AWEsome-Cache: dependency-free rule-caching for arbitrary wildcard patterns in TCAM

Zeyu Luan[1,2], Qing Li [2*], Zutao Zhang[3], Yong Jiang[1,2], Meng Chen[4], Yu Wang[2], Kejun Li[2]

[1]*Tsinghua Shenzhen International Graduate School, Tsinghua University, Shenzhen, China*
[2]*Department of Mathematics and Theories, Peng Cheng Laboratory, Shenzhen, China*
[3]*Software College of Northeastern University, Northeastern University, Shenyang, China*
[4]*Department of Big Data and Software Engineering, Chongqing University, Chongqing, China*

2023 IEEE 31st International Conference on Network Protocols (ICNP) | 979-8-3503-0322-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICNP59255.2023.10355586

***Abstract*—Ternary Content Addressable Memory (TCAM) is a specialized high-speed memory that enables fast parallel lookups for both exact-match rules and wildcard-match rules. TCAM has become a standard hardware component in Software-Defined Networking (SDN) switches to implement flow tables for packet classification. However, limited TCAM storage capacity poses a significant scalability challenge for SDN to enforce fine-grained policy-based forwarding. To this end, TCAM-based rule-caching systems are proposed by combining TCAM with Random Access Memory (RAM). Specifically, TCAM caches heavy-hitting rules to capture packets from hot flows, while RAM accommodates the complete ruleset for other cache-miss packets. However, previous rule-caching systems either failed to eliminate cross-rule dependencies or restricted their applications to prefix rules only. In this work, we propose AWEsome-Cache, a unifying framework to fundamentally eliminate cross-rule dependencies for wildcard rules with arbitrary matching patterns. The rationale behind AWEsome-Cache is to concretize a minimum number of wildcard bits in the best-match rule, thereby pruning its overlapping match fields with all direct dependent rules. AWEsome-Cache also develops replacement algorithms during TCAM updates to adapt to dynamic traffic locality. Experiments with prefix and non-prefix rules show that AWEsome-Cache outperforms baselines in achieving a comparable cache-hit rate but requiring 75.9% less TCAM occupancy.**

***Index Terms*—Rule Caching, TCAM, Software-Defined Networking**

I. Introduction

Software-Defined Networking (SDN) [1] enables network operators to customize fine-grained per-flow forwarding policies, wherein multi-field packet headers are used as the search key to match against forwarding rules in flow tables. Ternary Content Addressable Memory (TCAM) is a standard hardware component to implement flow tables in SDN-enabled switches [2]. TCAM supports parallel lookups at line rate for both exact-match rules and wildcard-match rules. However, due to expensive hardware costs and intensive power consumption, TCAM suffers from a limited storage capacity, ranging from thousands to tens of thousands of entries [3]. In large-scale networks, fine-grained forwarding policies populate millions of forwarding rules, far exceeding available table entries in TCAM [4]. Consequently, the broadening gap between limited TCAM capacity and ever-increasing forwarding rules raises a significant scalability challenge for SDN to implement fine-grained policy-based forwarding [5].

*Corresponding author: Qing Li (liq@pcl.ac.cn)

Inspired by traffic locality observed in real-world networks, TCAM-based rule-caching [6] is proposed as a promising solution to address the scalability challenge in SDN. Specifically, TCAM serves as the high-speed cache memory maintaining the most heavy-hitting rules for hot flows, while Random Access Memory (RAM) serves as the auxiliary memory maintaining the complete ruleset for other cache-miss flows. In contrast to TCAM, RAM has a cost-effective large storage capacity, but it requires CPU-involved software algorithms for packet classifications, resulting in relatively slow lookups [7]. Hence, rule-caching systems combine the characteristics of TCAM and RAM to enable fast parallel lookups across a large ruleset for SDN-enabled switches.

One of the most significant challenges in TCAM-based rule-caching systems is the issue of cross-rule dependencies [8]. In the multi-dimensional field space, the match fields of multiple wildcard rules can partially overlap; therefore, packets can hit multiple overlapping wildcard rules in the ruleset, and the one with the highest priority is identified as the best-match rule. Given the best-match rule for a specific hot flow, there are two types of dependent rules: direct and indirect. Direct dependent rules have overlapping match fields with the best-match rule, while indirect dependent rules overlap with the direct dependent rules, though they may not overlap with the best-match rule. The naive way to address the cross-rule dependency is caching the best-match rule and all dependent rules, both direct and indirect. These dependent rules occupy additional TCAM space but reduce TCAM utilization because not all dependent rules are as heavy-hitting as the best-match rule.

Many efforts have been devoted to addressing the cross-rule dependency in TCAM-based rule-caching systems [9]–[12]. CacheFlow [13] and its spin-offs [14]–[16] truncate the dependency chain by excluding indirect dependent rules, which saves part of TCAM entries but still requires caching direct dependent rules. In contrast, T-Cache [17] [18] fundamentally eliminates cross-rule dependencies by constructing a so-called *isolate rule*, which prunes the original match field of the best-match rule. However, T-Cache applies to prefix rules only where wildcard bits follow strictly behind exact bits.

In this work, we develop novel algorithms for constructing isolate rules and generalize their applications to any form of wildcard rules, making T-Cache a special case of our design. We propose AWEsome-Cache, a unifying framework to eliminate cross-rule dependencies for wildcard rules with arbitrary matching patterns, including prefix and non-prefix rules. To the best of our knowledge, AWEsome-Cache is the first work focusing on the cross-rule dependency for multi-field non-prefix rules. We design algorithms for constructing isolate rules under different circumstances, preserving as many wildcard bits from the best-match rule as possible to improve cache-hit rates. We also develop the replacement strategy for TCAM updates to adapt to dynamic traffic locality.

In summary, our contributions to this paper include:

- We propose AWEsome-Cache, a unifying framework to fundamentally eliminate cross-rule dependencies, which can be applied to wildcard rules with arbitrary wildcard patterns, including prefix and non-prefix rules.
- We develop algorithms for constructing dependency-free isolate rules by concretizing a minimum number of wildcard bits in the best-match rule and design the replacement strategy for TCAM updates.
- Extensive experiments on prefix and non-prefix rules demonstrate that AWEsome-Cache outperforms state-of-the-art baselines in achieving comparable TCAM utilization with 75.9% less TCAM occupancy.

The remainder of the paper is organized as follows. Section II and Section III present background and motivation, respectively. Section IV extends the definition of the XOR operation to identify dependency relationships. Section V and Section VI design algorithms for constructing isolate rules and refreshing TCAM, respectively. We evaluate AWEsome-Cache in Section VII and conclude the paper in Section VIII.

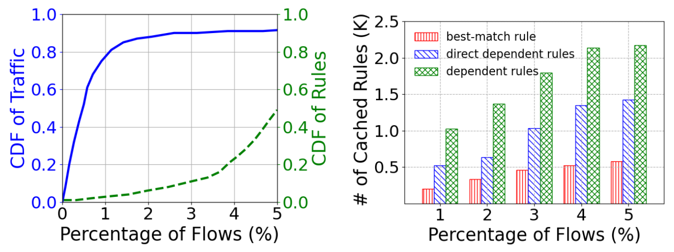
II. Background and Related Work

A. *Traffic Locality and Rule Caching*

While TCAM supports fast parallel lookup and flexible matching patterns, the shortage of its storage capacity can hardly accommodate an ever-increasing number of wildcard rules populated by fine-grained forwarding policies. Fortunately, studies have revealed a skewed traffic distribution in real-world networks, known as the traffic locality [19].

From a temporal perspective, a limited number of hot flows contribute to most network traffic. Statistics from the Equinix dataset in Fig. 1(a) show that 78.30% of traffic attributes to the top 1% flows [20]. From a spatial perspective, packets from hot flows are mostly captured by a limited number of heavy-hitting rules, which account for a small fraction of the complete ruleset. As shown in Fig. 1(b), all packets from the top 1% flows hit only 0.04% (332 out of 8.3K) forwarding rules in the ruleset.

The observation of the traffic locality inspires wildcard rule-caching systems with a mix use of TCAM and RAM, where a limited number of heavy-hitting rules for hot flows are cached



(a) top 1% flows contribute to 78.30% traffic but hit only 0.04% of ruleset (b) top 1% flows hit 198 best-match rules and 1022 dependent rules

Fig. 1. Statistics of traffic locality and cross-rule dependencies from Equinix

in TCAM for fast lookups, while the complete ruleset is stored in RAM for other cache-miss traffic.

However, the distribution of traffic locality is not static but constantly varies over time. A measurement from the Equinix dataset shows that the traffic percentage claimed by the initial 1% top flows fluctuates from 81% to 48% in an hour [20]. To sustain high TCAM utilization, efficient TCAM update is required to adapt to the dynamics of traffic locality.

B. *Cross-Rule Dependency*

Prioritized wildcard rules with overlapping match fields raise the issue of cross-rule dependency. Specifically, given a hot flow f and its best-match rule r_f in the ruleset R, high-priority rules overlapping with the best-match rule r_f in the field space constitute the set of direct dependent rules $D(r_f) \subseteq R$. Each direct dependent rule $r_i \in D(r_f)$ may also depend on other higher-priority rules in R, defined as the indirect dependent rules for the best-match rule r_f. Both direct and indirect dependent rules collectively constitute the set of dependent rules $P(r_f) \subseteq R$ for the best-match rule r_f. Their relationship can be denoted as $D(r_f) \subseteq P(r_f)$.

Given a hot flow f, its best-match rule r_f and all dependent rules $P(r_f)$ should be cached in TCAM to ensure semantic correctness. The issue of cross-rule dependencies not only occupies additional TCAM space but also complicates TCAM updates. An experiment from the Equinix dataset in Fig. 1(b) shows that the top 1% hot flows hit only 198 best-match rules but require caching additional 1,019 dependent rules in TCAM [20].

C. *Related Work*

Many efforts have been devoted to addressing the cross-rule dependency in TCAM-based rule-caching systems. CAB [10] [11] and CRAFT [12] convert wildcard rules into a set of non-overlapping sub-rules to limit the number of dependent rules, which causes severe ruleset expansion. CacheFlow [13] truncates the dependency chain and caches only direct dependent rules $D(r_f)$ rather than $P(r_f)$ in TCAM, alleviating cross-rule dependencies but still requiring additional TCAM space. In contrast, T-Cache [17] [18] fundamentally eliminates cross-rule dependencies by constructing isolate rules, which prunes the original match field of the best-match rule. However, T-

Cache applies to prefix rules only where wildcard bits follow strictly behind exact bits.

AWEsome-Cache generalizes the application of isolate rules to arbitrary wildcard patterns, including prefix and non-prefix rules. To the best of our knowledge, this is the first work focusing on addressing cross-rule dependencies for multi-field non-prefix rules.

III. Motivation and System Overview

This section introduces the concept of isolate rules in Section (III-A) and provides a motivating example in Section (III-B). Section (III-C) presents the system overview.

A. Isolate Rule

AWEsome-Cache follows the idea of constructing isolate rules to eliminate cross-rule dependencies in the TCAM-based rule-caching system. The isolate rule is a trade-off between two extremes: caching the exact rule for the hot flow, or caching the best-match rule along with all dependent rules. Compared with caching the exact rule, the isolate rule preserves part of the wildcard bits from the best-match rule, leading to larger match fields than the exact rule. On the other hand, compared with caching the best-match rule along with all dependent rules, the isolate rule eliminates cross-rule dependencies, saving TCAM space to cache more hot flows.

Isolate rules are beneficial to TCAM-based rule-caching systems in three-fold. First, it fundamentally eliminates cross-rule dependency, saving TCAM space originally reserved for dependent rules. Second, it improves TCAM utilization by preserving as many wildcard bits as possible from the best-match rules, leading to large match fields to capture more packets from future flows. Third, it simplifies TCAM updates because the isolate rule allows for free placement at any empty entry without the need to relocate existing rules in TCAM.

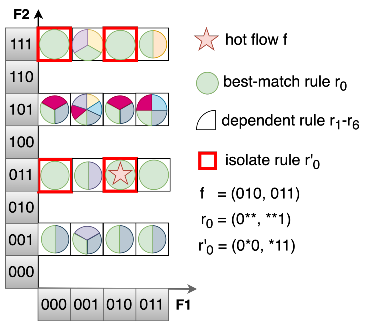
Given a hot flow f, the best-match rule $r_f \in R$, and each direct dependent rule $r_i \in D(r_f) \subseteq R$, the optimal isolate rule r'_f satisfies the following three criteria: (i) r'_f still matches f in the field space; (ii) r'_f is dependency-free with any direct dependent rule $r_i \in D(r_f)$; (iii) r'_f concretizes a minimum number of wildcard bits in r_f, and thus preserves the original match field of r_f as much as possible.

B. Motivating Example

Fig. 2 is a motivating example used to illustrate the rationale behind isolate rules. For the sake of simplicity, we assume forwarding rules from r_0 to r_6 with 2-dimensional match fields. Fig. 2(b) visualizes the 2-dimensional field space, where the exact rule for a hot flow occupies a unique point, and therefore caching the exact rule for the hot flow in TCAM involves no cross-rule dependency. For example, the exact rule for the hot flow f corresponds to a unique point at the position of $(010, 011)$, shown as a star mark in Fig. 2(b). However, caching this exact rule $r = (010, 011)$ alone for the hot flow $f = (010, 011)$ undermines TCAM utilization, as it captures packets only from this specific hot flow.

Flow & Rules	Match	Action	Priority
f	(010, 011)	-	-
r_0	(0**, **1)	Fwd (0)	0
r_1	(001, **1)	Fwd (1)	1
r_2	(001, 1*1)	Fwd (2)	2
r_3	(0*1, 101)	Fwd (3)	3
r_4	(*11, 111)	Fwd (4)	4
r_5	(0**, 101)	Fwd (5)	5
r_6	(0**, *01)	Fwd (6)	6

(a) example of hot flow f and wildcard rules in ruleset R



(b) visualization of cross-rule dependencies in 2-dimensional field space

Fig. 2. Illustration of cross-rule dependencies and the isolate rule in the field space. The exact rule $(010, 011)$ for the given flow f is marked as a star. The best-match rule $r_0 = (0**, **1)$ with four wildcard bits corresponds to 16 positions in green. Direct dependent rules $r_1 \sim r_6$ correspond to the overlapping positions in six different colors. For example, the position $(001, 101)$ is overlapped by the best-match rule r_0 and five direct dependent rules r_1, r_2, r_3, r_5, and r_6. The isolate rule $r'_0 = (0*0, *11)$ with two wildcard bits corresponding to four positions also captures flow f but does not overlap any direct dependent rules, thus eliminating the dependency between r_0 and $r_1 \sim r_6$.

On the other hand, the best-match rule $r_0 = (0**, **1)$ comprising four wildcard bits for the hot flow f occupies 16 discrete positions in the field space and incurs the problem of cross-rule dependency, as these discrete positions are co-occupied by direct dependent rules of r_0. As shown in Fig. 2(b), the best-match rule r_0 occupies 16 green points in the field space, as the four wildcard bits in $r_0 = (0**, **1)$ represent a combination of 16 exact rules. Note that other rules from r_1 to r_6 in the ruleset R are all direct dependent rules of r_0 because they share different positions with r_0 in the field space. For example, the point $(001, 101)$ is co-occupied by six rules, including r_0 and $\{r_1, r_2, r_3, r_5, r_6\}$.

Accordingly, all rules from r_1 to r_6 constitute a set of direct dependent rules $D(r_0)$ for the best-match rule r_0. Since their priorities are all higher than r_0, they should be also cached in TCAM along with r_0 to ensure semantic correctness. The existence of co-occupation on specific positions in the field space between the best-match rule r_f and direct dependent rules in $D(r_f)$ is the root cause of cross-rule dependencies.

To eliminate cross-rule dependencies, AWEsome-Cache constructs the so-called isolate rule r'_0 by reducing the original match fields of the best-match rule r_0. Fig. 2(b) represents the resulting isolate rule $r'_0 = (0*0, *11)$ with four red blocks, while exclusively occupying these four positions without co-occupancy with any direct dependent rule. Compared with the best-match rule $r_0 = (0**, **1)$, the isolate rule $r'_0 = (0*0, *11)$ is constructed by concretizing two carefully selected wildcard bits into binary bits. The isolate rule r'_0 concretizes a minimum number of wildcard bits to preserve the original match fields of r_0 as much as possible.

C. System Overview

Fig. 3 overviews the system design and presents the workflow of packet processing. An incoming packet first accesses

TCAM to search for the best-match rule among all cached rules. If it is a cache-hit packet, it will be processed according to the action specified in the best-match rule. Otherwise, the cache-miss packet will be punted to RAM to match against the complete ruleset using software lookup algorithms.

Once the best-match rule r_f for the hot flow f is identified in the complete ruleset R, the isolate rule r'_f will be constructed based on three inputs: the match field of the hot flow f, the best-match rule r_f in the ruleset, and the set of all direct dependent rules $D(r_f)$. Since the resulting isolate rule r'_f is dependency-free, it can be placed at any empty entry or replaced with a cold rule from TCAM.

IV. Extended Definition of XOR Operation

In the context of the ternary match in TCAM, we extend the definition of conventional XOR operation in Sec.IV-A, making it applicable to both exact bits and wildcard bits. Then, we apply the extended XOR operation to identify the best-match rule (Sec.IV-B), direct dependent rules (Sec.IV-C), and all candidate bit positions for concretization to eliminate cross-rule dependencies (Sec.IV-D).

A. XOR Operation with Wildcard Bits

In the context of the ternary match in TCAM, we extend the definition of conventional Exclusive OR (XOR) operation, making it applicable to exact bits ('1' or '0') and wildcard bits ('*'). XOR is a conventional logical bitwise operation determining whether two binary operands are identical or not. XOR operation returns true ('1') if the two input bits are matched, or false ('0') if mismatched. The extended XOR operation inputs two ternary bits and outputs a one-bit Boolean value. The truth table for all possible inputs is shown in Fig. 4(a).

For exact bits, the extended XOR operation inherits the truth table of the conventional XOR operation, which returns true ('1') if and only if two exact bits are mismatched; otherwise, it returns false ('0'). For wildcard bits, on the other hand, if either of the two input operands is a wildcard bit (i.e., both are wildcard bits, or one wildcard and one exact bit), the extended XOR operator returns false ('0'). The rationale behind the extended XOR operation is that the wildcard bit in the ternary match field is considered to match another input bit of any form, regardless of its value ('1', '0', or '*'). The extended XOR operator can also be applied to a pair of input bit strings X and Y beyond a pair of input bits, formulated as follows.

$$\begin{aligned}
& X[1:K] \oplus Y[1:K] \\
= \; & (X[1], ..., X[k], ..., X[K]) \oplus (Y[1], ..., Y[k], ..., Y[K]) \\
= \; & (X[1] \oplus Y[1], ..., X[k] \oplus Y[k], ..., X[K] \oplus Y[K]) \\
& \forall\ k \in [1, K] \qquad (1)
\end{aligned}$$

where X and Y are a pair of input bit strings, each consisting of K bits. If the output bit string is all-zeros, the two input bit strings match each other. If the output bit string contains

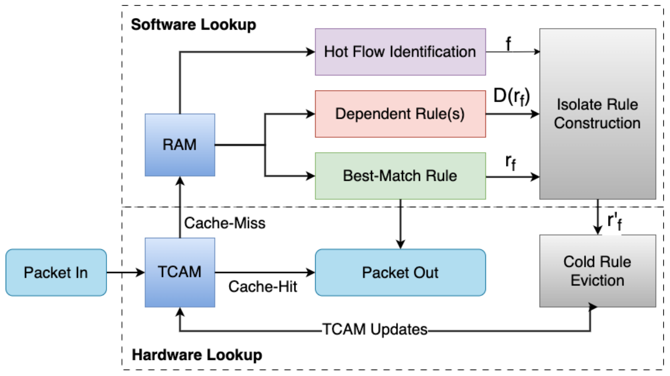
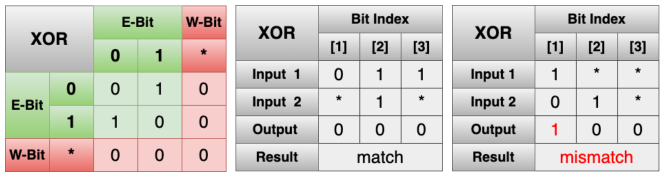


Fig. 3. System overview and packet classification pipeline



XOR		E-Bit		W-Bit
		0	1	*
E-Bit	0	0	1	0
	1	1	0	0
W-Bit	*	0	0	0

XOR	Bit Index		
	[1]	[2]	[3]
Input 1	0	1	1
Input 2	*	1	*
Output	0	0	0
Result	match		

XOR	Bit Index		
	[1]	[2]	[3]
Input 1	1	*	*
Input 2	0	1	*
Output	1	0	0
Result	mismatch		

(a) truth table for extended XOR operation (b) example of matched inputs (011) and $(*1*)$ (c) example of mismatched inputs $(1**)$ and $(01*)$

Fig. 4. Examples of extended XOR operation for ternary match fields

at least one non-zero bit(s), the two input bit strings mismatch each other. For example, Fig. 4(b) and Fig. 4(c) show two examples of matched ((011) and (*1*)) and mismatched ((1**) and (01*)) bit strings, respectively.

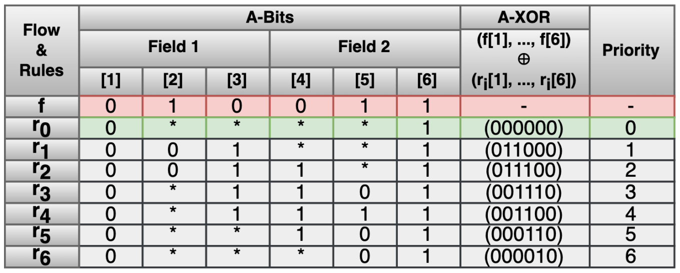
We can apply the extended XOR operation to specific bits to identify the best-match rule, direct dependent rules, and all possible concretization bit positions given a hot flow and the complete ruleset.

B. A-XOR to Identify Best-Match Rule

To identify the best-match rule for a hot flow, we apply the extended XOR operation to *all bits* of the given flow and each rule in the ruleset. We refer to *all bits* of the two input bit strings as A-Bits and define the extended XOR operation on A-Bits as A-XOR. We formulate A-XOR as follows.

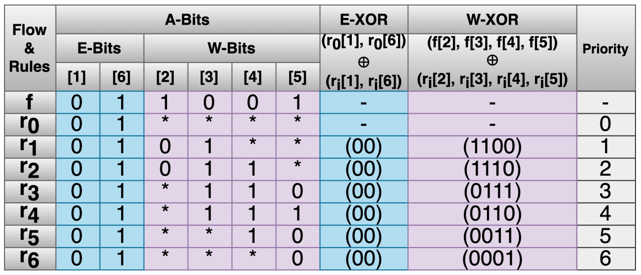
$$\begin{aligned}
& f[1:A] \oplus r[1:A] \\
= \; & (f[1], ..., f[k], ..., f[A]) \oplus (r[1], ..., r[k], ..., r[A]) \\
= \; & (f[1] \oplus r[1], ..., f[k] \oplus r[k], ..., f[A] \oplus r[A]) \\
& \forall\ r \in R, k \in [1, A] \qquad (2)
\end{aligned}$$

where f is a given hot flow, $r \in R$ is a forwarding rule in the ruleset R, and $k \in [1, A]$ is the index of A-Bits. If the output of the A-XOR operation is all-zeros, the result indicates that flow f matches rule r. Conversely, flow f mismatches rule r if the output of the A-XOR operation contains any non-zero bit(s). Note that, if flow f matches more than one rule in R, the one with the highest priority will be finalized as the best-match rule r_f.



Flow & Rules	A-Bits						A-XOR	Priority
	Field 1			Field 2			(f[1], ..., f[6]) ⊕ (r_i[1], ..., r_i[6])	
	[1]	[2]	[3]	[4]	[5]	[6]		
f	0	1	0	0	1	1	-	-
r_0	0	*	*	*	*	1	(000000)	0
r_1	0	0	1	*	*	1	(011000)	1
r_2	0	0	1	1	*	1	(011100)	2
r_3	0	*	1	1	0	1	(001110)	3
r_4	0	*	1	1	1	1	(001100)	4
r_5	0	*	*	1	0	1	(000110)	5
r_6	0	*	*	*	0	1	(000010)	6

Fig. 5. A-XOR operation identifies r_0 as the best-match rule for hot flow f



Flow & Rules	A-Bits						E-XOR	W-XOR	Priority
	E-Bits		W-Bits				(r_0[1], r_0[6]) ⊕ (r_i[1], r_i[6])	(f[2], f[3], f[4], f[5]) ⊕ (r_i[2], r_i[3], r_i[4], r_i[5])	
	[1]	[6]	[2]	[3]	[4]	[5]			
f	0	1	1	0	0	1	-	-	-
r_0	0	1	*	*	*	*	-	-	0
r_1	0	1	0	1	*	*	(00)	(1100)	1
r_2	0	1	0	1	1	*	(00)	(1110)	2
r_3	0	1	*	1	1	0	(00)	(0111)	3
r_4	0	1	*	1	1	1	(00)	(0110)	4
r_5	0	1	*	*	1	0	(00)	(0011)	5
r_6	0	1	*	*	*	0	(00)	(0001)	6

Fig. 6. E-XOR operation identifies direct dependent rules, and W-XOR operation identifies concretization positions to construct isolate rules

As shown in Fig.5, we perform the A-XOR operation to flow $f = (010, 011)$ and each rule $r \in R$. The result shows that only r_0 satisfies the all-zeros criterion, and r_0 is identified as the best-match rule for flow f. However, other rules in the ruleset (from r_1 to r_6) mismatch flow f, as they contain at least one non-zero bit in the output bit strings.

C. E-Operation to Identify Dependent Rules

We further categorize *all-bits* (A-Bits) of the best-match rule r_f into *exact bits* (E-Bits) and *wildcard bits* (W-Bits) according to the value of each bit. As shown in Fig. 6, in the best-match rule $r_0 = (0**, **1)$, $r_0[1]$ and $r_0[6]$ are E-bits while other bits, from $r_0[2]$ to $r_0[5]$, are W-Bits. Similarly, we define the extended XOR operation on E-Bits and W-Bits as E-XOR and W-XOR, respectively. We apply the E-XOR operation to the best-match rule r_f and each rule $r \in R - \{r_f\}$ in the ruleset to identify all direct dependent rules $D(r_f)$. We formalize the E-XOR operation on E-Bits as follows.

$$\begin{aligned} & r_f[1:E] \oplus r[1:E] \\ = \; & (r_f[1], ...r_f[k], ..., r_f[E]) \oplus (r[1], ..., r[k], ..., r[E]) \\ = \; & (r_f[1] \oplus r[1], ..., r_f[k] \oplus r[k], ..., r_f[E] \oplus r[E]) \\ & \forall \; r \in R - \{r_f\}, k \in [1, E] \end{aligned} \tag{3}$$

where $k \in [1, E]$ is the index of E-Bits. The rationale behind the definition of the E-XOR operation is that the cross-rule dependency arises between r_f and any rule $r \in R - \{r_f\}$ if and only if their exact bits are matched regardless of their wildcard bits.

As shown in Fig. 6, the results of the E-XOR operation between the best-match rule r_0 and the other rules (from r_1 to r_6) are all-zeros, indicating that they are all dependent rules of the best-match rule r_0, i.e., $D(r_0) = \{r_1, r_2, r_3, r_4, r_5, r_6\}$.

D. W-Operation to Identify Concretization Positions

To construct isolate rules, we apply the W-XOR operation to the hot flow f and each direct dependent rule $r_i \in D(r_f)$ to obtain all candidate bit positions in the best-match rule to be concretized as exact bits. We formalize the W-XOR operation on W-Bits as follows.

$$\begin{aligned} & f[1:W] \oplus r_i[1:W] \\ = \; & (f[1], ..., f[k], ..., f[W]) \oplus (r_i[1], r_i[k], ..., r_i[W]) \\ = \; & (f[1] \oplus r_i[1], ..., f[k] \oplus r_i[k], ..., f[W] \oplus r_i[W]) \\ & \forall \; r_i \in D(r_f), k \in [1, W] \end{aligned} \tag{4}$$

where f is a given hot flow, $r_i \in D(r_f)$ is one of its direct dependent rules, and $k \in [1, W]$ is the index of W-Bits.

For example, in Fig. 6, the result of W-XOR between f and r_6 is (0001), which means that the cross-rule dependency between r_0 and r_6 can be eliminated by concretizing the wildcard bit $r_0[5]$ into a binary value, i.e., $r'_0[5] = f[5] = 1$. Note that the result of W-XOR between f and a dependent rule $r_i \in D(r_f)$ may contain more than one non-zero bit. For example, the result of W-XOR between f and r_1 is (1100), which means that r_0 can eliminate its dependency from r_1 by concretizing either $r_0[2]$ or $r_0[3]$, or both.

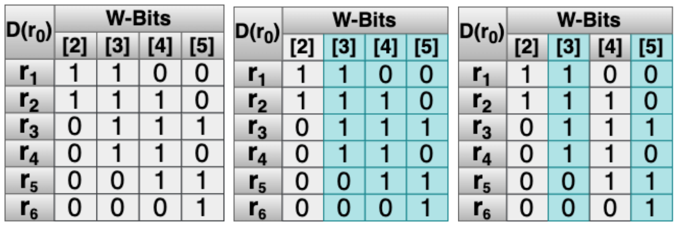
E. Concretization Values

We eliminate the cross-rule dependency between r_f and $r_i \in D(r_f)$ by concretizing $r_f[k]$ from '*' to '1' or '0' at the k-th bit according to the value of $f[k]$, i.e., $r'_f[k] = f[k] = 1/0$. The constructed isolate rule r'_f still matches f at the k-th bit but mismatches the direct dependent rule r_i at the k-th bit, thus eliminating the dependency between r_f and $r_i \in D(r_f)$. The concretization values for wildcard bits can be summarized in two cases.

Case 1: If $f[k] = 1, r_f[k] = *, r_i[k] = 0$, then $r'_f[k] = 1$.

Case 2: If $f[k] = 0, r_f[k] = *, r_i[k] = 1$, then $r'_f[k] = 0$.

As shown in Fig. 6, let $r'_f[5] = f[5] = 1$ according to **Case 1**, the resulting isolate rule $r'_0 = (0**, *11)$ still matches the hot flow $f = (010, 011)$ but no longer matches $r_6 = (0**, *01)$, thus eliminating the dependency between r_0 and r_6. However, $r'_0 = (0**, *11)$ still has dependencies with other dependent rules, including $r_1 = (0**, **1)$, $r_2 = (001, 1*1)$, and $r_4 = (0*1, 111)$. Therefore, concretizing only one wildcard bit w_5 in the best-match rule is insufficient as cross-rule dependencies still exist between r_f and other dependent rules r_1, r_2, and r_4 in $D(r_0)$. The final isolate rule is $r'_0 = (0*0, *11)$, which concretized a total of two wildcard bits in the best-batch rule $r_0 = (0**, **1)$.



$D(r_0)$	W-Bits			
	[2]	[3]	[4]	[5]
r_1	1	1	0	0
r_2	1	1	1	0
r_3	0	1	1	1
r_4	0	1	1	0
r_5	0	0	1	1
r_6	0	0	0	1

(a) concretization matrix for direct dependent rules in $D(r_0)$

$D(r_0)$	W-Bits			
	[2]	[3]	[4]	[5]
r_1	1	1	0	0
r_2	1	1	1	0
r_3	0	1	1	1
r_4	0	1	1	0
r_5	0	0	1	1
r_6	0	0	0	1

(b) example of a sub-optimal solution to concretize $\{w_3, w_4, w_5\}$

$D(r_0)$	W-Bits			
	[2]	[3]	[4]	[5]
r_1	1	1	0	0
r_2	1	1	1	0
r_3	0	1	1	1
r_4	0	1	1	0
r_5	0	0	1	1
r_6	0	0	0	1

(c) example of optimal solution to concretize $\{w_3, w_5\}$

Fig. 7. Examples of concretization matrix C for direct dependent rules $D(r_0)$

V. Algorithm of Wildcard Concretization

In this section, we design algorithms to concretize a minimum number of wildcard bits in the best-match rule under different circumstances to construct isolate rules.

A. Concretization Matrix

Recall that in Fig. 6, the output bit string of the W-XOR operation between the hot flow f and each direct dependent rule $r_i \in D(r_f)$ represents all candidate wildcard bits that can be concretized to eliminate the cross-rule dependency between r_f and r_i. AWEsome-Cache aims to concretize a minimum number of wildcard bits in r_f to construct the isolate rule r'_f, which preserves the original match field of r_f as much as possible.

We organize the output of the W-XOR operation between f and each direct dependent rule $r_i \in D(r_f)$ into a $|D(r_f)| \times |W(r_f)|$ matrix, called the *concretization matrix* as shown in Fig. 7. Each row indexed by r_i represents all candidate wildcard bits $\{w_j\} \subseteq W(r_f)$ in the best-match rule r_f that can be concretized to eliminate its dependency from r_i. For example, the row indexed by r_1 is $[1,1,0,0]$, which means that the dependency between r_0 and r_1 can be eliminated by concretizing the wildcard bits(s) from $\{w_2, w_3\}$, either w_2 or w_3, or both. Each column indexed by $w_j \in W(r_f)$ represents a subset of direct dependent rules $D_j \subseteq D(r_f)$ whose dependencies with the best-match rule r_f will be eliminated if w_j is concretized as $f[j]$. In other words, all direct dependent rules in D_j are *covered* by the j-th wildcard bit w_j in r_f. For example, the column indexed by w_2 is $[1,1,0,0,0]^T$, which means that rules in $D_2 = \{r_1, r_2\}$ will eliminate their dependencies with r_0 if the wildcard bit w_2 of r_0 is concretized as $f[2]$, i.e., $r'_0[2] = f[2]$.

For all wildcard bits $W(r_f)$ in the best-match rule, our objective is to concretize a minimum number of wildcard bits $W^*(r_f) \subseteq W(r_f)$, such that the union of direct dependent rules covered by all selected wildcard bits in $W^*(r_f)$ equals $D(r_f)$, which can be formulated as follows.

$$\cup_{w_j \in W^*(r_f)} D_j = D(r_f), \quad \forall r_f \in R \tag{5}$$

For example, in Fig. 7(b), one feasible solution is $W''(r_0) = \{w_3, w_4, w_5\}$ such that the union of dependent rules covered by these wildcard bits is $D_3 \cup D_4 \cup D_5 = D(r_0)$, and the resulting isolate rule is $r''_0 = (01*, 011)$. As shown in Fig. 7(c), however, the optimal solution in this example is $W^*(r_0) = \{w_3, w_5\}$ and the union of covered dependent rules are $D_3 \cup D_5 = D(r_0)$, and the resulting isolate rule is $r'_0 = (0{*}0, {*}11)$. Compared with the suboptimal solution $r''_0 = (01*, 011)$, the optimal solution $r'_0 = (0{*}0, {*}11)$ preserves one more wildcard bit, and thus potentially captures more packets from future flows.

B. Problem Formulation

The problem of concretizing a minimum number of wildcard bit $W^*(r_f) \subseteq W(r_f)$ to cover $D(r_f)$ can be reduced as the classic Set Cover Problem (SCP), which is proven to be NP-hard [21]. SCP is formulated as follows: given a universe set $U = \{1, 2, ..., m\}$ of m distinct members and a collection of subsets $S = \{S_1, S_2, ..., S_n\}$, where $S_j \subseteq U$. The objective of SCP is to find an optimal sub-collection $S^* \subseteq S$ such that the union of all members in S^* equals U, while the number of subsets in S^* is minimized.

From a matrix perspective, SCP can be formally defined as follows. Let $C = (c_{ij})$ be a binary $m \times n$ matrix. We refer to the index of rows and columns in C as $M = \{1, 2, ..., m\}$ and $N = \{1, 2, ..., n\}$, respectively. We define a column $j \in N$ *covers* a row $i \in M$ if $c_{ij} = 1$. SCP aims to find a minimum number of columns S^* such that all rows are covered by at least one column $j \in S^*$.

The reduction between SCP and our problem can be performed in polynomial time as follows. A member in U is mapped to a unique direct dependent rule $r_i \in D(r)$. A subset $S_i \subseteq S$ corresponds to $D_j \subseteq W$. If our problem can be solved in polynomial time, then SCP can also be solved in polynomial time, which contradicts the NP-hardness of SCP. Therefore, we prove the NP-hardness of our problem.

Next, we propose three algorithms to solve the problem under different circumstances by making trade-offs between optimality and time efficiency. Furthermore, to reduce the solution space, we promote three optimization techniques to pre-process the concretization matrix.

C. Algorithm for Limited Wildcard Bits

Despite the NP-hardness of SCP, the number of wildcard bits $|W(r_f)|$ and the number of direct dependent rules $|D(r_f)|$ are both finite for any best-match rule r_f in our instances. When the number of wildcard bits $|W(r_f)|$ is far less than the number of direct dependent rules $|D(r_f)|$, or the concretization matrix C has a relative small number of columns, we can exhaustively enumerate all possible combinations of wildcard bits in the best-match r_f, and then determine whether the union of rules in $\cup_{j \in S}\{D_j\}$ *covers* all direct dependent rules in $D(r_f)$. The computational complexity of this algorithm is $O(|D(r_f)| \cdot 2^{|W(r_f)|})$, where $|D(r_f)|$ is the number of direct dependent rules, and $|W(r_f)|$ is the number of wildcard bits in the best-match rule r_f.

Moreover, this algorithm can accelerate its convergence speed across the solution space with an early-stop condition by enumerating all possible combinations on an increasing order of wildcard bits: i.e., firstly enumerating combinations with one wildcard bit only, then combinations with two wildcard bits, and finally the combination with all wildcard bits. As the number of wildcard bits in the combinations increases from 1 to $|W(r_f)|$, this variant algorithm would trigger a stop-early condition if a specific combination covers all direct dependent rules in $D(r_f)$. In this case, it is unnecessary to continue enumerating other possible combinations with additional wildcard bits, as they would require concretizing additional number of wildcard bits than those in the early-stop condition. For example, in Fig. 7(c), the algorithm triggered an early-stop condition for the combination with two wildcard bits w_3 and w_5, i.e., $D_3 = \{r_1, r_2, r_3, r_4\}$ and $D_5 = \{r_3, r_5, r_6\}$ as $D_3 \cup D_5 = D(r_f)$.

D. Algorithm for Limited Dependent Rules

When the number of direct dependent rules $|D(r_f)|$ is far less than the number of wildcard bits $|W(r_f)|$, or the concretization matrix C has a relative small number of rows, we propose another algorithm using dynamic programming to find the optimal solution.

We define $F(j) = \{w_1, w_2, ..., w_j\} \subseteq W(r_f)$ as the first j wildcard bits in the best-match rule r_f. When $j = 1$, we constrain the range of candidate wildcard bits to be concretized to w_1, i.e., whether or not to concretize the first wildcard bit w_1. When $j = 1$, we only consider concretizing the first two wildcard bits w_1 and w_2. When the value of j increases to $|W(r_f)|$, all the wildcard bits in r_f are regarded as candidate wildcard bits to be concretized. We design a 2-dimensional table $T[F_j][S]$ for the recursion function, where $S \subseteq D(r_f)$ represents desired subsets of direct dependent rules to be covered. We enumerate S as all possible subsets of direct dependent rules in $D(r_f)$. The value of $T[F_j][S]$ records the minimum number of wildcard bits selected from the first j wildcard bits in F_j that can cover all desired direct dependent rules in $S \subseteq D(r_f)$. Therefore, we have the following recurrence function:

$$T[F_j][S] = \min(T[F_{j-1}][S], T[F_{j-1}][S - D_j] + 1). \quad (6)$$

Recall that we define $D_j \subseteq D(r_f)$ as a subset of direct dependent rules covered by the j-th wildcard bit w_j of the best-match rule r_f.

The value of $T[F_j][S]$ will be updated depending on whether or not to concretize the j-th wildcard bit in the best-match rule r_f. The first term refers to the case that we maintain the original form of the j-th wildcard bit as it was and only concretize specific wildcard bits selected from the first $(j-1)$-th wildcard bits to cover all desired direct dependent rules in S. The second term, however, refers to the case that we assume the j-th wildcard bit in the best-match rule r_f will be concretized, which will accordingly cover all direct dependent rules in D_j. To cover the remaining direct dependent rules in $S - D_j$, we just need to select a minimum number of wildcard bits from the first $(j-1)$-th wildcard bits in F_{j-1} for concretization, denoted by $T[F_{j-1}][S - D_j]$.

As shown in Fig. 8(a), we first initialize the value of $T[F_j][\emptyset]$ as 0, $T[F_1][\{r_1\}]$ and $T[F_1][\{r_2\}]$ as 1, and others as ∞. The algorithm applies dynamic programming until it iterates over all subsets of $D(r_f)$. Since there are $|W(r_f)|$ rows and $2^{|D(r_f)|}$ columns, the computational complexity is $O(|W(r_f)| \cdot 2^{|D(r_f)|})$.

E. Algorithm of Greedy Heuristic

For a large-scale instance in terms of both large number of $|W(r_f)|$ and $|D(r_f)|$, we also propose a greedy heuristic to prioritize time efficiency over solution optimality. We define $w_j.weight$ as the weight of the j-th wildcard bit in the best-match rule r_f, representing the number of direct dependent rules covered by w_j. From the concretization matrix's perspective, the value of $w_j.weight$ equals the number of non-zero entries in the j-th column. As shown in Fig. 7(a), $w_2.weight = 2$ because both r_1 and r_2 will be covered if w_2 is concretized as $f[2]$.

The greedy heuristic selects the wildcard bit with the largest weight at each iteration until all direct dependent rules are covered. The complexity of this greedy heuristic is $O(|W(r_f)||D(r_f)|)$. As shown in Fig. 7(b), $w_3.weight = w_4.weight > w_5.weight > w_2.weight$. Therefore, the solution of the greedy heuristic is $\{w_3, w_4, w_5\}$, though it is suboptimal compared with the optimal solution $\{w_3, w_5\}$ consisting of only two wildcard bits.

F. Optimization Techniques

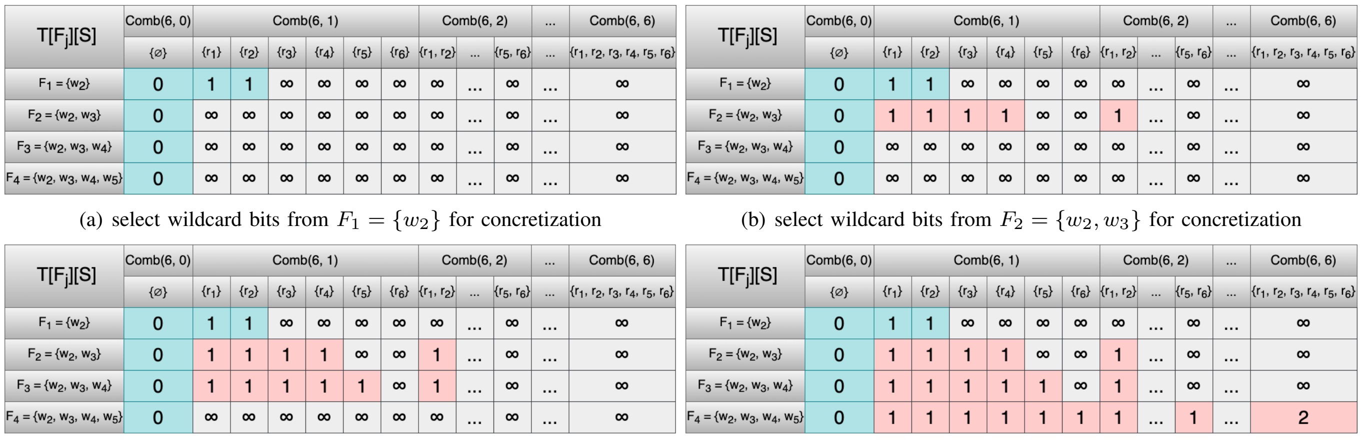
To reduce the solution space, we propose three optimization techniques to pre-process the concretization matrix before applying the above mentioned three algorithms.

1) Inclusion of Unique Wildcard Bit: If a specific direct dependent rule $r_i \in D(r_f)$ can be covered by only one specific wildcard bit w_{uniq} in r_f, then w_{uniq} has to be included in the optimal solution. From a matrix perspective, if a row consists of only one non-zero entry, this wildcard bit corresponding to this non-zero entry must be concretized. For example, in Fig. 9(a), the row indexed by r_6 has only one non-zero entry at w_5; therefore, w_5 is recognized as the unique wildcard bit to cover r_6. The unique wildcard bit w_{uniq} for r_i can be formally defined as follows.

$$w_{uniq} = \{j \in N | c_{ij} = 1, \sum_{k \in N} c_{ik} = 1\}, \quad \forall i \in M \quad (7)$$

2) Removal of Associated Rules: Once a specific wildcard bit w_j is concretized, all rules in D_j will also be covered by w_j. We define dependent rules in D_j are associated rules of w_j. From a matrix perspective, if the column indexed by w_j is concretized, then all associated rules in the j-th column will be removed from $D(r_f)$. For example, in Fig. 9(b), once w_5 is concretized for its uniqueness to cover r_6, then r_3 and r_5 will also be covered as they are both associated rules of w_5. We can formally define associated rules as follows.

$$D_j = \{r_i \in D(r_f) | c_{ij} = 1, i \in M\}, \quad \forall j \in N \quad (8)$$



T[F_j][S]	Comb(6, 0)	Comb(6, 1)						Comb(6, 2)			...	Comb(6, 6)
	{∅}	{r1}	{r2}	{r3}	{r4}	{r5}	{r6}	{r1, r2}	...	{r5, r6}	...	{r1, r2, r3, r4, r5, r6}
F1 = {w2}	0	1	1	∞	∞	∞	∞	∞	...	∞	...	∞
F2 = {w2, w3}	0	∞	∞	∞	∞	∞	∞	∞	...	∞	...	∞
F3 = {w2, w3, w4}	0	∞	∞	∞	∞	∞	∞	∞	...	∞	...	∞
F4 = {w2, w3, w4, w5}	0	∞	∞	∞	∞	∞	∞	∞	...	∞	...	∞

(a) select wildcard bits from $F_1 = \{w_2\}$ for concretization

T[F_j][S]	Comb(6, 0)	Comb(6, 1)						Comb(6, 2)			...	Comb(6, 6)
	{∅}	{r1}	{r2}	{r3}	{r4}	{r5}	{r6}	{r1, r2}	...	{r5, r6}	...	{r1, r2, r3, r4, r5, r6}
F1 = {w2}	0	1	1	∞	∞	∞	∞	∞	...	∞	...	∞
F2 = {w2, w3}	0	1	1	1	1	∞	∞	1	...	∞	...	∞
F3 = {w2, w3, w4}	0	∞	∞	∞	∞	∞	∞	∞	...	∞	...	∞
F4 = {w2, w3, w4, w5}	0	∞	∞	∞	∞	∞	∞	∞	...	∞	...	∞

(b) select wildcard bits from $F_2 = \{w_2, w_3\}$ for concretization

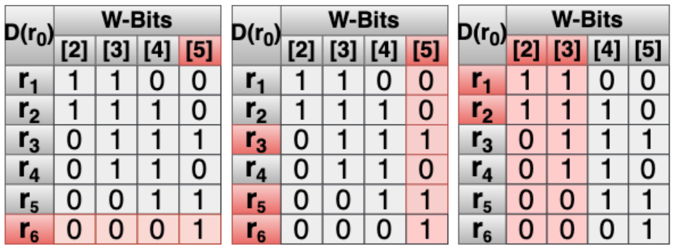
T[F_j][S]	Comb(6, 0)	Comb(6, 1)						Comb(6, 2)			...	Comb(6, 6)
	{∅}	{r1}	{r2}	{r3}	{r4}	{r5}	{r6}	{r1, r2}	...	{r5, r6}	...	{r1, r2, r3, r4, r5, r6}
F1 = {w2}	0	1	1	∞	∞	∞	∞	∞	...	∞	...	∞
F2 = {w2, w3}	0	1	1	1	1	∞	∞	1	...	∞	...	∞
F3 = {w2, w3, w4}	0	1	1	1	1	1	∞	1	...	∞	...	∞
F4 = {w2, w3, w4, w5}	0	∞	∞	∞	∞	∞	∞	∞	...	∞	...	∞

(c) select wildcard bits from $F_3 = \{w_2, w_3, w_4\}$ for concretization

T[F_j][S]	Comb(6, 0)	Comb(6, 1)						Comb(6, 2)			...	Comb(6, 6)
	{∅}	{r1}	{r2}	{r3}	{r4}	{r5}	{r6}	{r1, r2}	...	{r5, r6}	...	{r1, r2, r3, r4, r5, r6}
F1 = {w2}	0	1	1	∞	∞	∞	∞	∞	...	∞	...	∞
F2 = {w2, w3}	0	1	1	1	1	∞	∞	1	...	∞	...	∞
F3 = {w2, w3, w4}	0	1	1	1	1	1	∞	1	...	∞	...	∞
F4 = {w2, w3, w4, w5}	0	1	1	1	1	1	1	1	...	1	...	2

(d) select wildcard bits from $F_4 = \{w_2, w_3, w_4, w_5\}$ for concretization

Fig. 8. Update of recurrence function $T[F_j][S]$ in dynamic programming to cover direct dependent rules in $D(r_0)$ by concretizing $\{w_3, w_5\}$



D(r0)	W-Bits			
	[2]	[3]	[4]	[5]
r1	1	1	0	0
r2	1	1	1	0
r3	0	1	1	1
r4	0	1	1	0
r5	0	0	1	1
r6	0	0	0	1

(a) w_5 is unique wildcard bit to cover r_6

D(r0)	W-Bits			
	[2]	[3]	[4]	[5]
r1	1	1	0	0
r2	1	1	1	0
r3	0	1	1	1
r4	0	1	1	0
r5	0	0	1	1
r6	0	0	0	1

(b) r_3 and r_5 are associated with r_6 once w_5 is concretized

D(r0)	W-Bits			
	[2]	[3]	[4]	[5]
r1	1	1	0	0
r2	1	1	1	0
r3	0	1	1	1
r4	0	1	1	0
r5	0	0	1	1
r6	0	0	0	1

(c) w_2 is redundant as w_3 also covers r_1 and r_2

Fig. 9. Example of concretization matrix with optimization techniques

3) Removal of Redundant Wildcard Bits: If there exists a redundancy between a pair of wildcard bits w_j and w'_j such that $D_j \subseteq D_{j'}$, then w_j is redundant as the concretization of w'_j also covers all dependent rules in D_j. Since there is no benefit to concretizing w_j and w'_j simultaneously, w_j can be removed from the solution space. From a matrix perspective, a column is considered redundant if all of its non its non-zero entries also exist in another column. For example, in Fig. 9(c), w_2 is redundant in the presence of w_3 because r_1 and r_2 can also be covered by concretizing w_3 even if w_2 is not concretized.

VI. TCAM Updates

The measurement from real-world datasets shows that the distribution of traffic locality is not static but changes over time. This section proposes a group-based replacement algorithm for evicting less heavy-hitting rules to make room for newly constructed isolated rules of hot flows.

A. Metrics for Past Performance and Future Potentials

An isolate rule can be freely placed at any empty entry in TCAM because it is dependency-free with any other cached rules. However, when there is no spare entry in TCAM, it requires evicting a cold rule to make room for a newly

Algorithm 1: Cold Rule Replacement

```
Input: $group[k][1:m]$: $k$-th group with $m$ entries
Output: $entryIndex$: index of the coldest entry
/* calculate U for each group */
Function calGroupU($entry[1:m]$):
    $groupHits = sum(entry[i].h)$
    $groupTime = queryTime - min(entry[i].t)$
    return $groupHits/groupTime$
End Function
/* build minHeap for group of minU */
for $k \leftarrow 1$ to $\lceil n/m \rceil$ do
    $minU = min($calGroupU$(group[k].entries))$
    if $minU = -\infty$ then
        return $k$
    end if
end for
$minHeap = MinHeap(minGroup.entries)$
for $i \leftarrow 1$ to $m$ do
    $entryTime = queryTime - minGroup[i].t$
    $u = minGroup[i].h/entryTime$
    $u' = u * |W(minGroup[i])|$
    $minHeap.push(i, u')$
end for
/* query coldest rule */
$entryIndex = minHeap.pop()$
$minHeap.push(isolateRule)$
return $entryIndex$
```

constructed isolate rule. Traditional replacement algorithms periodically query each counter to find the coldest entry [22]–[24]. However, they require frequent interactions between the control and data planes. Worse, they only compare the past performance of cached rules without considering their potential capabilities to capture packets in the future flows. Therefore, we proposes a comprehensive algorithm to identify the coldest entry by considering past performance and future potential.

We define the past performance of a specific entry u_i in TCAM as the cumulative number of cache-hit packets divided by the time since its placement in the TCAM.

$$u_i = \frac{h_i}{t - t_i} \tag{9}$$

where h_i refers to the cumulative cache-hit packets from its initial placement time t_i to the current time t.

Similarly, we define the past performance U for a group of entries as the cumulative number of all cache-hit packets since placing the earliest entry of this group in TCAM.

$$U = \frac{\sum_{i \in U} h_i}{t - t_0} \tag{10}$$

where h_i refers to the cumulative cache-hit packets of the i-th entry in the group, t is the current time, and t_0 represents the current and earliest placement times among all group entries. Note that, we set $U = -\infty$ if there is an empty entry in the group, as there is no need to evict an already cached entry from this group.

Besides the past performance of each entry, we define the potential capability of the i-th entry in TCAM for future flows as the number of wildcard bits in the match field, i.e., $|W(r_i)|$. An isolate rule with more wildcard bits is an aggregation of more exact rules, which is more likely to be hit by packets from future flows than the one with smaller match fields. As a comprehensive metric, we define the utilization of the i-th entry in TCAM as the product of its past performance u_i and its future potential $|W(r_i)|$.

$$u_i' = u_i \times |W(r_i)|. \tag{11}$$

B. Algorithm for Cold Rule Eviction

As shown in **Algorithm 1**, n entries in TCAM are divided into a set of groups with at most m entries in each group. Once there is no empty entry in TCAM to accommodate a newly constructed isolate rule, the control plane will identify a group with the minimum value of U (Lines 6-11) and then evict the coldest rules from this $minGroup$. Then, $minGroup$ builds a $minHeap$ for m entries with respect to their utilizations (Lines 12-18), and returns the coldest one to make room for a newly constructed isolate rule (Lines 19-21). The utilization for the new isolate rule is initialized as the average value of u_i' within its group to avoid an immediate eviction. Note that, if there are groups with empty entries, the newly constructed isolate rule will be directly placed at one of these groups, as there is no need to evict existing entries from TCAM (Lines 8-10). The computational complexities of identifying $minGroup$, building $minHeap$, and returns the coldest rules winthin the group are $O(\lceil n/m \rceil)$, $O(m)$, and $O(\log m)$, respectively.

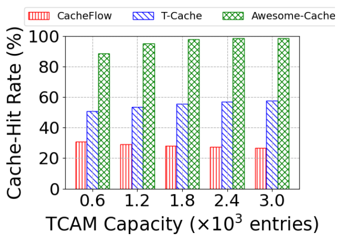
VII. Evaluation

A. Experimental Setup

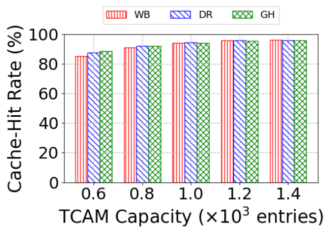
1) Datasets: We evaluate AWEsome-Cache on a Dell PowerEdge R750 commodity server with a 4-core Intel Xeon Silver 4314 CPU and 480GB memory. We use ClassBench [25] to synthesize prefix and non-prefix rules and generate packet traces following the traffic locality, and add timestamps to reflect real-world dynamics. We also use the real-world Tor traffic dataset [26] with both prefix and non-prefix rules in the experiment. The statistics of two types of rulesets and packet traces are detailed in Table I.

TABLE I
Rulesets And Packet Traces

Dataset Type		**# of Rules**	**# of Flows**	**# of Packets**
Synthetic	Prefix	3.00×10^5	1.83×10^5	1.31×10^6
	Non-Prefix	3.61×10^5	1.71×10^6	1.03×10^7
Realistic	Prefix	3.00×10^5	3.12×10^3	4.21×10^5
	Non-Prefix	3.61×10^5	1.18×10^4	5.98×10^5

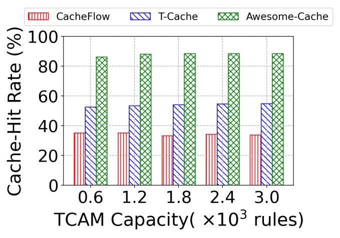


(a) Prefix Rules (ClassBench)

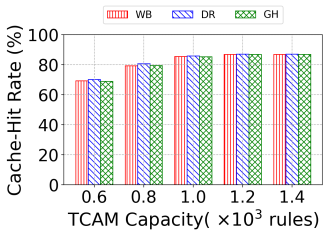


(b) Non-Prefix Rules (ClassBench)

Fig. 10. Cache-hit rates under different TCAM capacities with synthetic traffic



(a) Prefix Rules (Tor)

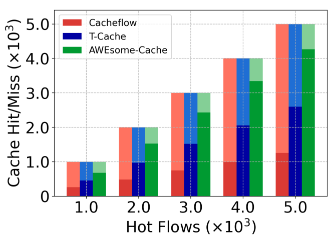


(b) Non-Prefix Rules (Tor)

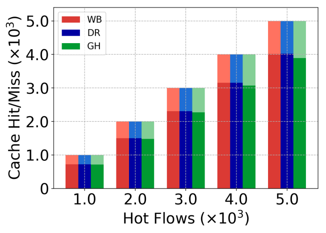
Fig. 11. Cache-hit rates under different TCAM capacities with realistic traffic

2) Baselines: We compare AWEsome-Cache on prefix rules with our most relevant works, i.e., CacheFlow and T-Cache. On the other hand, AWEsome-Cache is the first work designed to eliminate cross-rule dependencies with arbitrary matching patterns, especially for non-prefix rules. Therefore, we compare three algorithms under different circumstances when constructing isolate rules for non-prefix rules in AWEsome-Cache.

- **Wildcard-Bit-Combination (WB)** is favorable when the number of wildcard bits is limited ($|W(r_f)| < |D(r_f)|$).
- **Dependent-Rule-Combination (DR)** is favorable when the number of direct dependent rules is limited ($|D(r_f)| < |W(r_f)|$).
- **Greedy Heuristic (GH)** optimizes time efficiency for large-scale instances but may return sub-optimal solutions.

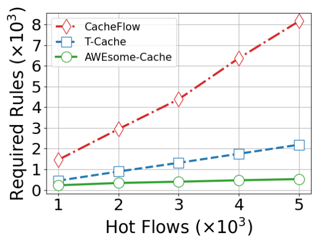


(a) Prefix Rules (ClassBench)

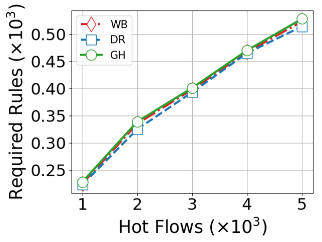


(b) Non-Prefix Rules (ClassBench)

Fig. 12. Match fields to capture hot flows with synthetic traffic

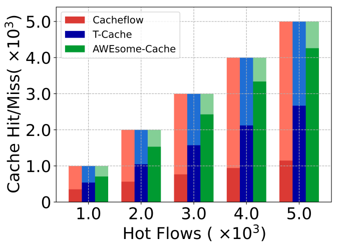


(a) Prefix Rules (ClassBench)

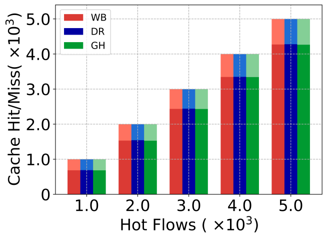


(b) Non-Prefix Rules (ClassBench)

Fig. 14. Comparison of TCAM occupancy for hot flows with synthetic traffic

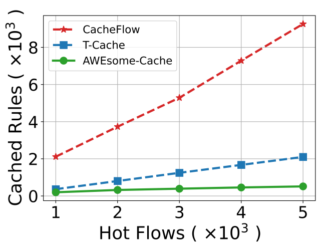


(a) Prefix Rules (Tor)

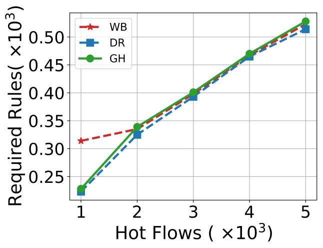


(b) Non-Prefix Rules (Tor)

Fig. 13. Match fields to capture hot flows with realistic traffic



(a) Prefix Rules (Tor)



(b) Non-Prefix Rules (Tor)

Fig. 15. Comparison of TCAM occupancy for hot flows with realistic traffic

B. Experimental Results

1) Cache-Hit Rates: Fig. 10 and Fig. 11 compare the cache-hit rate under different TCAM capacities for synthetic and real-world datasets, respectively.

For prefix rules, AWEsome-Cache consistently outperforms CacheFlow and T-Cache. In Fig. 10(a), when TCAM's capacity reaches 3.0K entries, cache-hit rates for CacheFlow, T-Cache, and AWEsome-Cache are 26.71%, 57.59%, and 98.70%, respectively. The reason is that AWEsome-Cache constructs isolate rules preserving more wildcard bits from the best-match rule than T-Cache; therefore, they capture more packets from future flows. In contrast, CacheFlow caches not only the best-match rules but also their direct dependent rules, some of which are not heavy-hitting rules.

As for non-prefix rules, all three algorithms implemented by AWEsome-Cache have similar performance in terms of cache-hit rates. In Fig. 10(b), when the TCAM's capacity reaches 1.2K entries, cache-hit rates for WB, DR, and GH are 95.86%, 95.78%, and 95.73%, respectively. Since the GH algorithm can obtain sub-optimal solutions, its cache-hit rate is slightly worse than those of WB and DR algorithms.

2) Match Fields: Fig. 12 and Fig. 13 compare the potential capability of match fields to capture packets from different numbers of hot flows.

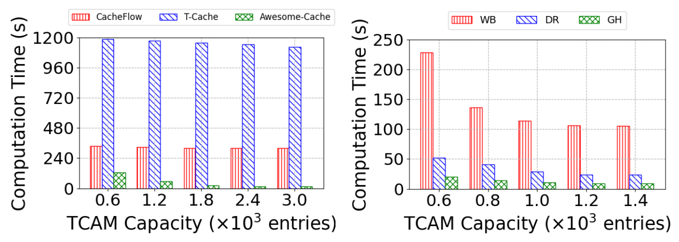
AWEsome-Cache captures more hot flows than CacheFlow and T-Cache for prefix rules because of the preservation of larger match fields in isolate rules. In Fig. 12(a), considering the top 5.0K flows, the number of cache-hit flows in CacheFlow, T-Cache, and AWEsome-Cache are 1257, 2597, and 4265, respectively. The reason is that CacheFlow requires additional entries to accommodate direct dependent rules, which occupies TCAM space that can be otherwise used to capture more hot flows. Compared with T-Cache, AWEsome-Cache preserves more wildcard bits from the best-match, enabling the capture of cache-hit packets from hot flows.

As for non-prefix rules in Fig. 12(b), the cache-hit/cache-miss performance of three AWEsome-Cache variants is similar under different numbers of flows. For example, for the top 2.0K flows, the number of cache-hit flows for WB, DR, and GH are 1499, 1500, and 1479, respectively. The reason is that both WB and DR algorithms construct isolate rules with the largest match fields, while the GH algorithm's sub-optimal leads to fewer cache-hit flows.

3) TCAM Occupancy: Fig. 14 and Fig. 15 show the number of wildcard rules required to be cached in TCAM to capture packets from different numbers of hot flows.

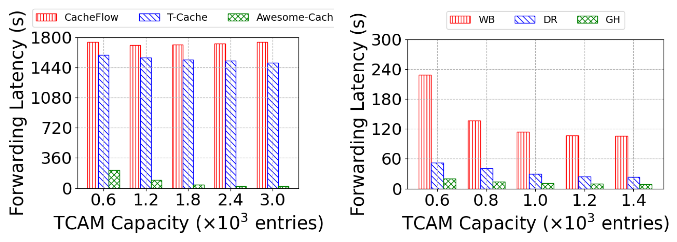
For prefix rules in Fig. 14(a), AWEsome-Cache requires less TCAM occupancy than CacheFlow and T-Cache. Specifically, capturing packets from the top 5K hot flows requires CacheFlow, T-Cache, and AWEsome-Cache to cache 8163, 2189, and 528 rules, respectively. In other words, each isolate rule constructed by AWEsome-Cache can capture an average number of 9.5 hot flows. AWEsome-Cache requires the least TCAM occupancy because it concretizes the least number of wildcard bits to eliminate cross-rule dependencies. In contrast, each isolate rule constructed by T-Cache can capture an average number of 2.3 hot flows due to its significant loss of original match fields. CacheFlow can only partly address cross-rule dependencies but still requires caching all direct dependent rules, leading to the largest TCAM occupancy.

As for non-prefix rules, two exact variants of AWEsome-Cache, including WB or DR, require slightly less TCAM occupancy than the greedy GH heuristic. In Fig. 14(b), considering the top 5.0K flows, WB, DR, and GH require caching 523,



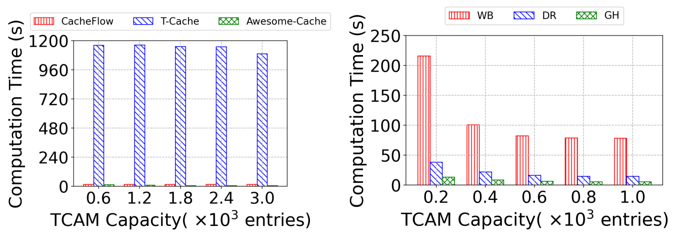
(a) Prefix Rules (ClassBench) (b) Non-Prefix Rules (ClassBench)

Fig. 16. Computation time to construct isolate rules with synthetic traffic



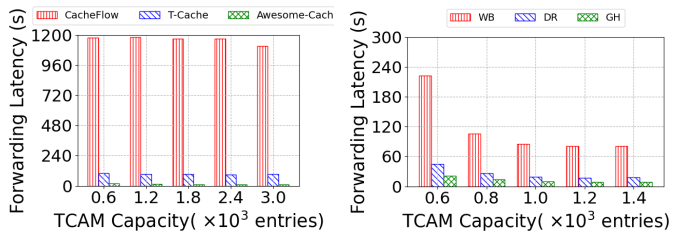
(a) Prefix Rules (ClassBench) (b) Non-Prefix Rules (ClassBench)

Fig. 18. Forwarding latency with synthetic traffic



(a) Prefix Rules (Tor) (b) Non-Prefix Rules (Tor)

Fig. 17. Computation time to construct isolate rules with realistic traffic



(a) Prefix Rules (Tor) (b) Non-Prefix Rules (Tor)

Fig. 19. Forwarding latency with realistic traffic

514, and 528 rules, respectively. The reason is that WB and DR guarantee the optimal solution, while the solution of GH may be sub-optimal.

4) Time Efficiency: Fig. 16 and Fig. 17 show the computation time to construct isolate rules for cache-miss flows under different TCAM capacities.

In Fig. 16(a), when the TCAM capacity reaches 3.0K entries, the computation time for CacheFlow, T-Cache, and AWEsome-Cache are 322s, 1126s, and 14s, respectively. AWEsome-Cache requires the least computational time for prefix rules because three pre-processing techniques significantly reduce the solution space.

As for non-prefix rules in Fig. 16(b), when the capacity of TCAM reaches 1.0K entries, the computation time for cache-miss flows required by WB, DR, and GH algorithms is 114s, 29s, and 11s, respectively. GH requires the least computational time because its computational complexity is not an exponential function of the number of wildcard bits in WB or the number of direct dependent rules in DR.

5) Forwarding Latency: Fig. 18 and Fig. 19 show the overall forwarding latency for all packets from hot flows under different TCAM capacities, which is significantly impacted by the slow lookup speed of cache-miss packets.

For the prefix rules, AWEsome-Cache has the least forwarding latency compared with CacheFlow and T-Cache, as its highest hit rate enables most packets to be forwarded at line rate. In Fig. 18(a), when the TCAM capacity reaches 2.4K entries, the overall forwarding latencies in CacheFlow, T-Cache, and AWEsome-Cache are 1725s, 1520s, and 24s, respectively. On the one hand, AWEsome-Cache constructs isolate rules with the largest match fields, thereby reducing the number of cache-miss flows punted to RAM. On the other hand, the computation time for constructing isolate rules is more efficient than T-Cache.

As for non-prefix rules, GH requires the least forwarding latency than WB and DR, as the optimization techniques significantly reduce the computation time when constructing isolate rules for cache-miss packets. In Fig. 18(b), when the TCAM capacity reaches 1.4K entries, forwarding latencies of WB, DR, and GH algorithms are 105s, 23s, and 9s, respectively.

VIII. Conclusion

This work proposed AWEsome-Cache as a unifying framework to eliminate cross-rule dependencies for wildcard rules with arbitrary matching patterns, especially non-prefix rules. Extensive evaluations on both prefix and non-prefix rules demonstrated that AWEsome-Cache achieves comparable TCAM utilization by caching 75.9% fewer isolate rules for hot flows and efficiently evicting cold rules with TCAM updates. AWEsome-Cache addresses the scalability challenge in the SDN data plane to support fine-grained forwarding policies. We seek to deploy AWEsome-Cache in real networks in our future work.

IX. Acknowledgments

This work is supported in part by the National Key R&D Program of China under Grant No.2022YFB3105000, the National Natural Science Foundation of China under Grant No. 61972189, the Major Key Project of PCL under Grant No. PCL2023AS5-1, and the Shenzhen Key Lab of Software Defined Networking under Grant No.ZDSYS20140509172959989.

References

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

[2] V. Ravikumar and R. N. Mahapatra, "Tcam architecture for ip lookup using prefix properties," *IEEE Micro*, vol. 24, no. 2, pp. 60–69, 2004.

[3] A. Kondel and A. Ganpati, "Evaluating system performance for handling scalability challenge in sdn," in *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pp. 594–597, IEEE, 2015.

[4] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," in *Proceedings of the 7th conference on emerging networking experiments and technologies*, pp. 1–12, 2011.

[5] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[6] B. Isyaku, M. S. Mohd Zahid, M. Bte Kamat, K. Abu Bakar, and F. A. Ghaleb, "Software defined networking flow table management of openflow switches performance and security challenges: A survey," *Future Internet*, vol. 12, no. 9, p. 147, 2020.

[7] V. S. Srinivasavarma and S. Vidhyut, "A tcam-based caching architecture framework for packet classification," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 1, pp. 1–19, 2020.

[8] P. He, W. Zhang, H. Guan, K. Salamatian, and G. Xie, "Partial order theory for fast tcam updates," *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 217–230, 2017.

[9] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 351–362, 2010.

[10] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "Cab: A reactive wildcard rule caching system for software-defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 163–168, 2014.

[11] B. Yan, Y. Xu, and H. J. Chao, "Adaptive wildcard rule cache management for software-defined networks," *IEEE/ACM Transactions on Networking*, vol. 26, no. 2, pp. 962–975, 2018.

[12] X. Li and W. Xie, "Craft: A cache reduction architecture for flow tables in software-defined networks," in *2017 IEEE Symposium on Computers and Communications (ISCC)*, pp. 967–972, IEEE, 2017.

[13] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: Dependency-aware rule-caching for software-defined networks," in *Proceedings of the Symposium on SDN Research*, pp. 1–12, 2016.

[14] J.-P. Sheu and Y.-C. Chuo, "Wildcard rules caching and cache replacement algorithms in software-defined networking," *IEEE Transactions on Network and Service Management*, vol. 13, no. 1, pp. 19–29, 2016.

[15] R. Li, B. Zhao, R. Chen, and J. Zhao, "Taming the wildcards: Towards dependency-free rule caching with freecache," in *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pp. 1–10, IEEE, 2020.

[16] R. Li, Y. Pang, J. Zhao, and X. Wang, "A tale of two (flow) tables: Demystifying rule caching in openflow switches," in *Proceedings of the 48th International Conference on Parallel Processing*, pp. 1–10, 2019.

[17] Y. Wan, H. Song, Y. Xu, Y. Wang, T. Pan, C. Zhang, and B. Liu, "T-cache: Dependency-free ternary rule cache for policy-based forwarding," in *Proceedings of IEEE INFOCOM 2020*, pp. 536–545.

[18] Y. Wan, H. Song, Y. Xu, Y. Wang, T. Pan, C. Zhang, Y. Wang, and B. Liu, "T-cache: Efficient policy-based forwarding using small tcam," *IEEE/ACM ToN*, vol. 29, no. 6, pp. 2693–2708, 2021.

[19] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging zipf's law for traffic offloading," *SIGCOMM Computer Communication Review*, vol. 42, no. 1, pp. 16–22, 2012.

[20] "The caida ucsd anonymized internet traces." www.caida.org/catalog/datasets/passive_dataset_download/.

[21] A. Caprara, P. Toth, and M. Fischetti, "Algorithms for the set covering problem," *Annals of Operations Research*, vol. 98, no. 1-4, pp. 353–371, 2000.

[22] L. Yu, J. Sonchack, and V. Liu, "Mantis: Reactive programmable switches," 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*, pp. 296–309, 2020.

[23] Z. Su, T. Wang, Y. Xia, and M. Hamdi, "Cemon: A cost-effective flow monitoring system in software defined networks," *Computer Networks*, vol. 92, pp. 101–115, 2015.

[24] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, pp. 254–265, 2011.

[25] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM transactions on networking*, vol. 15, no. 3, pp. 499–511, 2007.

[26] A. H. Lashkari, G. D. Gil, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of tor traffic using time based features," in *International Conference on Information Systems Security and Privacy*, vol. 2, pp. 253–262, SciTePress, 2017.