. Hardware Acceleration

Many existing works have proposed to accelerate traffic classification using hardware. For example, Leira et al. [9] and Sun et al. [19] proposed to accelerate ML-based methods on GPU, which significantly speed up traffic classification using Single Instruction Multiple Data (SIMD) and multiple threads architecture. However, these methods face a performance bottleneck due to feature extraction and data exchange between main memory and video memory [19], [20]. There are also FPGA-based acceleration methods [21], [22], which accelerate machine learning algorithms on FPGA to achieve higher throughput. However, the hardware acceleration methods listed above mainly focus on accelerating machine learning algorithms but they do not resolve the problems of interpretability and extensibility associated with machine learning. Moreover, these accelerations still cannot reach Tbps level traffic classification.

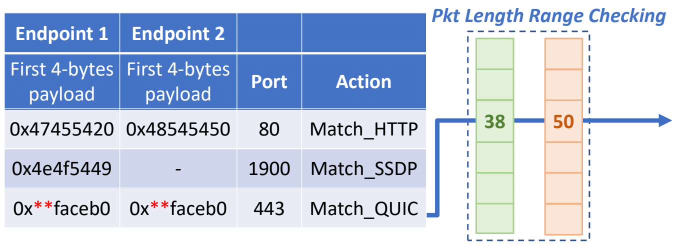
C. Opportunities by Programmable Switches

Programmable switching ASICs are specialized for high-speed packet processing, which can provide several orders of magnitude higher throughput than highly-optimized servers [23] and shows great potential for terabit TCS. With domain-specific languages like P4 [17], developers can easily deploy stateful packet processing with user-defined logics on the dedicated resource in programmable switches including match-action tables, stateful memory and ALUs. Besides, programs can also run collaboratively between the switching ASICs and the control plane switch CPUs, enabling advanced and flexible packet processing. As a result, these unique characteristics bring unprecedented opportunities to address the limitations of current traffic classification methods.

III. System Design

A. HyperClassifier Overview

HyperClassifier is designed to be an accurate, extensible and scalable traffic classifier based on the programmable switch. The switch can be deployed as a middlebox in the link, or as a dedicated application analyzing the traffic like a bypass tap. HyperClassifier can offer simple analysis to classification results in the data plane directly [24], [25], e.g., the amount of



Endpoint 1	Endpoint 2		
First 4-bytes payload	First 4-bytes payload	Port	Action
0x47455420	0x48545450	80	Match_HTTP
0x4e4f5449	-	1900	Match_SSDP
0x**faceb0	0x**faceb0	443	Match_QUIC

Fig. 2: HyperClassifier's classifying table.

each protocol in received traffic. Besides, operators can also deploy more advanced tools on CPUs to analyze classification results obtained from the switch.

The workflow of HyperClassifier is illustrated in Fig. 1. Operators need to provide a list of traffic classification rules (i.e., rules from libprotoident [14]) for application protocols. Our rules translator in the control plane will translate these classification rules into underlying match-action table entries for the traffic classifying table. To avoid misclassification caused by packets HyperClassifier does not care about, we design a flow table and completion list to let the system focus on only key packets that contribute to traffic classification in libprotoident's rules. To scale to a huge amount of flows with limited memory resource, the flow activeness checker recycles the flow entries that inactive flows occupied and interact with the flow status check module to dynamically update the completion list. Furthermore, a self-defense module is designed and integrated to make our system more robust when facing potential malicious attacks.

B. Classifying Table

To install all the classification rules completely, an intuitive way is to directly translate all the rules into match-action entries of the classifying table. However, simply translating classification rules into match-action table entries will exhaust the precious TCAM memory resource in switches. TCAM resource is very scarce in programmable switches and equally distributed to multiple stages in each pipeline. The classification rules usually require that the first 4-bytes payload, port number, and packet length of key packets should match a given pattern value. For example, payload matching requires ternary matching because some bytes of payload may be ignored by rules. Furthermore, range matching is required by packet length matching, but the TCAM resource consumed by range matching is at least four times than that required by ternary matching, which further exacerbates the TCAM resource shortage problem.

To address the problem, we convert the range matching of packet length to range checking instead of using TCAM resource. Fig. 2 illustrates our design of classifying table. Classifying table only takes payload and port as matching fields, and no longer matches packet length. Accordingly, packet length range matching is converted to range checking. We set two arrays to represent the upper and lower bound of the range and the packet length is compared if it meets this range. The traffic classification is successful only when the table matching and range checking are both passed. In

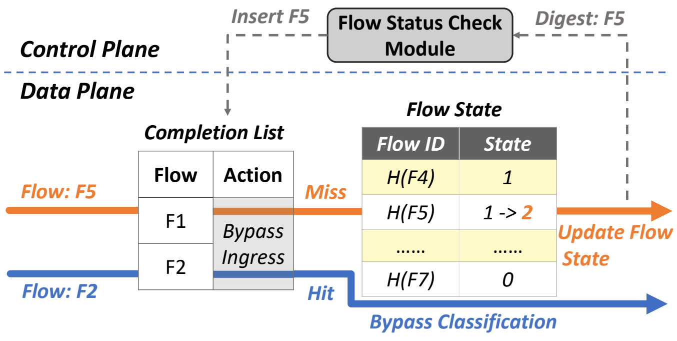


Fig. 3: The key packets tracking design of HyperClassifier.

this way, we reduce the requirements for TCAM resource significantly.

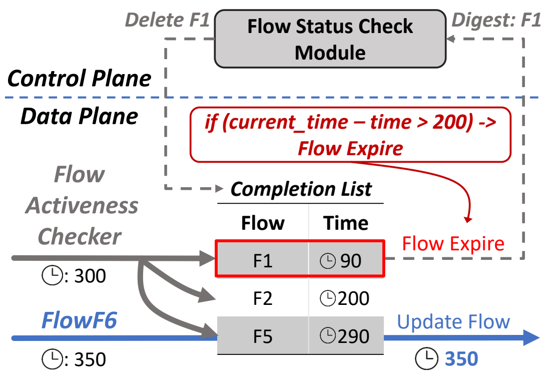
C. The Key Packets Tracking

To accurately classify flows based on rules, HyperClassifier should focus on only key packets that indeed contribute to traffic classification accuracy. Libprotoident requires that only the first payload-bearing packet sent from each direction of every flow contribute to the traffic classification and we call them key packets. Feeding other packets instead of key packets into the classifying table may lead to the flow being misclassified to an incorrect application protocol, which will then induce operators to configure wrong strategies.

To resolve the problem of key packet tracking, we maintain a state for each flow to indicate whether its key packets are both observed. The flow state has 3 values: 0, 1, 2. As shown in Fig. 3, all the states are initialized as 0, which means no key packet is observed in any direction and is ready for accepting the key packet of incoming direction. The state will transition from 0 to 1 when the key packet of incoming direction is first seen so any subsequent packet of this direction will be ignored. And we also need to temporarily buffer the key packet of incoming direction until the key packet of outcoming direction is seen. We use a table called flow table to buffer key packet and flow state, which contains first 4 bytes-payload, port number, packet length and flow state. Upon receiving the outcoming direction's key packet, HyperClassifier will transition the flow state from 1 to 2, which represents it is ready for traffic classification. Then our system loads information of incoming direction's key packet and feeds information of both key packets to the classifying table for matching. Similarly, any subsequent packet of the outcoming direction will be ignored.

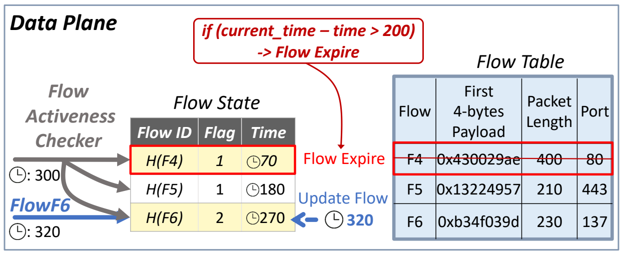
The memory resource in programmable data plane is limited and equally distributed to each stage in pipelines so HyperClassifier cannot store states of all flows. As a consequence, we employ the hash table as the table structure of flow table and each flow state is indexed by the hash value of flow's 5-tuple (i.e., srcIP, dstIP, srcPort, dstPort, protocol). The hash collision occurs when two different flows obtain the same hash value as the index, which will reduce the accuracy of our system. For example, flow A and flow B are mapped to the same position in flow table. Flow A's packet arrives first and sets the flow state to 2, then flow B will be ignored because the flow state indexed by flow B is 2 which means flow B is considered to have completed traffic classification. To address this issue, we design a completion list in the data plane, which uses match-action table to record the flows that have completed traffic classification (i.e., flow's state is 2), so HyperClassifier can distinguish which flow has completed classification. Any packet entering in the data plane will be firstly checked whether it is in the completion list, and will bypass the flow state transition and classifying table matching when it matches the completion list. By distinguishing flows that have completed traffic classification, these flows will not contend for the flow table. The flow status check module will insert the flow that has completed traffic classification (i.e., transition) into the completion list as soon as possible, in order to make room for new flows in flow table. Due to the hardware constraints, it cannot directly add or remove entries of match-action table in data plane. So we utilize a hardware mechanism in the switch called "digest" to send only 5-tuple to control plane and let the control plane issue the corresponding flow entry into the complete list.

D. Flow Expiration Mechanism



Flow	Time
F1	🕒90
F2	🕒200
F5	🕒290

(a) Expiration for completed classification flow.



Flow ID	Flag	Time
H(F4)	1	🕒70
H(F5)	1	🕒180
H(F6)	2	🕒270

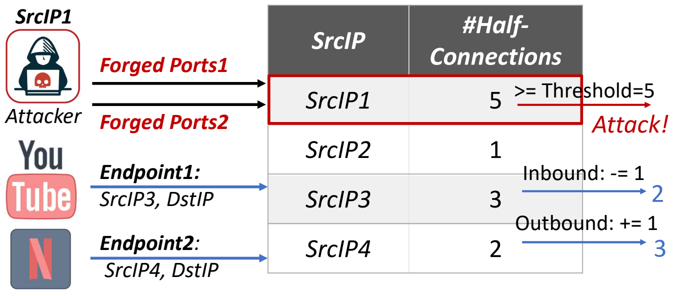
Flow	First 4-bytes Payload	Packet Length	Port
F4	0x430029ae	400	80
F5	0x13224957	210	443
F6	0xb34f039d	230	137

(b) Expiration for incomplete classification flow.

Fig. 4: Flow expiration mechanism.

To scale to the huge amount of flows and traffic size with limited memory resources, an intuitive way is to scan the flow table and the completion list periodically to clear the inactive flows. Nevertheless, scanning and updating a large number of entries via control plane will be inevitably time-consuming and make the system less responsive [26].

To resolve this problem, instead of running in control plane, we design a flow activeness checker, which scans and clear inactive flows in data plane periodically. We correlate each flow in the flow table and the completion list with a timestamp to represent a flow's latest active time. And each packet of a flow will automatically update the timestamp in data plane. To scan all entries in the completion list and flow table efficiently, the flow activeness checker periodically generates packets from the hardware internally, which are generated in the data plane within nanoseconds, and an entry is scanned by a packet. A flow is considered inactive when the flow activeness checker found its inactive time has exceeded the timeout threshold. The inactive time can be calculated by subtracting the latest active time from the current time. Fig. 4 illustrated how to recycle flows in the completion list and the flow table. The flow activeness checker informs the flow status check module in control plane to remove it from completion list and reset the timestamp. For inactive flows in the flow table, the flow activeness checker will reset the flow's timestamp and flow state so that any new flow can store its flow state in flow table. Besides, the checker will also recycle the memory occupied by key packets. With designs above, the flow expiration process can be executed entirely in data plane.



SrcIP	#Half-Connections
SrcIP1	5
SrcIP2	1
SrcIP3	3
SrcIP4	2

Fig. 5: Half-connection table for self-defense.

E. Self-Defense Strategy

To mitigate potential malicious attacks, HyperClassifier requires a robust self-defense strategy. As detailed in § III-C, when HyperClassifier receives the key packet from the incoming direction, it temporarily buffers the packet. However, one potential issue arises with respect to the security of the flow table: upon the arrival of the key packet at the data plane, HyperClassifier cannot distinguish whether the flow is legitimate or malicious. As a result, an attacker could generate numerous packets belonging to different flows to flood the flow table, resulting in no available space for legitimate flows. This type of attack is known as flow table flooding.

To solve this problem, we can easily ignore unestablished connections for TCP flows as SYN and SYN+ACK packets carry no payload. However, for UDP flows, the attacker may forge the source IP address or port number of the key packet to trick HyperClassifier into treating them as new flows and buffering them. One intuitive approach is to forward the first UDP key packet to the control plane and perform traffic classification after receiving the key packet from the outgoing direction. However, the bandwidth between the data plane and control plane constrains the performance of this method. Recent research efforts, NetHCF [27], [28] have tried to filter spoofed IP traffic with programmable switches, which are easy to be integrated into our system to filter spoofed IP flows. However, an attacker can still launch flooding attacks for flow table by generating key packets with forged port number and real IP address. To mitigate this issue, HyperClassifier employs a half-open connection table that records the number of half-open connections per IP address, where ”half-open” refers to a state in which the key packet from the incoming direction has been seen and buffered, but another key packet has not yet been received. Note that the concept of connection is also applied to UDP flows. As shown in Fig. 5, we increment the count of half-open connections by one upon seeing the first key packet and decrement it by one when the second key packet arrives. When the count exceeds the threshold, HyperClassifier identifies the flow table flooding attack, adds the IP address to a blacklist, and terminates its traffic classification for that IP address.

IV. Implementation and Evaluation

A. Implementation

We implement a prototype of HyperClassifier, including all data plane and control plane features described in § III. Our code is publicly available here [29]. The data plane part is implemented in ∼1K lines of P4 [17] code for the Intel Tofino switch [18]. The classifying table will take first 4 bytes-payload, port number and packet length of key packet as match fields. Its actions contain match success and match fails. We set the size of flow table as 65536. We use periodic timer pktgen in Tofino to implement flow activeness checker, which can periodically generate packets to scan inactive flows in data plane.

The control plane part is written in ∼600 lines of Python code. It is responsible for converting the classification rules into match-action table entries in Classfying table, enabling the flow activeness checker, and dynamically update the completion list. Besides, we implement a simple traffic classification results analyzer in the backend server, which extracts the classification result set by HyperClassifier in packet header.

B. Experimental Setup

Our testbed is composed of one 3.3 Tbp/s Intel Tofino switch (Wedge 100BF-32X) and two Dell R740 servers. All servers are directly connected to the switch and both servers are equipped with Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GH and 64 GB memory and connected to the switch via 40 GbE Intel XL710 NICs. The classification rules in our experiments are provided by libprotoident [30].

Traffic workloads. We use the packet trace collected from a campus network [31], which is a publicly available traffic trace and contains 18 applications' traffic of UDP or TCP. We replay the traces using DPDK-pktgen [32] on the packet generator node.

Baseline. We use libprotoident [30] as our baseline in our experiments. It's deployed in a Dell R740 server with a 40 GbE Intel XL710 NIC to connect to packet generator node.

TABLE II: Evaluated accuracy on different applications.

	Application	Accuracy		Application	Accuracy
TCP	HTTP	99.47%	UDP	USP_UDP	100%
	Facebook	100%		mDNS	100%
	HTTPS	97.57%		NTP	98.9%
	WhatsAPP	100%		DNS	99.76%
	SIP	100%		STUN_UDP	100%
	IMAPS	100%		DropBox	100%
	STUN_TCP	100%		QUIC	99.43%
	XMPP	100%		SSDP	100%
				NetBIOS_UDP	100%
				ISAKMP	100%

TABLE III: Switch resource utilization.

	Computational			Memory	
Resource	HashBits	VLIWs	Gateways	SRAM	TCAM
Usage	11.34%	8.33%	7.81%	31.06%	7.28%

We follow the recommended configuration for the baseline traffic classifier to get the best performance.

C. Overall Effectiveness

Classification accuracy. To determine whether HyperClassifier can correctly perform traffic classification based on libprotoident's [14] rules, we examine the results that HyperClassifier generates against the same trace and calculate the classification accuracy by baseline (libprotoident). We replay the trace for 60 seconds and collect the results generated from HyperClassifier and baseline. Table II shows that HyperClassifier is capable of discovering nearly all samples and achieving high traffic classification accuracy compared to baseline (libprotoident is set as ground truth, so it always has 100% accuracy). The performance of HyperClassifier may not achieve perfect accuracy in certain applications due to the presence of a subset of rules that necessitate more intricate comparisons beyond the capabilities of match-action tables.

Resource overhead. To evaluate the resource consumption of HyperClassifier, we measure the resource usage of our switch based on compiler's log. Table III shows the overall hardware resource utilization of switch. As we can see, HyperClassifier occupies less than 32% SRAM and 8% TCAM to accommodate libprotoident's classification rules. The TCAM usage is far less than SRAM because we convert most TCAM-consuming range matching for packet length to exact, so the result shows that our classifying table is high memory efficient.

D. System Performance

Scalability. To demonstrate the scalability of our system, we measure the classification accuracy under varying workloads. For each workload, we replay the trace at different speeds and collect the results obtained by both our system and baseline. As shown in Fig. 6, the accuracy of libprotoident drops drastically because CPU cannot afford high-speed packet processing and ignores classification for a lot of flows. In contrast, HyperClassifier can achieve almost same accuracy under varying workloads and find all potential samples. Note that 40Gbps is not the upper limit of HyperClassifier, and this is limited by the speed of our available trace replay tool.

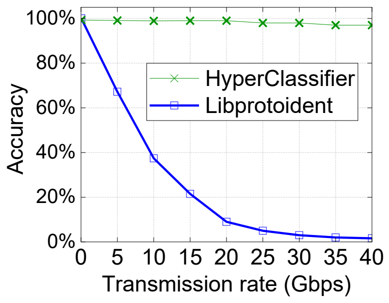


Fig. 6: Accuracy under different workloads.

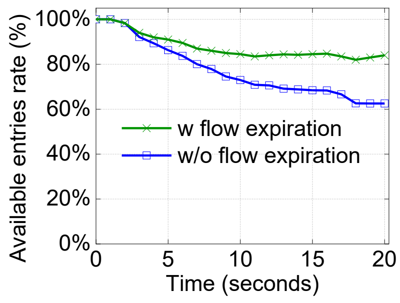


Fig. 7: Available table entries with flow expiration.

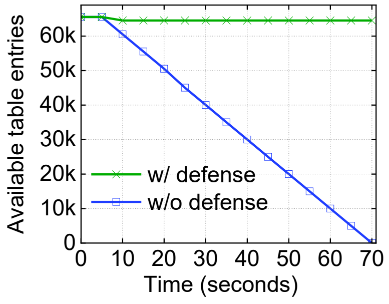


Fig. 8: Flow table flooding attacks.

Flow expiration mechanism. To evaluate how flow expiration mechanism clears inactive flows, we conduct a trace replay over a 20-second interval and measure the number of available table entries. To assess the impact of the flow expiration mechanism, we set the timeout for flow activeness checking to 2 seconds. Our experimental results, as illustrated in Fig. 7, demonstrate that in the absence of flow expiration, the available table entries steadily decrease, ultimately resulting in only 60% of the table being available. In contrast, the use of the flow expiration mechanism enables the timely removal of inactive flows, thereby maximizing the number of available table entries and reducing the occurrence of hash collisions.

E. System Robustness

The attacker could generate a large number of flows to flood the flow table, which can accommodate 65k active flows. HyperClassifier employs half-open connection table for self-defense, which limits the maximum number of connections per endpoint could establish and let flow activeness checker periodically remove the inactive connection in case of false positives. We use DPDK-pktgen to generate a large number of fake key packets and send them to switch. The result is shown in Fig. 8, without self-defense, the available space of flow table decreases quickly and eventually is fully occupied by malicious traffic so that any normal flow cannot be classified. Self-defense strategy could effectively limit the amount of entries that a flow can occupy.

V. Conclusion

In this paper, we propose HyperClassifier , an accurate, extensible and scalable traffic classifier driven by programmable switches. We design a set of techniques and optimizations: with an efficient classifying table, our system can accommodate a large number of classification rules; by tracking key packets, our system can enable accurate, lightweight packet inspection; with an effective flow expiration mechanism, our system is scalable to massive flows. We implement an open-source prototype of HyperClassifier on Tofino switch [18]. Our evaluation shows HyperClassifier can accurately, effectively and safely conduct traffic classification with low overheads.

Acknowledgments

We would like to thank anonymous ICC reviewers for their valuable comments. This work is supported in part by National Key Research and Development Program of China (No. 2018YFB1800405), National Natural Science Foundation of China (No. 61772307), Beijing Postdoctoral Research Foundation (2022-ZZ-078), and China Postdoctoral Science Foundation (2022M720202). Ying Liu is the corresponding author.

References

[1] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[2] Elie F Kfoury, Jorge Crichigno, and Elias Bou-Harb. Offloading media traffic to programmable data plane switches. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2020.

[3] Andrea Bianco, Paolo Giaccone, Seyedaidin Kelki, Nicolas Mejia Campos, Stefano Traverso, and Tianzhu Zhang. On-the-fly traffic classification and control with a stateful sdn approach. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[4] Patrick Schneider. Tcp/ip traffic classification based on port numbers. *Division Of Applied Sciences, Cambridge, MA*, 2138(5):1–6, 1996.

[5] IANA Port Numbers. Internet assigned numbers authority (iana), 2019.

[6] Thomas Karagiannis, Andre Broido, Michalis Faloutsos, and KC Claffy. Transport layer identification of p2p traffic. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 121–134, 2004.

[7] Riyad Alshammari and A Nur Zincir-Heywood. Machine learning based encrypted traffic classification: Identifying ssh and skype. In *2009 IEEE symposium on computational intelligence for security and defense applications*, pages 1–8. IEEE, 2009.

[8] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *ACM SIGCOMM Computer Communication Review*, 36(2):23–26, 2006.

[9] R Leira, P Gomez, I Gonzalez, and JE Lopez de Vergara. Multimedia flow classification at 10 gbps using acceleration techniques on commodity hardware. In *2013 International Conference on Smart Communications in Network Technologies (SaCoNeT)*, volume 3, pages 1–5. IEEE, 2013.

[10] Lixuan Yang, Alessandro Finamore, Feng Jun, and Dario Rossi. Deep learning and zero-day traffic classification: Lessons learned from a commercial-grade dataset. *IEEE Transactions on Network and Service Management*, 18(4):4103–4118, 2021.

[11] Luca Deri, Maurizio Martinelli, Tomasz Bujlow, and Alfredo Cardigliano. ndpi: Open-source high-speed deep packet inspection. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 617–622. IEEE, 2014.

[12] Fulvio Risso, Mario Baldi, Olivier Morandi, Andrea Baldini, and Pere Monclus. Lightweight, payload-based traffic classification: An experimental evaluation. In *2008 IEEE International Conference on Communications*, pages 5869–5875. IEEE, 2008.

[13] Alessandro Finamore, Marco Mellia, Michela Meo, Maurizio M Munafo, Politecnico Di Torino, and Dario Rossi. Experiences of internet traffic monitoring with tstat. *IEEE Network*, 25(3):8–14, 2011.

[14] libprotoident. https://github.com/wanduow/libprotoident, 2022. [Online; accessed 25-oct-2022].

[15] Giuseppe Aceto, Alberto Dainotti, Walter De Donato, and Antonio Pescapé. Portload: taking the best of two worlds in traffic classification. In *2010 INFOCOM IEEE Conference on Computer Communications Workshops*, pages 1–5. IEEE, 2010.

[16] Stênio Fernandes, Rafael Antonello, Thiago Lacerda, Alysson Santos, Djamel Sadok, and Tord Westholm. Slimming down deep packet inspection systems. In *IEEE INFOCOM Workshops 2009*, pages 1–6. IEEE, 2009.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[18] Intel Tofino. P4-programmable ethernet switch asic that delivers better performance at lower power, 2020.

[19] Guanglu Sun, Xuhang Li, Xiangyu Hou, and Fei Lang. Gpu-accelerated support vector machines for traffic classification. *International Journal of Performability Engineering*, 14(5):1088, 2018.

[20] Jiasong Bai, Menghao Zhang, Guanyu Li, Chang Liu, Mingwei Xu, and Hongxin Hu. Fastfe: Accelerating ml-based traffic analysis with programmable switches. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, pages 1–7, 2020.

[21] Da Tong, Yun Rock Qu, and Viktor K Prasanna. Accelerating decision tree based traffic classification on fpga and multicore platforms. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3046–3059, 2017.

[22] Tristan Groleat, Matthieu Arzel, and Sandrine Vaton. Hardware acceleration of svm-based traffic classification on fpga. In *2012 8th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 443–449. IEEE, 2012.

[23] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. {NetChain}:{Scale-Free}{Sub-RTT} coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[24] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. Cocosketch: high-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 207–222, 2021.

[25] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[26] Guanyu Li, Menghao Zhang, Cheng Guo, Han Bao, Mingwei Xu, Hongxin Hu, and Fenghua Li. {IMap}: Fast and scalable {In-Network} scanning with programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 667–681, 2022.

[27] Guanyu Li, Menghao Zhang, Chang Liu, Xiao Kong, Ang Chen, Guofei Gu, and Haixin Duan. Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering. In *2019 IEEE 27th international conference on network protocols (ICNP)*, pages 1–12. IEEE, 2019.

[28] Menghao Zhang, Guanyu Li, Xiao Kong, Chang Liu, Mingwei Xu, Guofei Gu, and Jianping Wu. Nethcf: Filtering spoofed ip traffic with programmable switches. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[29] HyperClassifier. https://github.com/hyperclassifier/hyperclassifier, 2022.

[30] Shane Alcock and Richard Nelson. Libprotoident: traffic classification using lightweight packet inspection. Technical report, Technical report, University of Waikato, 2012.

[31] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying iot devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2018.

[32] pktgen dpdk. https://github.com/pktgen/pktgen-dpdk, 2011.