BOLT: Scalable and Cost-Efficient Multi-string Pattern Matching With Programmable Switches

Shicheng Wang, Menghao Zhang, Guanyu Li, Chang Liu, Zhiliang Wang, Member, IEEE, Yixing Liu, Member, IEEE, and Mingwei Xu

Abstract—Multi-string pattern matching is a crucial building block for many network security applications and thus of great importance. Since every byte of a packet has to be inspected by a large set of patterns, it often becomes a bottleneck of these applications and dominates the performance of an entire system. Many existing studies have been devoted to alleviating this performance bottleneck either by algorithm optimization or hardware acceleration. However, neither one provides the desired scalability and costs that keep pace with the drastic increase in network bandwidth and traffic today. To address these issues, in this paper, we present BOLT, a scalable and cost-efficient multi-string pattern matching system leveraging the capability of programmable switches. BOLT combines the following techniques: (1) an efficient state encoding scheme to fit a large number of strings into the limited memory on a programmable switch; (2) a variable k-stride transition mechanism to increase the throughput significantly with the same level of memory cost; and (3) a compact pattern2rule mapping method to accommodate multiple co-existing strings in one rule. We implement a prototype of BOLT and make its source code publicly available. Extensive evaluations demonstrate that BOLT can provide multi-hundred Gbps throughput and scales well with various pattern sets and workloads.

Index Terms—Programmable switch, pattern matching.

I. INTRODUCTION

MULTI-STRING pattern matching serves as a fundamental building block for many network security applications, especially network intrusion/prevention systems (NIDS/NIPS) [2], [3], web application firewalls (WAF) [4], application identification systems [5] and some network censor/piracy systems [6], [7]. In these applications, multiple strings are usually represented as the attack signatures (rules), which are then used to inspect whether the payload of a packet matches any of the predefined rules. Since every byte of the packets has to be scanned by a large set of patterns, this often becomes a bottleneck of these applications and dominates the performance of an entire system [8], [9].

Prior studies often alleviate this bottleneck via algorithm optimization [10], [11], [12], [13], [14] or GPU/FPGA/NPU acceleration [9], [15], [16], [17], [18]. However, neither these existing software-improved nor hardware-accelerated solutions provide the desired scalability or costs that catch up with the drastic increase in network bandwidth and traffic today. Recently, the network bandwidth at traffic aggregation points in regional ISPs has already reached multi-100s of Gbps [19], [20]. Many network device providers [21], [22] and standard organizations [23] are embracing the era of 400Gbps bandwidth [22], [24]. These high network bandwidths require that the ability of security applications to maintain network monitoring should also keep pace with such high traffic volume. Even when we fully utilize the potential of CPUs on servers, however, it is difficult for a pattern matching engine to reach 20 Gbps packet processing throughput [12], [13]. While GPU/FPGA/NPU-enhanced servers can achieve higher throughput (multi-10s Gbps) due to the inherent parallelism of the hardware [11], [14], [25], there is still an impassable throughput gap. Although we can scale up the pattern matching capacity by adding more servers, doing so raises the capital cost and the management complexity significantly [26], [27].

We observe that the emerging programmable switching ASICs [28] in the network community provide a promising opportunity to bridge this gap. Since a single programmable switch can efficiently process multi-Tbps traffic at line rate at the same level of power and capital costs as regular switches [29], it has several orders of magnitude higher packet processing throughput than highly-optimized servers. Even for FPGA/GPU/NPU-enhanced servers, there is still a significant gap to match this performance. In addition, programmable switches allow users to specify the hardware logic using domain-specific languages (e.g., P4 [30]) and support packet processing with use-defined logic at terabit line rate. These new characteristics of programmable switches are particularly valuable for next-generation scalable and cost-efficient multi-string pattern matching.

However, implementing multi-string pattern matching on programmable switches (i.e., Protocol Independent Switch Architecture (PISA)) is non-trivial. First, current security applications usually maintain a large set of rules (e.g., the latest...
community rule set of Snort has ~4000 rules), while the memory resources in the switch are pretty limited (50-100MB [31]). Simply translating string patterns in the rule set into a deterministic finite automaton (DFA) and then the corresponding match-action entries, as PPS [32] does, will exhaust precious resources in switches. Worse yet, a large number of entries will inevitably increase the time to update the rule set in switches, making the system less responsive. Second, the computational model in programmable switches is quite restricted compared with x86 CPUs. In particular, current programmable switches cannot support iterations and loops, which are key components in pattern matching algorithms. This indicates that the depth of payload inspection is limited in one pass of the pipeline. An intuitive method is to increase the stride of the DFA transition [32], but it incurs the explosion of entries. Third, many security applications, such as Snort signature databases, usually employ multiple strings to express one attack signature (rule). For example, Fig. 1 shows one typical snort rule (sid 978), which requires %20, &CiRestriction=none and &CiHiLiteType=Full co-exist. This raises the requirement that programmable switches should support multiple strings in one rule, which is challenging to be efficiently implemented on the restricted computational model of programmable switches. Simply forwarding the packets once one pattern is matched to the backend server running a complete NIDS for “full matching”, as some methods on other hardware do [18], still causes significant burdens on the backend server since the packet processing throughput of the switch is several orders of magnitude higher than that of the commodity server.

To address these problems, in this paper, we propose BOLT, a system for inspecting multiple string patterns with programmable switches. First, BOLT develops an efficient state encoding scheme to fit a large number of rules into the limited memory in programmable switches. Second, BOLT proposes a variable k-stride transition mechanism to improve the throughput significantly with an acceptable entry number increase. Third, BOLT designs a compact pattern2rule mapping method to support multiple strings in one rule. We implement a prototype of BOLT and make the source code publicly available [33]. Extensive evaluations show the compressing techniques of BOLT can significantly decrease the number of entries and memory usage, and the pattern2rule mapping method achieves an efficient pattern-to-rule mapping with acceptable overheads. To conclude, BOLT can provide multi-hundred Gbps throughput, about one order of magnitude improvement in throughput, and scale well with various pattern sets and workloads.

In summary, this paper makes the following contributions:

- We highlight the challenges that current multi-string pattern matching faces in dealing with the soaring network bandwidth today and identify the opportunities provided by programmable switches (§II).
- We propose BOLT, a scalable and cost-efficient multi-string pattern matching system with programmable switching ASICs (§III). To this end, we design an efficient state encoding scheme, a variable k-stride transition mechanism, and a compact pattern2rule mapping method to overcome the restrictions of the computational model and memory resources of programmable switches (§IV, §V, §VI).
- We implement an open-source prototype of BOLT, and conduct extensive evaluations to show the advantages of BOLT (§VIII).

Finally, we make some discussions in §IX, describe related works in §X and conclude this paper in §XI.

II. BACKGROUND & MOTIVATION

In this section, we introduce the background of multi-string pattern matching, highlight the problems of the state of the art in this field, and discuss the opportunities provided by programmable switching ASICs.

A. Multi-String Pattern Matching

As one of the most classic problems in algorithms, string-based pattern matching has been studied for decades. Formally, the string-based pattern matching algorithm can be denoted as follows. Given an alphabet $\Sigma$, the algorithm’s input contains 1) a text string $T = t_1...t_n$, and 2) a pattern string set $P = \{p_1, p_2, ..., p_k\}$, where each element $p_i = s_1...s_m$ is a predefined string pattern (every $t_i$ or $s_i$ is a character belonging to $\Sigma$). The algorithm should output all the positions where each $p_i$ stands as a substring in $T$.

Many algorithms have been devoted to this field. The Knuth-Morris-Pratt (KMP) algorithm [34], as one of the most effective exact string matching solutions, runs in $O(n)$ time by constructing a partial match table in pre-processing. However, it only performs well for a singleton pattern set. A pattern set with $m$ elements will lead to an $O(mn)$ running time complexity. The Boyer-Moore (BM) algorithm [35] is another efficient string-searching algorithm, which is usually viewed as a standard benchmark [36] for string matching. It gathers information during the pre-processing stage to skip multiple characters in the text, reducing the best running time to $\Omega(n/m)$, while the worst case is still $\Omega(mn)$. However, it does not work efficiently for multi-string pattern matching either. Naively applying the BM algorithm to multi-string patterns requires iterating the text string $|P|$ times in order to look for every $p_i$ in $P$. Some algorithms improve the matching efficiency by using approximation methods. The Rabin–Karp algorithm [37] innovatively speeds up the string matching by using hash functions. DFC [12] and Hyperscan [13] further promote the performance by designing cache-friendly data structure or SIMD (single-instruction-multiple-data)-accelerated algorithm variants. However, hash-based algorithms require many hash units, which is difficult to support on other hardware.

The Aho-Corasick (AC) algorithm [38] is an efficient solution to this multi-string pattern matching problem. It builds a non-deterministic finite automaton (NFA) by constructing goto transitions from a trie resembling the pattern set, and failure transitions between nodes sharing a common prefix. The AC algorithm runs in $O(n + m + z)$ time, where $z$ is the count of matches. Because of its efficiency, the AC algorithm has become the de facto standard for the pattern matching, and is widely used in many state-of-the-art network security applications, such as Snort [2] and Suricata [39].

B. Problems of Current Approaches

Since the pattern matching has to inspect every byte of a packet across a set of patterns, it usually becomes a
bottleneck of an entire network security application. Many studies have been conducted to alleviate this performance bottleneck, either through algorithm optimization or through hardware acceleration. Software-based algorithm optimization techniques attempt to either minimize the memory usage [40], [41], [42] or increase the number of characters per transition [10], [11], [14], achieving several times larger throughput. However, the packet processing performance of software is intrinsically limited, because the CPU on servers is not specialized for high-speed packet processing. For example, even with highly-optimized algorithms and data structures, it is still impossible for a server-based pattern matching engine to reach 20 Gbps packet processing throughput [13]. Although we can achieve higher throughput by deploying more servers, doing so would increase the capital and operational costs drastically [26], [27], which is not symmetric to the rapid growth of network bandwidth and network traffic nowadays.

Besides the solutions to optimize the pattern matching algorithms on software, utilizing dedicated hardware to accelerate the pattern matching has also attracted great attention. GPUs are becoming popular for the pattern matching tasks because of their high parallelism compared with CPUs. The single instruction multiple threads (SIMT) architecture can efficiently execute an algorithm in parallel, thus providing higher multi-10 Gbps traffic processing throughput [9], [43], [44], [45]. However, keeping the performance at the peak rate is difficult since it requires that all computing elements have the same instruction flow, which is quite challenging to achieve [46]. Besides, GPUs also induce higher power costs. FPGA-based solutions leverage the underlying circuit-level parallelism to accelerate multi-string pattern matching and even regular expression matching [15], [17], [18], [47], achieving higher throughput but lower flexibility and a longer development cycle. Moreover, it is still difficult for these hardware alternatives to match their performance with the current network traffic volume. Worse yet, these hardware alternatives are usually attached to servers through PCIe, making it difficult to explore their full potential because of the limited PCIe bandwidth.

To summarize, neither prior algorithm-optimized nor hardware-accelerated solutions provide the desired throughput that can catch up with the drastic increase of network traffic and network bandwidth today. There is a strong desire for a next-generation high-throughput and cost-efficient pattern matching engine.

C. Opportunities by Programmable Switches

Recent advances in the programmable network have enabled programmability from the control plane to the data plane, with emerging hardware such as switching ASICs [28], and the corresponding domain-specific language such as P4 [30] and NPL [48]. Programmable switching ASICs provide flexible data plane forwarding capability with customized packet processing logic at high throughput and low costs [49], [50], and thus equip us with a promising opportunity to offload a group of network functions [27], [31], [51], [52], [53] from expensive servers or middleboxes.

In programmable switching ASICs, there are multiple ingress and egress pipelines, and each of them has several ingress and egress ports. Incoming packets will be sequentially processed by multiple stages in the ingress/egress pipeline respectively. Each stage is a packet processing unit with dedicated resources, including match-action tables, registers, and stateful ALUs. Match-action tables match certain header fields or metadata fields, and perform customized actions configured by the table entries, e.g., modifying headers/metadata, reading/writing registers. Stateful ALUs support customized calculations based on headers/metadata/registers. Registers store states to support stateful packet processing. With programmable switching ASICs, operators can customize the data plane logic by using domain-specific languages (e.g., P4 [30]). Then the source P4 code is compiled into binaries to be loaded into the switch, and interactive APIs to be invoked by the control plane to update the match-action tables and registers during runtime. Once the P4 program is installed successfully into the switch pipeline, it runs at line rate with a fixed stage number as well as bounded memory access.

Programmable switching ASICs and P4 language make it straightforward to achieve customized line-rate terabit network functions, as long as the user-defined logic satisfies the computational model and resource constraints of programmable switches. Furthermore, programmable switches have a similar level of power consumption (0.1Watts/Gbps [27], [54], [55]) and capital costs as traditional fixed-function switches, which enables orders of magnitude cost reduction compared to commodity CPUs or other hardware alternatives (e.g., GPU, FPGA, NPU).

III. DESIGN OVERVIEW

In this section, we describe the expected scenario and the workflow of BOLT in more detail.

A. Expected Scenario

BOLT focuses on accelerating the core function of many network security applications, multi-string pattern matching. It acts as a sub-system or a function instance of a network security application, inspecting the payload in byte granularity, and taking the corresponding action defined by the rule (e.g., alerting, passing, and dropping, etc.). The switch could be deployed as a middlebox in the link, or as a dedicated application analyzing the traffic like a bypass tap. Notably, programmable switching ASICs will discard the inspected portion of the payload and recirculate the packet for deeper payload inspection, thus truncating the packets during the inspection. Therefore, when deploying in-line, an additional buffer mechanism is required to avoid information loss [32], [55], [56]. The packets are temporarily buffered during the inspection, and will be evicted and forwarded normally or for further complete inspection (such as regular expression pattern matching) according to the matching results.

We assume the incoming packet consists of an Ethernet header, an IP header, a UDP/TCP header and a payload to inspect. The packet payload can be encoded in any pattern (ASCII, UTF-8 or binary data) and we adopt ASCII in our paper, where each character in the alphabet is encoded in 1 byte. Note that our method is devoted to cases where the payload only consists of plain text, such as traffic scrubbing and inspection without encryption deployed. Encrypted traffic processing is out of our scope, and a decryption mechanism

1The cost-efficiency of packet processing on programmable switches has been verified by numerous other recent works [27], [54], [55], as a result, we do not illustrate more in this paper.

2In fact, we can take any layer header as payload.
must be integrated in that case, such as the SSL acceleration service in WAF (Web Application Firewall) [4], [12].

B. Workflow

The workflow of BOLT is illustrated in Fig. 2. Operators first need to define a list of matching rules from the signature database (e.g., Snort community rules [57]). Then the controller extracts string patterns and their correlations from the matching rules. These string patterns are constructed into a nondeterministic finite automaton (NFA) with the AC algorithm [38], and then translated into underlying pattern table entries. The correlations between different string patterns in a rule is translated into the rule table in the switch pipeline. Our first key enabler here is an efficient state encoding scheme, which takes advantage of the “don’t care” feature of TCAMs in the match-action table of programmable switches (§IV). We also apply a variable k-stride transition mechanism to increase the average transition stride and the average throughput, while acceptably increasing the number of table entries (§V). To determine which rule in the ruleset is triggered by the incoming packet according to the matched patterns, we also design a compact pattern2rule mapping method to express multiple co-existing string patterns in one rule (§VI).

With these efficient match-action tables, the data plane conducts matching for every packet byte by byte in the pipeline. During the matching procedure, the data plane carries out the corresponding actions, such as dropping, passing, alerting or forwarding to the backend server.

IV. EFFICIENT STATE ENCODING

In this section, we analyze the shortcomings of existing DFA-based entry generation methods, and give our observation that the feature of match-action tables (i.e., ternary match and entry priority) helps implement the AC NFA efficiently on programmable switches (§IV). We also apply a variable k-stride transition mechanism to increase the average transition stride and the average throughput, while acceptably increasing the number of table entries (§V). To determine which rule in the ruleset is triggered by the incoming packet according to the matched patterns, we also design a compact pattern2rule mapping method to express multiple co-existing string patterns in one rule (§VI).

A. Problem Analysis

To achieve multi-string pattern matching on hardware, a typical method [25], [32] is to (1) construct an NFA consisting of goto and failure transitions with the AC algorithm; (2) convert the NFA into an equivalent DFA; (3) translate each transition edge in the DFA into a single entry in match-action tables. However, the NFA-equivalent DFA is built by powerset construction [38], which has many more transitions than its corresponding NFA. Fig. 3(a) shows the NFA for matching \{she, her, he\} built by the AC algorithm, where the solid lines denote goto transitions and the dashed lines denote failure transitions (failure transitions to $s_0$ are omitted for clarity and the complete transitions are defined in table $f(s_i)$). Its corresponding DFA is shown in Fig. 3(b), and we also omit the trivial transitions returning to the root state $s_0$. From Fig. 3, we can see that the DFA has many more transition edges than the NFA, which requires more table entries in the data plane. Although we can represent all the trivial transitions to $s_0$ by setting a default action, the increased non-trivial transitions (lines in red) still account for a large amount of data plane memory. While many previous works [12], [13], [25], [58], [59] have attempted to compress the DFA to save memory, they either require multiple memory accesses for a single input character [25], [58], [59], wasting precious pipeline stages, or require complex computation [12], [13] which is difficult to support on the switches. Besides, it is still hard for these compression algorithms to achieve an optimal condition on programmable switches, which would inevitably lead to table entry increase or resource efficiency degradation.

We observe that the unique features of match-action tables in programmable switches, including the ternary match and...
the priority entry, provide us a unique opportunity to directly translate the AC NFA into match-action table entries efficiently. However, an AC NFA’s matching logic is hard to implement under the matching semantics of match-action tables, because it requires iterations and loops, which are quite challenging to support in the data plane. To illustrate this, we highlight the key difference between the DFA matching and the AC NFA matching here. In the DFA matching, the current state under a current input will move to the next state in a deterministic manner by following the transition edge in the DFA. While in the AC NFA, under the current input, the next state is not only determined by the goto transitions starting from the current state, but also the failure transitions. The NFA will examine the input multiple times along the path defined by the failure transition. This feature of the NFA means that a single goto transition could be potentially executed by multiple states, as long as the source state for this goto transition exists on the failure transition path. For example, for the DFA in Fig. 3(a), when state $s_3$ gets input $r$, it will deterministically move to $s_6$ according to the transition edges. But in the NFA, as Fig. 3(b) shows, when state $s_3$ gets input $r$, $s_3$ itself has no goto transition matching $r$, so the current state gets moved to $s_0$ along its failure transition ($f(s_3) = s_0$), which has a goto transition matching $r$ and going to $s_0$. Therefore, this input is consumed by the goto transition of $s_3$.

Based on this observation, if we can smartly encode the state with ternary bits so that the entry for a specific state $s_i$ could match not only itself, but also the states which can move to it along the failure transition (i.e., these states are shadowed by $s_i$), we will only need to convert every goto transition into an entry, reducing the number of entries drastically. As discussed above, the encoding scheme should satisfy the following properties:

- The ternary code of a state $s_i$ should cover the exact code of $s_i$ and every state deferring to $s_i$ by the failure function.
- The ternary code of a state $s_i$ must not cover the exact code of the states not deferring to $s_i$ by the failure function.
- Any two distinct states should have different ternary codes and exact codes.

B. State Encoding and Table Entry Generation

In this subsection, we elaborate our approach to encoding the states and generating the table entries, as depicted in Fig. 4:

1. Construct a failure transition tree denoting the deferment relationship defined by failure transitions;
2. Encode the state with a shadow encoding scheme, assigning each state a ternary code in the match field and an exact code in the action field;
3. Assign different priorities for each entry to achieve the complete semantics of the AC NFA. Details are illustrated below.

First, we build a failure transition tree from the failure transition table. The failure transition tree holds the property that every node may move to its ancestors looking for goto transitions to match an input character. As Fig. 4 shows, $s_2$ has ancestors $s_4$ and $s_0$, because $f(s_2) = s_4$, and $f(s_4) = s_0$. Obviously, the root state $s_0$ is the root of the failure transition tree because every state will eventually move to $s_0$ according to the failure transition table, determined by the construction process of failure transitions.

Second, we encode each state with two codes, a ternary code and an exact code, based on the failure transition tree. For each node, the ternary code should cover the exact code of itself and also the descendants in the failure transition tree. To achieve this, we build our encoding scheme on a classic shadow encoding [60] algorithm, which can assign codes for each state in a specific failure transition tree effectively. The shadow encoding algorithm was originally proposed in the D$^2$FA (Delay-input DFA) [61], which was introduced to reduce the entry number of DFA. It removes redundant transitions from the state $p$ whose input character and destination state are the same as the transitions from $q$, making this transition deferred to $q$ to be executed. This algorithm assigns each state a binary exact code and a ternary code, to make the exact state code of $p$ be matched by the ternary code of $q$, achieving the “delay” matching. We find it surprisingly satisfies the three properties we claimed in §IV-A. By utilizing a Huffman coding style algorithm, the shadow encoding algorithm provides a unique signature to distinguish different states, while increasing the state code width negligibly.

Finally, we convert each goto transition into a table entry. We assign priority values for the entries in the post-order of the failure transition tree so that the transition entry priority of children is higher than that of their parents, achieving the logic that children will look up for matching along the failure transition path iteratively. Taking Fig. 4 as an example, entries for $s_2$ is arranged in front of entries for $s_4$, because $s_2$ is the child of $s_4$. And we place the entries of $s_0$ in the last, because $s_0$ is the parent of all the other states in the failure transition tree. As we can see from Fig. 4, the entry number is equal to the number of goto transitions in the NFA.

V. VARIABLE k-STRIDE TRANSITIONS

In this section, we first identify why the current methods that increase the stride size of transitions add significant memory costs, and then we illustrate our variable $k$-stride AC NFA method and why it works correctly.

A. Strawman Methods

Increasing the stride of transitions to $k$ would improve the throughput by a factor of $k$. However, naively increasing the stride size comes at significant memory costs. One common method is to construct a $k$-stride DFA from the 1-stride AC DFA using ternary matching to omit trivial transitions,
to achieve deterministic $k$ characters consumed per matching [32], [62]. Although this method can lower the base of exponential increase of transition number, unfortunately, many redundant transitions are still introduced. Assuming $\delta_k(s_i, str) = s_j$ denotes a $k$-stride transition function from $s_i$ to $s_j$ by string $str$. For example, in Fig. 3(b), concatenating $\delta_1(s_3, e) = s_3$ with the transition function of the successive state, $\delta_1(s_3, r) = s_6$, can obtain a 2-stride transition function for $s_2, \delta_2(s_2, er) = s_6$. 1-stride trivial transitions for $s_2$ (input is $\neg\{e\}$) can be represented by a single one $\delta_1(s_2, \ast) = s_0$ to concatenate with its successive state, as Fig. 5 shows. But this method still has many redundancies to optimize. For example, $\delta_2(s_3, sh) = s_2$ is redundant with $\delta_2(s_0, sh) = s_2$, because $s_2$ will move along the failure transition to $s_0$ to look for the next transition, in other word, $s_2$ can be potentially shadowed by $s_0$ even in 2-stride transition. Such redundancy in $k$-stride transition results in extra memory wastes. Worse yet, besides the transition explosion, enlarging the stride will also lead to a state explosion. A $k$-stride transition path contains $k - 1$ intermediate states, and any combination of accepting states on this transition path implies matching a unique combination of patterns. For example, in Fig. 5, both input “$\ast e$” ($\neg\{s, h, r\}$) and “$\ast \ast$” ($\neg\{e\} \neg\{s, h\}$) lead a transition from $s_2$ to $s_0$, but state transition with input “$\ast e$” will output matching for pattern “she” and “he” while “$\ast \ast$” not. We need to allocate a new state for every possible combination of accepting states in the path [32], [60], causing the state explosion. Alicherry et al. [25] propose a highly-optimized DFA which has variable stride length, but it still has large memory usage resulting from additional states, and the average stride is small because there could be a negative stride in some cases. Backwards packet inspection is also difficult to implement in the switch pipeline.

In addition to multi-stride DFAs, some works increase the stride size for AC NFAs by only concatenating the $goto$ transitions, reducing the redundant transitions. Yun et al. [63] propose a $k$-AC NFA constructing method, which consumes exact $k$ input characters on state transition in a memory-efficient way. They solve the states explosion problem by decoupling the state transition and output into two match-action tables to achieve simultaneous matching. This requirement is infeasible in current switching ASICs, because it needs two stages to implement one complete state transition, leading to $k^2$ characters consumption per stage. Since the number of stages is very limited in programmable switches, reducing the bytes consumed by each stage will increase the recirculation times in the pipeline and the bandwidth usage per packet, finally degrading the throughput.

3Assuming an alphabet $\Sigma$, in a naive $k$-stride DFA, each state may have conceptually $|\Sigma|^k$ transitions outgoing from it.

To summarize, currently, neither DFA-based methods nor NFA-based methods are suitable in the model of programmable switches. In DFA-based methods, redundant transitions and new assistant states should be added [25], [32], [60], while in NFA-based methods, it requires multiple tables (i.e., multiple stages in a programmable switch) to handle a single transition [63]. Both methods lead to unnecessary resource waste for programmable switches.

B. Variable k-Stride Transitions

In this subsection, we propose a variable $k$-stride AC NFA under the programmable switch scenario, to increase the average throughput, while ensuring space usage increases slowly.

The root drawback in previous approaches is (1) the redundant transitions and (2) accepting state(s) in the intermediate node(s) on a transition path. Our method constructs the variable $k$-stride AC NFA whose failure transition is exactly the same as the 1-stride AC NFA, which means that we can use the shadow code from the original AC NFA directly. Therefore, we can avoid redundant transitions by allowing states to match the variable $k$-stride $goto$ transitions from the ones shadowing them, as we do in §IV. Besides, our methodology in constructing the variable $k$-stride $goto$ transition will stop increasing the transition stride once encountering an accepting state, to capture as many as possible $k$-stride transitions using relatively fewer entries. Furthermore, for root state, we deploy a self-unlooping mechanism to unroll the self-loops, increasing the average stride of transitions from root, while ensuring the correctness of matching.

Constructing the variable $k$-stride $goto$ transitions from non-root states is intuitive but efficient. For example, to get variable 2-stride $goto$ transitions from $s_1$, we concatenate the 1-stride transition $\delta_1(s_1, h) = s_2$ with the 1-stride transition for the successive state $s_2, \delta_1(s_2, e) = s_3$, to obtain a 2-stride transition function $\delta_2(s_1, he) = s_3$. In this way, we can compute the $k$-stride $goto$ transitions iteratively from $(k - 1)$-stride $goto$ transitions. The left side in Fig. 6 shows the variable 2-stride $goto$ transition from the AC NFA in Fig. 3(a). The entry stride for $s_2$ is 1, because it encounters $s_3$, an accepting state within 1 stride. For any state $s_i$, we stop increasing the stride size when encountering an accepting state on its $k$-stride transition. This can avoid extra states or table entries to represent the combination of multiple accepting states in the path, as we discussed above.

The root state $s_0$ is special because when $k$ characters are processed at a time, the pattern in the pattern set could start at any position within the $k$ block. If we directly expand the transition stride size to 2 for $s_0$ in the same way as other non-root states, we can get the table entries for $s_0$ as the upper right part in Fig. 6 shows. However, this table cannot carry out the pattern matching correctly. For example, the input $\text{her}$ will not match, so the pattern $\text{her}$ will miss. Meiners et al. [60] propose a self-loop unrolling mechanism to solve this problem in $k$-stride DFA by prepending wildcard $*$ to the initial transition table entries, increasing the stride of the transitions with a linear increase in the number of entries. However, this method does not increase the stride of transitions evenly. As the lower right part in Fig. 6 shows, the self-unlooping transitions for $s_0$ provide the stride from 1 to $k$. We address this by first constructing $k$-stride $goto$ transitions as other states, then deploying self-loop unrolling on them.
In summary, our method goes as follows: We first increase the stride of transitions to \( k \) for all the states (stopping transitions at accepting states) by concatenating the goto transitions in the AC NFA, as the upper right side shows in Fig. 6. Then we unroll the self-loops in \( s_0 \). We iteratively right-shift the \( k \)-stride-transitions from the root state and prepend them with wildcards (*). This method increases the stride to \( k \) with only additional \( O(k) \) growth of the entry number, scaling linearly with the stride \( k \). The lower left side in Fig. 6 illustrates a 2-stride self-loop unrolling table for \( s_0 \). With these two steps, we get all the variable \( k \)-stride goto transitions.

In particular, Fig. 7 shows the final variable 2-AC NFA derived from Fig. 3(a).

### C. Correctness Proof

In this subsection, we mainly explain why this variable \( k \)-AC NFA has the same state set and failure transition table as the original NFA, i.e., why the variable \( k \)-AC NFA executes the pattern matching correctly.

Let string(\( s_0, s_i \)) denote the sequence of input characters that changes state from \( s_0 \) to \( s_i \) in the AC NFA. Every state \( s_i \) is uniquely labeled by the string string(\( s_0, s_i \)) in the AC NFA [64]. In variable \( k \)-AC NFA, since our variable \( k \)-AC NFA, goto transition function is obtained by concatenating \( k \) consecutive goto transition, so the sequence of input characters transiting \( s_0 \) to any state \( s_i \) is the same as in 1-stride AC NFA, which is the label of the state \( s_i \). Besides, we stop any state transition stride increasing at accepting states, so the output of every state remains unchanged. Therefore, this variable \( k \)-goto transition will not introduce a new state, and we will have exactly the same state set as the 1-stride AC NFA. In 1-stride NFA, the failure function \( f(s_i) = s_j \) if and only if string(\( s_0, s_j \)) is the longest suffix of string(\( s_0, s_i \)) [38], [63].

In variable \( k \)-AC NFA, the state and its label string(\( s_0, s_i \)) are exactly the same as the AC NFA, so the failure function of variable \( k \)-AC NFA is exactly the same as the failure function in the corresponding 1-stride AC NFA, i.e., the failure transition tree and the code for each state in 1-stride AC NFA can be applied without modification.

### VI. Compact Pattern2rule Mapping

In this section, we first describe two strawman solutions that either place a heavy burden on the backend server, or scale poorly with the ruleset size, when accommodating multiple strings in one rule. Then we propose a compact pattern2rule mapping method that achieves a good balance between processing effectiveness and hardware resource usage.

#### A. Strawman Methods

In BOLT, an intuitive approach when identifying a string pattern in the packet payload is to forward the packet to the backend server with complete NIDS functionality, which can carry out a fine-grained payload inspection and determine which patterns and thus which rule the packet triggers. However, naively forwarding any packet containing one pattern in the ruleset can put a heavy burden on the backend server. Due to the high throughput of the switch, forwarding a relatively small part of the traffic may still overwhelm the server processing capability, as Pegasus [18] identifies.

Another approach is to offload the rule-checking function to data plane ASICs, as we did in our prior poster work [65]. As Fig. 8(a) shows, the switch maintains a bit-vector in the metadata for every packet, where each bit stands for the matching status of one pattern. The pattern tables will set the corresponding bits when matching a pattern. At the end of the switch pipeline, we set a rule table, with the bit-vector as the key to determine which rule is hit. However, this method requires a large number of bits in the metadata and the match field, which requires significant hardware packet header vector (PHV) resources in the switching ASICs and is not scalable to the increasing pattern set size. Therefore, for example, there are \( \sim 20K \) string patterns in Snort and Suricata ruleset [66], which require a 20K-bit-long vector in the metadata. Such a design is unfeasible under the current programmable switching ASICs.

#### B. Pattern2rule Mapping Workflow

The strawman bit-vector-based pattern2rule mapping method has an unacceptable memory overhead because it maintains the matching state for every pattern in the ruleset. However, we observe that, compared to the complete pattern set containing thousands of patterns, only quite a few patterns appear simultaneously in a rule and a packet. For example, the Snort ruleset contains thousands of patterns, but no rule has more than 20 patterns. Therefore, for a single input packet, only a small amount of memory is required to maintain the matched patterns. Based on this observation, we design a compact pattern2rule mapping method to check which rule is matched by the incoming packet. As shown in Fig. 8(b), similar to the bit-vector-based strawman solution [65], we set a rule table at the end of the switch pipeline, and store the matching logic for a rule containing multiple patterns in one entry.

Our transformation from rules with multiple patterns into the rule table entries works as follows. Firstly, we assign a binary code to each pattern \( p \) in the pattern set \( \mathbb{P} \) and denote it as code(\( p \)) (the code bit-width is \( O(\lg(|\mathbb{P}|)) \)). We then set \( M \) buckets and allocate each pattern \( p \) a unique bucket ID, id(bucket), i.e., we map each pattern \( p \) to a tuple \((id(bucket), \text{code}(p))\). The distribution of patterns in different buckets should satisfy that any two patterns contained in a single rule must NOT be assigned to the same bucket. For
example, in Fig. 8(b), the patterns “she”, “he”, “his” in rule $r_1$ are distributed into different buckets to avoid collision. This constraint leads to a $k$-SAT problem, which is NP-Complete. We omit the proof process here, and interested readers may refer to Appendix (§A) for details.

According to the pigeonhole principle, the number of buckets $M$ is at least the maximum pattern number in a single rule ($3$ for the case in Fig. 8(b)), i.e., $\max \{ |r_i| \mid r_i \in R \}$, where $R$ is the ruleset, and should not exceed the size of the pattern set $|P|$, because it is a trivial solution when each pattern has its exclusive bucket. Such a value range gives a feasible solution to this NP-C problem. We iterate $M$ over the value range from $\max \{ |r_i| \mid r_i \in R \}$ to $|P|$, and for each $M$, we invoke Z3 [67], a classical SMT solver, to find whether a pattern distribution exists. Once one pattern distribution is found, the Z3 solver stops. The solution exists in the worst case where $M = \max \{ |r_i| \mid r_i \in R \}$.

Then we can generate the table entries according to the bucket distribution and the codes of the patterns. We set $M$ buckets in the metadata as the key of the rule table, and translate each rule into an entry in this table. For each pattern in the rules, we fill the code of this pattern into the corresponding bucket in the match field. After we map each pattern to a bucket, the switch will forward the packet immediately to the backend server instead of continuing inspecting the payload. In this way, BOLT can guarantee an accurate and efficient multi-string rule matching.

VII. IMPLEMENTATION

We implement a prototype of BOLT, including all data plane and control plane features described above. The data plane part is implemented in $\sim 1K$ lines of P4 code, while the control plane part is written in $\sim 3K$ lines of Python code. Our code is publicly available here [33].

In the data plane, the switch parser extracts the payload into a customized header, and the pattern table takes the current NFA state in the metadata and the next $k$ bytes of the packet payload as match fields. Its actions include popping a specific number of byte, changing the current NFA state, and flagging the packet if some patterns get matched [68]. All these actions can be implemented using the primitives defined by P416 specification [68].

In the control plane, the table entry generator module adopts the existing library Pyahocoracisk [69] to construct the AC NFA efficiently, and employ the state encoding (§IV) and the variable $k$-stride transition (§V) to generate the final entries for the pattern tables. For the rule table entries, we first utilize Z3-solver [70] to distribute the patterns into different buckets, and then generate corresponding entries for the rule table according to those distribution and the pattern code (§VI). All the generated table entries are installed into the underlying switch with the interactive APIs provided by the Tofino runtime. The only parameter in BOLT is the maximum stride $k$, which should be predefined by operators in terms of the TCAM constraint of their switches and the number of patterns.

VIII. EVALUATION

Our evaluation mainly focuses on the following questions:

- How efficient and effective is the pattern table entry generating method of BOLT? (§ VIII-B)

Note that pattern2rule table induces no false positive because it is sufficient to prove that a packet truly hits a rule if its bucket contains all the patterns corresponding to a rule, while the inverse proposition does not necessarily hold.
How effective is the rule table entry generating approach that BOLT uses? (§ VIII-C)

How about the performance and the scalability of BOLT? (§ VIII-D)

A. Experimental Setup

We compile the data plane of BOLT with the Barefoot Capilano software suite [29] and deploy it on a 12-stage 6.4Tb/s Barefoot Tofino switch. The controller of BOLT runs on a Dell R730 server, equipped with Intel(R) Xeon(R) E5-2600 v4 CPUs (2.4 GHz, 2 NUMA, each with 6 physical cores and 12 logical cores), 15360K L3 cache, 64G RAM and one Intel XL710 10GbE NIC to connect to the switch. The pattern sets used in our experiments are constructed from the rulesets of Snort 2.9.7.0 and Suricata 5.0 provided by ET-OPEN® [66]. We identify and extract string patterns from the ruleset with the keyword content. The traffic traces in our experiments include CTU-Mixed-Capture-1∼5, CTU-Normal-7 and CTU-Malware-3, which are collected from Stratosphere [71].

B. Pattern Table Efficiency & Effectiveness

To demonstrate the memory efficiency of our entry generating method, we implement three existing schemes discussed in §IV and compare them with BOLT: AC DFA method which has every transition converted into an entry, AC DFA with default actions to omit trivial transitions back to $s_0$, and CompactDFA [59], a typical method for compressing AC DFA. For fair comparison, we choose $k=1$ here. We first count the number of entries generated by BOLT and other three methods under different numbers of patterns. As Fig. 9 shows, compactDFA and BOLT always generate the least number of entries, which is an order of magnitude lower than the AC DFA with default actions and two orders of magnitude lower than the naive AC DFA. Note that Fig. 9 also indicates BOLT will generate the same number of entries as compactDFA, but it does not mean these two schemes occupy the same amount of TCAM memory.

To demonstrate this, we also measure the size of TCAM required by BOLT and other three methods for different numbers of patterns, and the results are shown in Fig. 10. The TCAM memory requirement (i.e., bit number), is the product of the entry number and the width of a state code. BOLT only needs half the TCAM of compactDFA, because the code width required for the BOLT to encode DFA/NFA states is only half of compactDFA. In addition, compared with the other two schemes based on the AC DFA, BOLT just takes up a really small amount of TCAM. Both experiments show that BOLT is able to generate entries efficiently and save precious TCAM memory resources.

To evaluate the effectiveness of the variable $k$-stride transition mechanism in BOLT, we not only count the number of generated entries, but also analyze and compute the average number of characters (average stride) that each stage matches with the Snort pattern set. As shown in Fig. 11(a) and Fig. 11(b), although employing the strawman 5-stride method discussed in §V-A directly increases the average stride to 5, it will introduce several orders of magnitude extra table entries, leading to an unacceptable memory explosion. In contrast, only applying the self-unloop unrolling algorithm will bring very few new table entries, but will increase the average stride greatly. Compared with the only-self-unloop method, our variable $k$-stride transitions do not introduce too many extra entries, but can further increase the average stride to nearly 5 at the same time, which is the theoretical maximum value for 5-stride based methods. To summarize, the variable $k$-stride optimization in BOLT achieves an excellent trade-off between memory usage and throughput gain.

C. Rule Table Effectiveness

To demonstrate the effectiveness of the rule table matching, we first evaluate the volume of the packets forwarded to the backend server (for further fine-grained payload inspection) on the strawman single-pattern method (i.e., forwarding the packet when one pattern in the rule is matched) and BOLT. We use the ratio between the backend server’s receiving packet number and the data plane processing packet number as the metric, i.e., the switch-server emitting ratio. Fig. 12(a) shows the variation of the switch-server emitting ratio for these two
The throughput over traffic with different payload lengths is shown in Fig. 14(b). The effective throughput of BOLT in the first case, where we assume every packet is recirculated, is $T_m = T_x + T_r$, and the throughput in the second case, where we assume only the first packet is recirculated, is $T_r$. The equations for these cases are given by:

$$
T_x = \frac{T_x}{H + P} = \frac{T_r}{H + P - B} = \frac{T_i}{H + P - 2B}
$$

$$
T_r = \frac{T_r}{H + P - 3B} = \frac{T_2}{H + P - 4B} = \frac{T_2}{H + P - 5B}
$$

According to these equations, we can calculate $T_x$, the throughput of the first packet, which is constant, but every pass of ingress/egress pipeline will pop $B$ bytes of payload, so we derive:

$$
pps(T_x) = \frac{T_x}{H + P} = \frac{T_r}{H + P - B} = \frac{T_i}{H + P - 2B}
$$

$$
T_x = \frac{T_r}{H + P - 3B} = \frac{T_2}{H + P - 4B} = \frac{T_2}{H + P - 5B}
$$

$$
T_r = \frac{T_r}{H + P - 3B} = \frac{T_2}{H + P - 4B} = \frac{T_2}{H + P - 5B}
$$

Solving the equations above, the upper bound of input throughput with no packet loss is $T_x = \frac{T_m}{n+1-nB/(H+P)} (n = \left\lceil \frac{P}{2B} \right\rceil - 1)$. Fig. 14(b) shows the maximum effective throughput of BOLT based on the equations above. For the Tofino switch we employed, $T_m = 6.4$ Tbps and the number of stages is 12. This simulation is reasonable because once the P4 program is compiled successfully into the switch pipeline, the switch is guaranteed to run at terabit line rate with bounded memory.

5. Although some programmable switches such as Tofino provides free recirculation bandwidth through dedicated recirculation ports instead of the Ethernet ports, their bandwidth is limited (e.g., 100Gbps in Tofino). And we set a set of Ethernet ports to loopback mode to obtain extra recirculation bandwidth.

6. Tofino has 12 to 20 stages available per pipeline. In practical usage, some stages may be reserved for pattern2rule or other functions like L2 forwarding. To demonstrate the maximum throughput it can achieve theoretically, we assume 12 stages are utilized for pattern matching.
access time [29], [30], [32]. According to this figure, for larger payload and fewer bytes inspected per pass, BOLT requires more recirculations, which would reduce performance super-linearly. Even so, it can still provide ∼1000 Gbps throughput in the case of medium-length payload and medium transition stride.

We also apply different $k$, pattern sets and traffic traces to demonstrate the performance and scalability of BOLT. The first trace is composed of short UDP packets, with a header of 42 bytes and a randomly generated payload of 200 bytes. The second one is a real trace collected from an enterprise sliced evenly, consisting of HTTP packets with a header length of 54 bytes and an average payload length of 1000 bytes. The effective throughput of BOLT over short packets is shown in Fig. 15. BOLT can provide high throughput for short-packet workloads, because they require only a few or even no recirculations and waste negligible bandwidth. Fig. 16 displays the effective throughput of BOLT over large packets. BOLT provides poorer performance on large-packet workloads due to more recirculations. In addition, increasing the stride $k$ can improve the effective throughput of BOLT significantly, and BOLT scales well with pattern sets and the number of patterns.

We also conduct another experiment to explore how the proportion of packets containing patterns influences the performance. To do so, we modify the content of payload to contain some patterns, with the payload remaining 200 bytes. Fig. 17 demonstrates BOLT can still keep a line-rate throughput when injecting more pattern-contained packets.

In a word, BOLT can achieve high throughput performance and scale well with different pattern sets and workloads.

IX. DISCUSSION

A. Further Optimizations

Our encoding-based method makes the entries number equal to the edge number of the trie composed of pattern set, which has much less edges than any AC DFA. For further optimization on entries number, Liu et al. [74] provide another entries compressing method based on the redundancy where transitions share the same source state and destination state, but only character differs. This method works well for regular expression matching table compression, but not so effective in mult-string pattern matching because this kind of redundancy is sparse in string matching. Our experiments shows that this method could reduces the entry number by a few percent. Besides, we can also split a large ruleset into multiple smaller ones, as PPS does [32], which is orthogonal to our work and left as future work.

B. Multiple Correlated Strings in One Rule

Our method can determine whether a packet matches a rule containing multiple string patterns. However, some security applications may employ multiple correlated string to express one attack signature (rule). For example, a rule may require 1 exists while 000 does not appear. This requires that programmable switches should support relational operations (e.g., ‘or’, ‘and’, ‘not’) for multiple strings, which is challenging to be efficiently achieved on the restricted computational model of programmable switches. The bit-vector-based method, as we illustrated in § VI-A and Fig. 8(a), can support such a requirement but scale poorly with the ruleset size, and we leave the support of more flexible correlations between multiple patterns in programmable switching ASICs as our future work.

C. Limitations of Our Bucket-Based pattern2rule Mapping Method

There still remain a few limitations in our pattern2rule mapping method. First, distributing patterns in $K$ buckets is actually a $K$-SAT problem. Theoretically, it does not always have solutions in polynomial time and may have some approximate solutions with some patterns within a rule allocated to one bucket. A carefully crafted ruleset may make the problem unsolvable, resulting in a large bucket number $K$ and a high memory cost. Fortunately, in our experiments, the Z3 solver can always give a valid bucket distribution solution in minutes for both Snort and Suricata rule set, each with thousands of rules. We believe that the bucket distribution is feasible for common rule sets in practice. Another problem is the bucket overwriting which introduces false negatives, as we mentioned in § VI-B. Although BOLT avoids this false negative problem by forwarding the overwriting packets to the backend server, this may still overwhelm the capability of the backend server when the traffic volume of forwarding packets is large. Worse yet, attackers may exploit this vulnerability by crafting packets to degrade the performance of the entire system. We leave the detailed exploration of these two problems as our future work.
Snort) or regular expressions (“pcre” field in Snort). BOLT can not only support offloading the rules with only string pattern signatures but also act as a prefilter for the rules with more than string patterns, such as regular expressions [18]. Besides, although BOLT is devoted to security applications in this paper, it can also be used in many other fields. For information retrieval and data analytics applications [75], pattern matching is used to locate the occurrences of user-defined strings (e.g., words, phrases) in text, which also dominates the performance of the entire system. BOLT can be leveraged to improve the throughput of these applications as well.

X. RELATED WORK

Besides the most relevant works discussed in the main text, our work is also inspired by the following topics.

A. NIDS/NIPS Acceleration

There have been numerous efforts to scale up pattern matching performance with dedicated hardware. Barker et al. [76], Mitra et al. [77], and Pighas [18] leverage FPGA to achieve high throughput string or regex pattern matching. MiDeA [43] and Kargus [9] demonstrate that a single GPU-enhanced server can achieve a much higher throughput (e.g., 40 Gbps for Kargus [9]). Liu et al. [78] and DeepMatch [79] accelerate string pattern matching as high as 40 Gbps utilizing NPU. In contrast, we focus on offloading multi-string pattern matching on an emerging type of network hardware, programmable switching ASICs, which enables flexible and performant network function offloading compared to traditional ASICs [80], [81], [82]. PPS [32] first offloads the AC algorithm onto programmable switches to carry out string pattern matching. However, PPS utilizes a straightforward DFA compiling scheme with high memory costs (subset construction and k-stride closure [83]). It takes no consideration of co-existing patterns either, making it difficult to identify which patterns and thus which rule a packet matches.

B. DFA Compression

A large number of studies have been proposed in recent years to optimize DFA-based pattern matching. The core goal is to minimize the memory footprint while guaranteeing high performance. Some existing efforts re-encode the state set $S$ [41] or alphabet $\Sigma$ [84], [85], [86], [87] to merge similar states or input symbols, and reduce the scale of the transition table with a size of $O(|S| \cdot |\Sigma|)$. In addition to the encoding schemes, some studies have been conducted to compress the transitions, leveraging the redundancy in transitions. For example, many efforts utilize bitmap [40], [41], [58], [87], [88] to remove the redundant transitions and reduce memory footprint. However, the methods above still require multiple times of memory access for a single input character, resulting in a lower average stride per stage and finally degrading the maximum throughput. Furthermore, some studies transform consecutive symbols into a single transition [89], [90], or propose default transitions [61], [91], [92], [93] to compress the transition redundancy. These methods do not work well on programmable switches because the consecutive transitions are sparse in string matching and the default transitions require multiple memory accesses, resulting in lower efficiency on programmable switches. There is also a considerable amount of work on efficient pattern matching, such as partitioning the rule set [94], [95], [96], [97], or designing new FA variants [47], [98], [99], [100], [101], [102], [103], [104], [105], [106]. We omit the details and suggest interested readers refer to Xu et al. [107] for a more comprehensive survey.

C. Programmable Switch

BOLT builds on the recent trends that leverage programmable switches to accelerate various network applications. Programmable switches have been utilized to accelerate load balancing [31], [108], key-value storage [51], coordination service [109], distributed deep learning training [110], [111], network monitoring or telemetry [53], [112], [113], network scanning [114] and DDoS defense [27], [52]. They deliver significant performance improvement with lower costs. BOLT focuses on a different problem: multi-string pattern matching. To this end, we design various techniques to translate string patterns into the match-action entries, increase the throughput with acceptable memory costs, and accommodate rules with multiple co-existing strings with data plane match-action tables.

XI. CONCLUSION

In this paper, we highlight the challenges that current multi-string pattern matching faces in dealing with the high-speed large-volume network traffic today, and identify the opportunities that programmable switches provide to resolve such issues. To this end, we present BOLT, a scalable and cost-efficient multi-string pattern matching system that overcomes several constraints of the computational model and memory resources of programmable switches. In particular, we design an efficient state encoding scheme to fit a large number of rules into the limited memory on programmable switches, a variable k-stride transition mechanism to increase the throughput significantly with acceptable memory costs, and a compact pattern2rule mapping method to support multiple co-existing strings in one rule. We implement an open-source prototype of BOLT and conduct extensive evaluations. These evaluations show that BOLT offers orders of magnitude improvements in throughput and scales well with various rulesets and workloads.

APPENDIX A
PROOF OF THE BUCKET ALLOCATION

We give a formal proof that the bucket allocation problem in § VI is a $k$-SAT problem.

First, we denote a rule containing a few patterns ($p_i$) as $r = \{p_1, p_2, \ldots, p_r\}$. A rule set, certainly, can be marked as $R = \{r_1, r_2, \ldots, r_s\}$, $r_i = \{p_{i_1}, p_{i_2}, \ldots, p_{i_{|r_i|}}\}$, where $|r_i|$ is the size of $r_i$, i.e., the number of patterns in $r_i$. For a rule set $R$, we can get a corresponding pattern set $P = \{p_1, p_2, \ldots, p_n\}$, composed of the patterns in the rules of $R$. As we discussed above, we want to allocate each $p_i \in P$ into $m$ buckets in a way that any two patterns present in each rule can not be assigned in the same bucket (according to the pigeonhole principle, $M$ should satisfy $\max\{|r_i| \mid r_i \in R\} \leq M \leq |P|$). Namely, the allocation requirement can be formalized as follows. An allocation partitions the pattern set $P$ into $m$ buckets, where each bucket $B_m$ is a set of patterns $B_m = \{p_{m_1}, p_{m_2}, \ldots, p_{m|B_m|}\}$, and $|B_m|$ is the pattern number in
\( C_{default} = (x_{11} \Rightarrow \neg x_{12}) \land (x_{11} \Rightarrow \neg x_{13}) \land \cdots \land (x_{11} \Rightarrow \neg x_{1K}) \land (x_{12} \Rightarrow \neg x_{11}) \land (x_{12} \Rightarrow \neg x_{13}) \land \cdots \land (x_{12} \Rightarrow \neg x_{1K}) \land (x_{13} \Rightarrow \neg x_{11}) \land (x_{13} \Rightarrow \neg x_{12}) \land \cdots \land (x_{13} \Rightarrow \neg x_{1K}) \land \cdots \land (x_{1K} \Rightarrow \neg x_{11}) \land (x_{1K} \Rightarrow \neg x_{12}) \land \cdots \land (x_{1K} \Rightarrow \neg x_{1(K-1)}) \land (x_{11} \lor x_{12} \lor \cdots \lor x_{1K}) \land \cdots \land (x_{N1} \Rightarrow \neg x_{N2}) \land (x_{N1} \Rightarrow \neg x_{N3}) \land \cdots \land (x_{N1} \Rightarrow \neg x_{NK}) \land \cdots \land (x_{NK} \Rightarrow \neg x_{N1}) \land (x_{NK} \Rightarrow \neg x_{N2}) \land \cdots \land (x_{NK} \Rightarrow \neg x_{N(K-1)}) \land (x_{N1} \lor x_{N2} \lor \cdots \lor x_{NK}) \)  

(7)

\( C_i = (x_{i1} \Rightarrow \neg x_{i2}) \land (x_{i1} \Rightarrow \neg x_{i3}) \land \cdots \land (x_{i1} \Rightarrow \neg x_{i_{r_i}-1}) \land (x_{i2} \Rightarrow \neg x_{i3}) \land \cdots \land (x_{i2} \Rightarrow \neg x_{i_{r_i}-1}) \land \cdots \land (x_{i_{r_i}-1} \Rightarrow \neg x_{i_{r_i}}) \land (x_{i_{r_i}-1} \Rightarrow \neg x_{i_{r_i}}) \land (x_{i_{r_i}} \Rightarrow \neg x_{i_{r_i}-1}) \land (x_{i_{r_i}} \Rightarrow \neg x_{i_{r_i}-1}) \)  

(8)

Hence, each row of the matrix has one and only one element of 1 (a pattern can be placed in only one bucket) and the rest are 0. Therefore, a default constraint can be obtained as (7), shown at the top of the page. According to (6), each rule \( r_i \) can raise a corresponding constraint \( C_i \), (8), as shown at the top of the page.

Therefore, whether a valid bucket allocation exists is abstracted into a boolean satisfiability problem whether there exists an assignment for the boolean variables in matrix \( A_{N \times K} \) satisfying the following constraint.

\( C = ( \bigwedge_{1 \leq i \leq |B|} C_i) \land C_{default} \)  

(9)

As shown in (7) and (8), each constraint formula has been expressed as a conjunctive normal form because each implication formula \( p \Rightarrow q \) is equivalent to \( p \lor \neg q \). And we can identify that the largest clause\(^7\) (which has the most variables) lies in \( C_{default} \) (7), denoted by \((x_{i1} \lor x_{i2} \lor \cdots \lor x_{iK}), i = 1, 2, \ldots, N\), and it obviously has \( K \) variables. Therefore, such a problem can be reduced to a \( K \)-SAT problem, and when \( K \geq 3 \), becomes an NP-C problem.

REFERENCES


\(^7\)A clause is a disjunction (logical OR) of variables or their negations.


Shicheng Wang received the B.S. degree in computer science from Tsinghua University, China, where he is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace. His research interests include cyber security and programmable data plane.

Menphao Zhang received the B.S. and Ph.D. degrees in computer science from Tsinghua University, China, in 2016 and 2021, respectively. He is currently a Joint Post-Doctoral Researcher with Tsinghua University and Kuaishou Technology. His research interests include programmable networks, high-performance networks, and network security.

Guanyu Li received the B.S. degree from the School of Computer Science and Technology, Huazhong University of Science and Technology, China. He is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include software-defined networking, network function virtualization, and cyber security.

Chang Liu received the B.S. degree from the School of Computer Science and Technology, Beijing Institute of Technology, China. He is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include software-defined networking, programmable data plane, and cyber security.

Zhiliang Wang (Member, IEEE) received the B.E., M.E., and Ph.D. degrees in computer science from Tsinghua University, China, in 2001, 2003, and 2006, respectively. He is currently an Associate Professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include formal methods and protocol testing, next generation internet, network measurement, and network security.

Ying Liu (Member, IEEE) received the M.S. degree in computer science and the Ph.D. degree in applied mathematics from Xidian University, China, in 1998 and 2001, respectively. She is currently a Full Professor with Tsinghua University, China. Her major research interests include network architecture design, next generation internet architecture, routing algorithm, and protocol.

Mingwei Xu received the B.S. and Ph.D. degrees from Tsinghua University. He is currently a Full Professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include computer network architecture, high-speed router architecture, and network security.