Making Multi-String Pattern Matching Scalable and Cost-Efficient with Programmable Switching ASICs

Shicheng Wang*, Menghao Zhang*°, Guanyu Li*, Chang Liu*, Ying Liu*, Xuya Jia†, Mingwei Xu*°
*Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University
°Department of Computer Science and Technology, Tsinghua University †Huawei Technologies Co., Ltd

***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 works have been devoted to alleviate 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 of the network bandwidth and network traffic today. In this paper, we present BOLT, a scalable and cost-efficient multi-string pattern matching system leveraging the capability of emerging programmable switches. BOLT combines the following two techniques, a smart state encoding scheme to fit a large number of strings into the limited memory on the programmable switch, and a variable k-stride transition mechanism to increase the throughput significantly with the same level of memory costs. We implement a prototype of BOLT and make its source code publicly available. Extensive evaluations demonstrate that BOLT could provide orders of magnitude improvement in throughput which is scalable with pattern sets and workloads, and could also significantly decrease the number of entries and memory requirement.**

I. INTRODUCTION

Multi-string pattern matching serves as a fundamental building block for many network security applications, especially network intrusion/prevention systems (NIDS/NIPS) [1], [2], web application firewalls (WAF) [3], application identification systems [4] and some network censorship/surveillance systems [5], [6]. 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 [7], [8].

Existing works often alleviate this bottleneck with algorithm optimization [9], [10], [11], [12], [13] or GPU/FPGA/NPU acceleration [8], [14], [15], [16]. However, neither these software-improved nor hardware-accelerated solutions provide the desired scalability or costs that catch up with the drastic increase of the network bandwidth and network traffic today. Recently the network bandwidth at traffic aggregation points in regional ISPs has already reached muti-100s of Gbps [17], [18]. Many network device providers [19], [20] and standard organizations [21] are embracing the era of 400Gbps bandwidth [20], [22]. These high network bandwidths require that security applications' ability to maintain network monitor should also keep pace with such a traffic volume. However, even when we fully exploit the potential of CPUs on servers, it is impossible for a pattern matching engine to reach ∼10 Gbps within a single server [11], [12]. While GPU/FPGA/NPU-enhanced servers can reach a higher throughput (i.e., at most 100Gbps under state of the art [23]) with these hardwares' inherent parallelism [24], [10], [13], 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 costs and the management complexity significantly [25], [26].

We observe that the emerging programmable switching ASICs [27] in network community provide an unprecedented opportunity to bridge this gap. One single programmable switch could easily process multi-Tbps traffic at line rate, which has several orders of magnitude higher throughput than highly-optimized servers. Even for FPGA/GPU/NPU-enhanced servers, there is still significant gap to match such a throughput. Besides, these programmable switches allow programmers to leverage domain-specific languages (e.g., P4 [28]) to use underlying hardware features (e.g., a pipeline of user-defined match-action tables) directly, with the same level of power consumption and capital costs with regular switches [29]. 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 in the programmable switch (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 ruleset of Snort has ∼4000 rules), while the memory resource in the switch is pretty limited (50-100MB [30]). Simply translating string pattern rules into a Deterministic Finite Automaton (DFA) and then the corresponding match-action entries, as PPS [31] 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 computation model in programmable switches is quite restricted compared with x86 CPUs. In particular, current programmable switch cannot support iterations and loops, which is a key component in the algorithms of the pattern matching. 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 [31], but it will incur the explosion of entries.

To address these problems, we propose BOLT, a system for matching multiple string patterns with programmable switches. First, BOLT develops a fast and efficient state encoding scheme, to fit a large number of rules into the limited memory in the programmable switch. Second, BOLT proposes a variable k-stride transition mechanism, to enlarge the throughput significantly with acceptable entry number increase. We implement a prototype of BOLT in Barefoot Tofino [29], and make the source code publicly available here [32][1]. Extensive evaluations show BOLT could provide orders of magnitude improvement in throughput which is scalable with pattern sets and workloads, and it could also significantly decrease the number of entries and memory requirement.

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 switching ASCIs (§II).
- We propose BOLT, a scalable and cost-efficient multi-string pattern matching system with programmable switching ASICs (§III). We design a smart encoding scheme and a variable k-stride transition mechanism to overcome the restrictions posed by the memory resources and the computational model of programmable switches (§IV, §V).
- We implement an open-source prototype of BOLT, and conduct extensive evaluations to show the advantages of BOLT (§VII).

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

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 the programmable switching ASICs.

A. Multi-String Pattern Matching

Multi-string pattern matching is a well-known research area that has been extensively studied during the past few decades. It can be formally defined that, given a text string $T = t_1...t_n$, and a pattern string set P, where every element $P_i = p_1...p_m$ means a predefined pattern string and each t_i or p_i belongs to the alphabet Σ, the pattern matching algorithm should output the set of all positions where P_i stands as a substring in T. The Aho-Corasick (AC) algorithm is an efficient way [33] for multi-string pattern matching. 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 becomes the *de facto* standard for the pattern matching, and is widely used in many state-of-the-art network security applications, such as Snort [1] and Suricata [34].

[1]Due to the non-disclosure agreement with Barefoot, we re-implement a BMv2 version and make it open-source.

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 works have been devoted to alleviate this performance bottleneck either by algorithm optimization or by hardware acceleration. Software-based algorithm optimization techniques attempt to either minimize the memory usage [35], [36], [37] or increase the number of characters per transition [9], [10], [13], achieving several times larger throughput. However, the packet processing performance on software is intrinsically limited, because CPUs on servers are not specialized for high-speed packet processing. For example, even with highly-optimized servers [12], it is still impossible for a pattern matching engine to reach 20Gbps efficiently. Although we could achieve higher throughput by deploying more servers, doing so would increase the capital and operational costs drastically, 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 the dedicated hardware to accelerate the pattern matching has also attracted great attention. GPUs get popular for the pattern matching tasks because of their high parallelism compared with CPUs. The single instruction multiple threads (SIMT) architecture could efficiently execute an algorithm in parallel, thus providing the throughput up to ∼40Gbps [8], [38], [39]. FPGA-based solutions also exploit the parallelism to accelerate multi-string pattern matching and even regular expression matching [14], [16], [40], achieving high throughput. However, it is still impossible for these hardware alternatives to match the performance or the costs of programmable switches. Worse yet, these hardware alternatives are usually attached to servers through PCIe, which makes it difficult to fully explore their potentials because of the limited PCIe bandwidth.

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

C. Opportunities by Programmable Switches

Programmable switches are an emerging networking technology that provides hardware programmability without compromising performance. 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 and egress pipeline respectively. Each stage is a packet processing unit, with its dedicated resources, including match-action tables, registers and stateful ALUs.

Match-action tables match certain header fields or metadatas in ternary/priority, and perform customized actions, e.g., modifying headers/metadatas, reading/writing registers. Stateful ALUs support customized calculation based on headers/metadatas/registers. Registers store states to support stateful packet processing. With the programmable switching ASICs, operators can use domain-specific languages (e.g., P4 [28]) to customize the data plane logic. 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.

The programmable switching ASICs and P4 language make it straightforward to implement a customized terabit packet processing device, as long as the defined logic satisfies the computational model and resource constraints of a programmable switch. Moreover, programmable switches have the similar level of power consumption and capital costs as traditional fixed-function switches, which enables orders of magnitude cost reduction compared to commodity CPU or other hardware alternatives (e.g., GPU, FPGA, NPU)[2]. These unique characteristics provide unprecedented opportunities for next-generation scalable and cost-efficient multi-string pattern matching.

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 recirculate the packet for deeper payload inspection, thus discarding the inspected portion. So when deploying in-line, an additional buffer mechanism is required to avoid information loss [42], [43], and we leave the integration of such works into BOLT as our future work.

We assume the incoming packet consists of an Ethernet header, an IP header, a UDP/TCP header and the payload to inspect[3]. The packet payload can be encoded in any pattern (ASCII, UTF-8 or binary data) and we adopt ASCII in our paper, where the alphabet has 256 elements encoded in 1 byte.

B. Workflow

The workflow of BOLT is illustrated in Fig. 1. Operators first need to define a list of matching rules from the signature database (e.g., Snort community rules [44]). Then the controller extracts *string patterns* from the matching rules, constructs an NFA for these patterns using the AC algorithm [33], and translates the AC NFA into underlying match-action table entries. Our key enabler here is a fast and 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 enlarge the average transition stride and the average throughput, with acceptable increasing of entry number (§V).

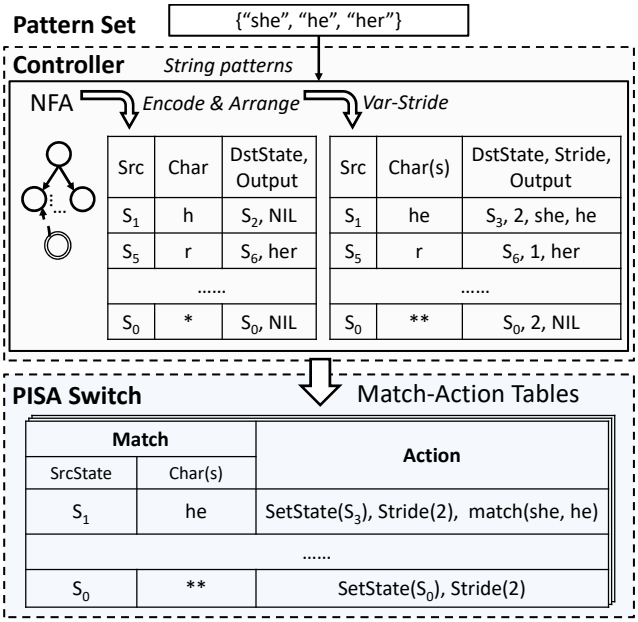


Fig. 1. BOLT overview and workflow.

With these efficient match-action tables, the data plane conducts matching for every packet byte by byte in the pipeline. If any predefined pattern is matched, the data plane will carry out the corresponding action, such as dropping, passing or alerting.

IV. Pattern Table Entry Transition

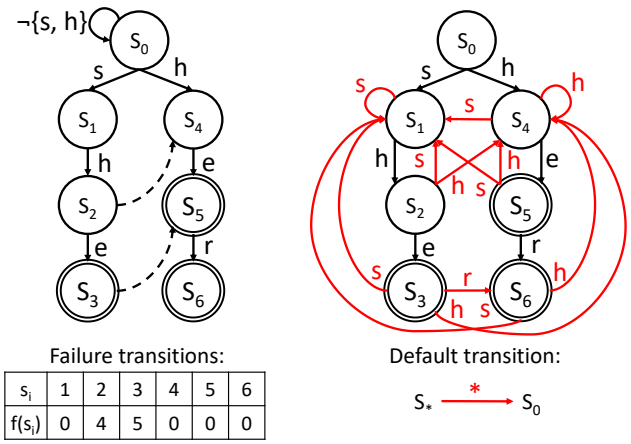
In this section, we analyze the shortcomings of existing DFA-based entry generation methods, and give our observation that the feature of match-action tables (i.e., ternary match and entry priority) helps implement the AC NFA efficiently in the programmable switch. We then elaborate our approach to translate the AC NFA into match-action table entries below.

A. Problem Analysis

To achieve multi-string pattern matching on hardware, a typical method [24], [31] is to (1) construct an NFA 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 [33], which has much more transitions than its corresponding NFA. Fig. 2(a) shows the NFA for matching {*she*, *her*, *he*} built by the AC algorithm, where the solid lines denote *goto* transitions and the

[2]The cost-efficiency of packet processing on programmable switches has been verified by numerous other recent works [41], [26], as a result, we do not illurate more in this paper.

[3]In fact, we can take any layer header as payload.



s_i	1	2	3	4	5	6
f(s_i)	0	4	5	0	0	0

(a) AC NFA (b) AC DFA

Fig. 2. AC NFA and AC DFA.

dashed lines denote *failure* transitions (failure transitions to s_0 are omitted for clarity). Its corresponding DFA is shown in Fig. 2(b), and we also omit the trivial transitions returning to the root state s_0. From Fig. 2, we can see that the DFA has much more transition edges than the NFA, which requires much 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 we have seen many existing works [24], [45], [46] attempting to compress AC DFAs to save memory, it is still hard for these compression algorithms to achieve an optimal condition, which would inevitably lead to table entry increase and resource efficiency degradation.

We observe that the unique features of match-action tables in the programmable switch, including the *ternary match* and the *priority entry*, provide us an unique opportunity to directly translate the AC NFA into match-action table entries efficiently. However, the matching logic of an NFA is hard to be implemented in the matching semantic of match-action tables, because it requires iterations and loops, which is quite difficult to support in the data plane. To illustrate this, we highlight the key difference between the DFA matching and the NFA matching here: in the AC 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 one 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. 2(b), 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. 2(a) 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_5 along its *failure* transition, which has a *goto* transition

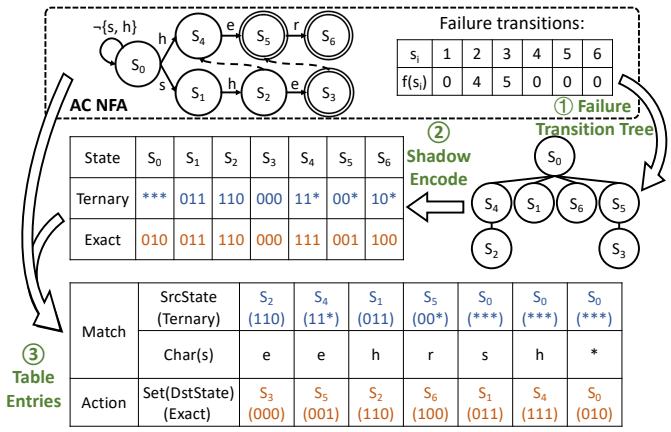


Fig. 3. BOLT state encoding and table entry generation procedure.

matching *r* and going to s_6. Therefore, this input is consumed by the *goto* transition of s_5.

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 not only match itself, but also the states who 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 encode the states and to generate the table entries, as depicted in Fig. 3: (1) construct a *failure transition tree* denoting the deferment relationship defined by *failure* transitions. (2) encode the state with a shadow encoding scheme, which assign each state a ternary code in the match field and an exact code in the action field. (3) assign different priority for each entry to achieve complete semantics of the AC NFA. Details are illustrated below.

First of all, 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. 3 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 finally moves to s_0 according to the failure transition table, which is determined by the construction process of *failure transition* [33].

Second, we encode each state with two codes, a *ternary code* and an *exact code*, based on the *failure transition tree*.

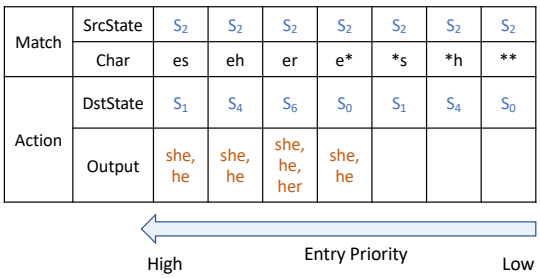


Fig. 4. 2-stride DFA table for s_2.

For each node, the *ternary code* should cover the *exact code* of itself and also the descendant nodes in the *failure transition tree*. To achieve this, we build our encoding scheme on a classic shadow encoding [45] 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) [47], 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 same to the transitions from q, making this transition deferred to q to be executed. The output of this algorithm assigns each state a binary *exact code* and a *ternary code*, to make the exact state code of p will 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 Hufman coding style algorithm, the shadow encoding algorithm provides a unique signature to distinguish different states, while increases the state code width negligibly.

Finally, we convert each *goto transition* into a table entry. We assign priorities for the entries in 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. 3 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. 3, 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 works that increase the stride size of transitions add significant memory costs, and then 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 [31], [48]. Although this method can lower the base of exponential increase of transition number[4], 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. 2(b), concatenating $\delta_1(s_2, e) = s_3$ 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 $\delta_1(s_2, *) = s_0$ to concatenate with its successive state, as Fig. 4 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 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. 4, both input "$e*$" ($e\neg\{s,h,r\}$) and "$**$" ($\neg\{e\}\neg\{s,h\}$) lead a transition from s_2 to s_0, but state transition with input "$e*$" will output matching for pattern "she" and "he" while "$**$" not. We need to allocate a new state for every possible combination of accepting states in the path [31], [45], causing the state explosion. Alicherry et al. [24] propose a highly-optimized DFA which has variable stride length, but it still has much memory usage resulted from additional states, and the average stride is small because there could be negative stride in some cases. Backwards packet inspection is also difficult to be implemented 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. [49] 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 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.

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 [24], [45], [31], whereas in NFA-based methods, this requires two tables (i.e., two stages in programmable switch) to handle a single transition [49]. Both methods lead to unnecessary resource

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

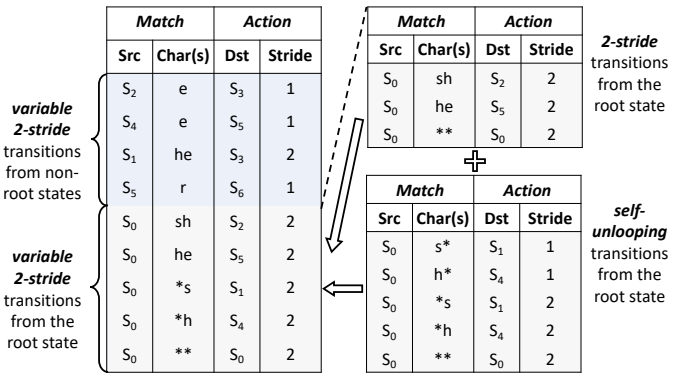


Fig. 5. Variable 2-stride table.

waste for programmable switches.

B. Variable k-Stride Transitions

In this subsection, we propose a variable k-stride AC NFA, to increase the average throughput, while ensuring space usage increasing 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 could 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 *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 could compute the k-stride *goto* transitions iteratively from $(k-1)$-stride *goto* transitions. The left side in Fig. 5 shows the variable 2-stride *goto* transition from the AC NFA in Fig. 2(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 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 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. 5 shows. However, this table could not carry out the pattern matching correctly. For example, the input $xher$ could not get matched and the pattern her will miss. Meiners et al. [45] propose *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 increasing of number entries. However, this method does not increases the stride of transitions evenly. As the lower right part in Fig. 5 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.

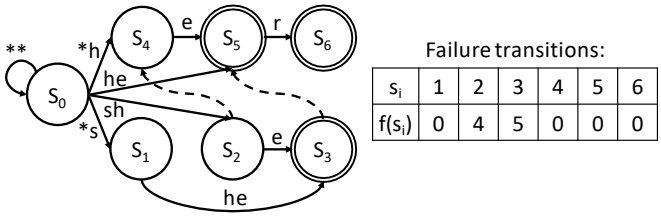


Fig. 6. Variable 2-stride AC NFA.

To conclude, 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. 5. 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 entry number, scaling linearly with the stride k. The lower left side in Fig. 5 illustrates a 2-stride self-loop unrolling table for s_0. In these two steps, we get all the variable k-stride *goto* transitions. In particular, Fig. 6 shows the final variable 2-AC NFA derived from Fig. 2(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 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 [50]. In variable k-AC NFA, since our variable k-stride *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 new state, we 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 $string(s_0, s_i)$ [33], [49]. In variable k-AC NFA, the state and its label $string(s_0, s_i)$ is 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. Implementation

We implement a prototype of BOLT, including all data plane and control plane features described above. The data plane part is implemented in ∼1K lines of P4 code for the Barefoot Tofino ASIC, while the control plane part is written in ∼2K lines of Python code. Our code is publicly available here [32].

In the data plane, the match-action table takes the current NFA state in the metadata and the next k bytes of the packet payload as match fields. Its actions contain popping bytes of specific length, changing the current NFA state, and flagging the packet if some patterns get matched. All are implemented with the built-in primitives of programmable switches. When deploying these tables into the switch, we replicate them across all the stages to increase the throughput, as PPS [31] does.

In the controller, the table entry generator module adopts the existing library Pyahocoracisk [51] 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 these tables. 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 stride k, which should be predefined by operators in terms of the TCAM constraint of their switches and the number of patterns.

VII. Evaluation

Our evaluation mainly focuses on the following questions:

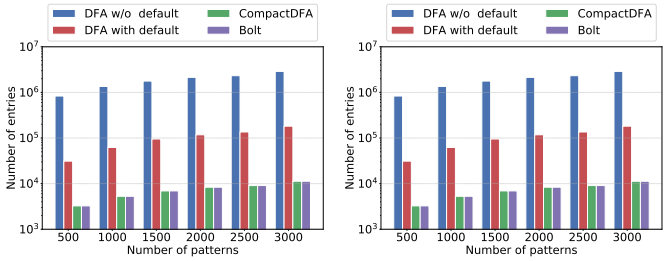
- How efficient is the entry generating method of BOLT in saving memory?
- How effective is the variable k-stride transition optimization that BOLT adopts?
- How about the performance and the scalability of BOLT?

A. Experimental Setup

We compile the data plane of BOLT with Barefoot Capilano software suits [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 logic 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® [52]. We identify and extract string patterns from the ruleset with the keyword *content*.

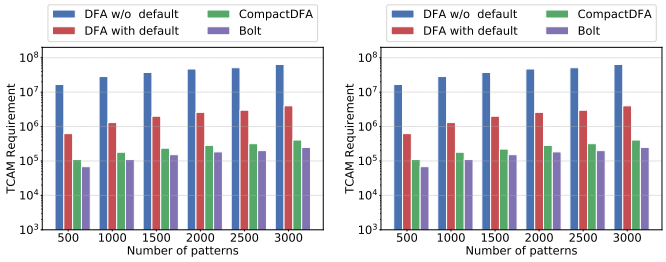
B. Entry Generating Efficiency

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, AC DFA with default actions, and CompactDFA [46] which compresses AC DFA. For fair comparison, we choose k=1 here. We first count the number of entries generated by BOLT and other three



(a) Snort 2.9.7.0 (b) Suricata 5.0.0

Fig. 7. Entry number on different pattern sets.



(a) Snort 2.9.7.0 (b) Suricata 5.0.0

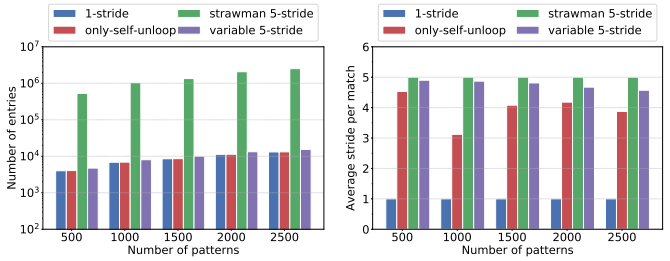
Fig. 8. TCAM requirement on different pattern set.

methods under different numbers of patterns. As Fig. 7 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. 7 also indicates BOLT will generate the same number of entries with 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 results are shown in Fig. 8. As we can see, BOLT only needs half TCAM of compactDFA, and this is 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 BOLT is able to generate entries efficiently and save precious TCAM memory resources.

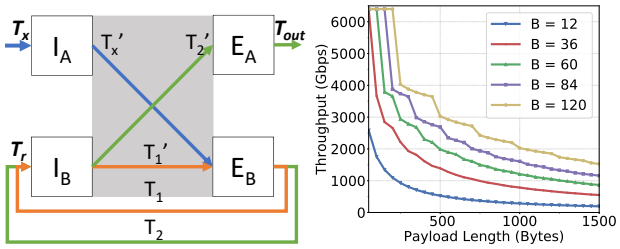
C. Variable k-Stride Effectiveness

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. 9(a) and Fig. 9(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 more extra table entries, leading to unacceptable memory waste. In contrast, only applying self-unloop unrolling algorithm just bring very few new table entries, but will increase the average stride greatly. Compared with only-self-unloop method, our variable k-stride transitions do not bring too many extra entries, but can further increase



(a) Entry number of variable 5-stride AC NFA (b) Average stride of variable 5-stride AC NFA

Fig. 9. Entry number and average stride.



(a) Simulation scenario (b) Simulation throughput

Fig. 10. Throughput over traffic with different payload.

the average stride to close to 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.

D. Performance and Scalability

To demonstrate the performance and the scalability of BOLT, ideally we should measure the highest pattern matching throughput that one Tofino switch [29] could provide, and how this throughput scales to different pattern sets and traffic traces. However, restricted by the capability of the traffic generator in our lab, we cannot fully cover the bandwidth of the switch. Therefore, we simulate the upper limit of throughput of BOLT, under different pattern sets and workloads.

We assume BOLT can inspect B-byte payloads each time the packet passes an ingress/egress pipeline, and B is the product of the average stride for NFA matching tables and the number of pipeline stages. To inspect more bytes in the payload, BOLT recirculates the packet to pass the pipeline again, leading to extra bandwidth occupation. Obviously, the effective throughput of BOLT is affected by the times of recirculation each packet requires. Assuming the packet header is H bytes, and the payload is P bytes, we can obtain the recirculation number for the packet, denoted as n, by the inequality $2nB < P \leq 2(n+1)B$, and thus $n = \lceil P/2B \rceil - 1$. To recirculate packets in a switch, we set some ports in loopback mode, where all packets will only be bounced into the ingress pipeline (i.e., recirulation ports) [53]. Assuming that a programmable switch provides T_m throughput, we set T_r throughput for recirculation, and the remaining ports offer T_x throughput for external traffic. Obviously, $T_x + T_r = T_m$. Fig. 10(a) shows an illustration of recirculation for $n = 2$ times. Packets recirculating for the first and the second time will compete for bandwidth of reciuculating ports. We observe that egress B is the bottleneck in switch whose pipeline throughput is the first to be exhausted. When egress B arrives at the maximum throughput, we get $T_r = T_x^{'} + T_1^{'}$. At this time, the pps (packets per second) of traffic is constant, but every pass of ingress/egress pipeline will pop B bytes of payload, so we derive:

$$pps(T_x) = \frac{T_x}{H+P} = \frac{T_x^{'}}{H+P-B} = \frac{T_1}{H+P-2B} = \frac{T_1^{'}}{H+P-3B} = \frac{T_2}{H+P-4B} = \frac{T_2^{'}}{H+P-5B}.$$

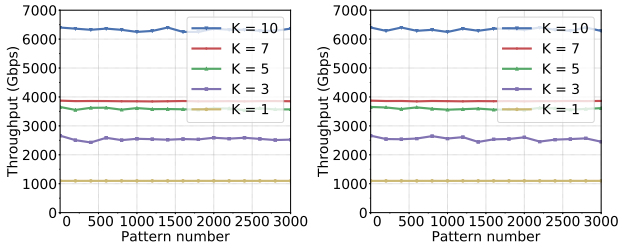
According to the equations above, we can calculate T_x, the maximum throughput a switch can provide, with no packet drop in internal pipelines. So, generally, when recirculating for n $(n \geq 1)$ time(s), the following equations can be obtained:

$$\begin{cases} pps(T_x) & = \frac{T_x}{H+P} = \frac{T_x^{'}}{H+P-B} = \frac{T_i}{H+P-2iB} \\ & = \frac{T_i^{'}}{H+P-(2i+1)B} \ (i = 1, 2, \ldots, n-1) \\ T_r & = T_x^{'} + \Sigma_{i=1}^{n-1}(T_i^{'}) \\ T_m & = T_x + T_r \end{cases} \quad (1)$$

Solving the equations above, the upper bound of input throughput with no packet loss is $T_x = \frac{T_m}{n+1-n^2B/(H+P)}$ $(n = \lceil \frac{P}{2B} \rceil - 1)$. Fig. 10(b) shows the maximum effective throughput of BOLT based on the equations above. For the Tofino switch we employed, Th_m is 6.4Tb/s 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 access time [28], [29], [31]. According to the 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.

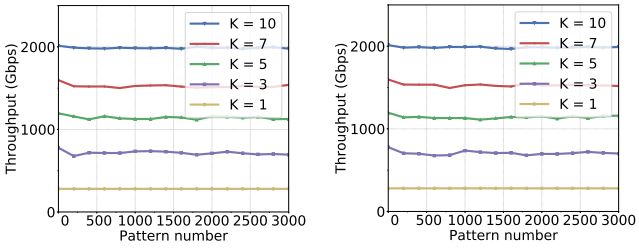
We also apply different k, pattern sets and traffic traces to demonstrate the performance and scalability of BOLT. The first trace is composed of short UDP packets, with a header of 42 bytes and a randomly generated payload of 200 bytes. The second one is a real trace collected from an enterprise sliced 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. 11. BOLT can provide high throughput for short-packet workloads, because they require only a few or even no recirculations and waste negligible bandwidth. Fig. 12 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. 13 demonstrates BOLT can still keep a line-rate throughput when



(a) Snort 2.9.7.0 (b) Suricata 5.0.0

Fig. 11. Throughput over short-packet traffic.



(a) Snort 2.9.7.0 (b) Suricata 5.0.0

Fig. 12. Throughput over large-packet traffic.

injecting more pattern-contained packets.

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

VIII. DISCUSSION

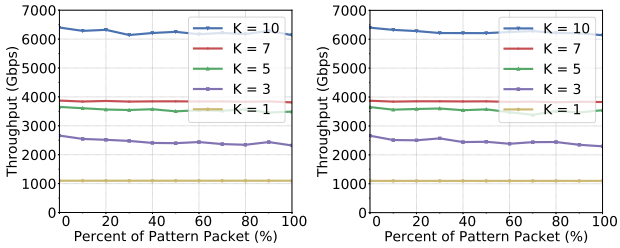
Further optimizations. Our encoding-based method makes the entries number equal to the edge number of the trie composed of pattern set, which has much less edges than any AC DFA. For further optimization on entries number, Liu et al. [54] provide another entries compressing method based on the redundancy 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 rule set into multiple smaller ones, as PPS does [31], which is orthogonal to our work and left as future work.

Extensibility of BOLT. Although multi-string pattern matching is devoted to security applications in this paper, it can also be used in many other fields. For information retrieval and text-editing applications, the pattern matching is used to locate the occurrences of user-defined string (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.

IX. RELATED WORK

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

NIDS/NIPS acceleration. There are many works that attempt to accelerate NIDS/NIPS with dedicated computation hardware, such as FPGA, GPU, NPU and ASICs. Barker et al. [55] and Mitra et al. [56] leverage FPGAs to accelerate the Snort NIDS. MIDeA [57] and Kargus [8] demonstrate that a single GPU-enhanced servers can achieve a much higher throughput. Liu et al. demonstrate NPU-enhanced servers can also significantly enhance the throughput [58]. Tan et al. propose a new ASIC design to accelerate NIDS [59]. However, as we discussed in §II, none of them provides the scalability or the costs that can compete with programmable switches.



(a) Snort 2.9.7.0 (b) Suricata 5.0

Fig. 13. Throughput on traffic with different pattern-packet percentage.

Programmable switch. BOLT builds on the recent trends that leverage programmable switches to accelerate various applications in networking [30], [60], [61], distributed systems [62], [63] and security [64], [26], [65], [66], but focuses on a different problem: multi-string pattern matching. To this end, we also design various techniques to translate string patterns into the match-action entries and increase the throughput with acceptable memory costs.

X. 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 a smart state encoding scheme to fit a large number of rules into the limited memory on the programmable switch and a variable k-stride transition mechanism to enlarge the throughput significantly with acceptable memory costs. 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 pattern set size/type and workload traffic.

ACKNOWLEDGMENT

We sincerely thank anonymous reviewers for their valuable comments, and also would like to thank Yangyang Wang from Tsinghua University, Hongxin Hu from Clemson University, Guofei Gu from Texas A&M University, Ang Chen from Rice University for joining some discussions of this paper. This work is supported in part by the National Key R&D Program of China (2017YFB0801701, 2018YFB1800405) and the National Science Foundation of China (No. 61625203, No. 61772307, No. 61832013). Menghao Zhang and Ying Liu are the corresponding authors.

REFERENCES

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

[2] T. Z. Project, "Zeek," https://zeek.org/, 2020.

[3] Trustwave, "Modsecurity," https://modsecurity.org/, 2020.

[4] ntop, "nDPI," https://www.ntop.org, 2020.

[5] Stanford, "How the great firewall works," https://cs.stanford.edu/people/eroberts/cs201/projects/international-freedom-of-info/china_2.html, 2020.

[6] E. F. Foundation, "How the nsa's domestic spying program works," https://www.eff.org/nsa-spying/how-it-works, 2020.

[7] S. Antonatos and et al., "Generating realistic workloads for network intrusion detection systems," in *WOSP*, 2004, pp. 207–215.

[8] M. A. Jamshed and et al., "Kargus: a highly-scalable software-based intrusion detection system," in *CCS*, 2012, pp. 317–328.

[9] L. Vespa and et al., "Ms-dfa: Multiple-stride pattern matching for scalable deep packet inspection," *The Computer Journal*, vol. 54, no. 2, pp. 285–303, 2011.

[10] X. Wang and et al., "Kangaroo: Accelerating string matching by running multiple collaborative finite state machines," *JSAC*, vol. 32, no. 10, pp. 1784–1796, 2014.

[11] B. Choi and et al., "{DFC}: Accelerating string pattern matching for network applications," in *NSDI*, 2016, pp. 551–565.

[12] X. Wang and et al., "Hyperscan: a fast multi-pattern regex matcher for modern cpus," in *NSDI*, 2019, pp. 631–648.

[13] E. Sadredini and et al., "Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching," in *HPCA*. IEEE, 2020, pp. 86–98.

[14] C. R. Clark and et al., "Scalable pattern matching for high speed networks," in *FCCM*. IEEE, 2004, pp. 249–257.

[15] R. Sidhu and et al., "Fast regular expression matching using fpgas," in *FCCM*. IEEE, 2001, pp. 227–238.

[16] D. Sidler and et al., "Accelerating pattern matching queries in hybrid cpu-fpga architectures," in *ACM MOD*, 2017, pp. 403–415.

[17] H. Dreger and et al., "Operational experiences with high-volume network intrusion detection," in *CCS*, 2004, pp. 2–11.

[18] V. Stoffer and et al., "100g intrusion detection," *LBL*, 2015.

[19] Cisco, "High capacity 400g data center networking," https://www.cisco.com/c/en/us/solutions/data-center/high-capacity-400g-data-center-networking/index.html, 2020.

[20] NVIDA, "Nvidia mellanox bluefield-2 dpu," https://www.mellanox.com/files/doc-2020/pb-bluefield-smart-nic.pdf, 2020.

[21] "IEEE Standard for Ethernet - Amendment 10: Media access control parameters, physical layers, and management parameters for 200 Gb/s and 400 Gb/s operation," *IEEE Std 802.3bs-2017*, pp. 1–372, 2017.

[22] Accton, "The new world of 400 gbps ethernet," https://www.accton.com/Technology-Brief/the-new-world-of-400-gbps-ethernet/, 2020.

[23] Z. Zhao and et al., "Achieving 100gbps intrusion prevention on a single server," in *OSDI*, 2020, pp. 1083–1100.

[24] M. Alicherry and et al., "High speed pattern matching for network ids/ips," in *ICNP*. IEEE, 2006, pp. 187–196.

[25] S. K. Fayaz and et al., "Bohatei: Flexible and elastic ddos defense," in *USENIX Security*, 2015, pp. 817–832.

[26] M. Zhang and et al., "Poseidon: Mitigating volumetric ddos attacks with programmable switches," in *NDSS*, 2020.

[27] P. Bosshart and et al., "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[28] ——, "P4: Programming protocol-independent packet processors," *SIGCOMM CCR*, vol. 44, no. 3, pp. 87–95, 2014.

[29] B. Networks, "Tofino: World's fastest p4-programmable ethernet switch asics," https://barefootnetworks.com/products/brief-tofino/, 2020.

[30] R. Miao and et al., "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *SIGCOMM*, 2017, pp. 15–28.

[31] T. Jepsen and et al., "Fast string searching on pisa," in *SOSR*, 2019, pp. 21–28.

[32] S. Wang, "Bolt," https://github.com/wangshicheng1225/BOLT, 2020.

[33] A. V. Aho and J. D. Ullman, *The theory of parsing, translation, and compiling*. Prentice-Hall Englewood Cliffs, NJ, 1973, vol. 1.

[34] Open Information Security Foundation, "Suricata: Open Source IDS," https://suricata-ids.org/, 2020.

[35] N. Tuck and et al., "Deterministic memory-efficient string matching algorithms for intrusion detection," in *IEEE INFOCOM*, vol. 4. IEEE, 2004, pp. 2628–2639.

[36] M. Becchi and et al., "Memory-efficient regular expression search using state merging," in *IEEE INFOCOM*. IEEE, 2007, pp. 1064–1072.

[37] J. Yu and et al., "Memory efficient string matching algorithm for network intrusion management system," *Tsinghua Science and Technology*, vol. 12, no. 5, pp. 585–593, 2007.

[38] N. Jacob and et al., "Offloading ids computation to the gpu," in *ACSAC*, 2006, pp. 371–380.

[39] G. Vasiliadis and et al., "Parallelization and characterization of pattern matching using gpus," in *IISWC*. IEEE, 2011, pp. 216–225.

[40] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *CoNEXT*, 2007, pp. 1–12.

[41] Y. Tokusashi and et al., "The case for in-network computing on demand," in *EuroSys*, 2019, pp. 1–16.

[42] D. Kim and et al., "Tea: Enabling state-intensive network functions on programmable switches," in *SIGCOMM*, 2020, pp. 90–106.

[43] S. Goswami and et al., "Parking packet payload with p4," *arXiv preprint arXiv:2006.05182*, 2020.

[44] Cisco, "Snort," https://www.snort.org/, 2020.

[45] C. R. Meiners and et al., "Fast regular expression matching using small tcams for network intrusion detection and prevention systems," in *USENIX Security*, 2010, pp. 8–8.

[46] A. Bremler-Barr and et al., "CompactDFA: Generic state machine compression for scalable pattern matching," in *INFOCOM*. IEEE, 2010, pp. 1–9.

[47] S. Kumar and et al., "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *SIGCOMM CCR*, vol. 36, no. 4, pp. 339–350, 2006.

[48] C.-C. Chen and et al., "An efficient multicharacter transition string-matching engine based on the aho-corasick algorithm," *TACO*, vol. 10, no. 4, pp. 1–22, 2013.

[49] S. Yun, "An efficient tcam-based implementation of multipattern matching using covered state encoding," *IEEE Transactions on Computers*, vol. 61, no. 2, pp. 213–221, 2010.

[50] A. V. Aho and et al., "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[51] W. MuÅĆa, "Pyahocorasick 1.4.0," https://pypi.org/project/pyahocorasick/, 2019.

[52] E. T. Rule, "Emerging threats open ruleset," https://rules.emergingthreats.net/OPEN_download_instructions.html, 2020.

[53] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch asic," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, 2019, pp. 141–149.

[54] A. X. Liu and et al., "Tcam razor: A systematic approach towards minimizing packet classifiers in tcams," vol. 18, no. 2. IEEE, 2009, pp. 490–500.

[55] Z. K. Baker and et al., "Time and area efficient pattern matching on fpgas," in *FPGA*, 2004, pp. 223–232.

[56] A. Mitra and et al., "Compiling pcre to fpga for accelerating snort ids," in *ANCS*, 2007, pp. 127–136.

[57] G. Vasiliadis and et al., "Midea: a multi-parallel intrusion detection architecture," in *CCS*, 2011, pp. 297–308.

[58] R.-T. Liu and et al., "A fast string-matching algorithm for network processor-based intrusion detection system," *TECS*, vol. 3, no. 3, pp. 614–633, 2004.

[59] L. Tan and et al., "A high throughput string matching architecture for intrusion detection and prevention," in *ISCA*. IEEE, 2005, pp. 112–122.

[60] S. Narayana and et al., "Language-directed hardware design for network performance monitoring," in *SIGCOMM*, 2017, pp. 85–98.

[61] A. Gupta and et al., "Sonata: query-driven streaming network telemetry," in *SIGCOMM*. ACM, 2018, pp. 357–371.

[62] X. Jin and et al., "Netcache: Balancing key-value stores with fast in-network caching," in *SOSP*, 2017, pp. 121–136.

[63] ——, "Netchain: Scale-free sub-rtt coordination," in *NSDI*, 2018, pp. 35–49.

[64] R. e. a. Meier, "Nethide: Secure and practical network topology obfuscation," in *USENIX Security*, 2018, pp. 693–709.

[65] Q. Kang and et al., "Programmable in-network security for context-aware byod policies," in *USENIX Security*, 2020.

[66] G. Li and et al., "NetHCF: Enabling line-rate and adaptive spoofed ip traffic filtering," in *ICNP*. IEEE, 2019, pp. 1–12.