Fast Multi-string Pattern Matching using PISA

Shicheng Wang, Chang Liu, Ying Liu, Guanyu Li, Menghao Zhang, Yangyang Wang, Mingwei Xu
Tsinghua University

CCS CONCEPTS

• **Security and privacy** → *Network security*; • **Networks** → *Network algorithms*;

KEYWORDS

String Matching; Programmable Switch; PISA

ACM Reference Format:
Shicheng Wang, Chang Liu, Ying Liu, Guanyu Li, Menghao Zhang, Yangyang Wang, Mingwei Xu. 2019. Fast Multi-string Pattern Matching using PISA. In *The 15th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '19 Companion), December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3360468.3368184

1 INTRODUCTION

Multi-string pattern matching serves as a fundamental build block for many network security applications, especially network intrusion/prevention systems (IDS/IPS) [2], web application firewalls (WAF) [8], and application identification systems [6]. It typically expresses certain attack signatures (rules) with multiple strings, which are then used to examine whether the payload of a packet matches any of the predefined rules. Unfortunately, this often becomes a bottleneck of these systems because every byte of the packet has to be scanned by a large ruleset of strings.

Existing works often alleviate this bottleneck with algorithm improvement [1] or GPU/FPGA acceleration [3, 7]. However, neither these software-improved nor hardware-accelerated approaches provide the desired performance that catches up with the dramatic increase of the network bandwidth and network traffic today (e.g., from multi-10s of Gbps to multi-100s of Gbps). We argue this performance gap lies in the performance capability of the hardware for acceleration, and the powerful programmable switch provides an unprecedented opportunity to bridge this gap. After all, the programmable switch is able to easily reach multi-Tbps throughput within a single box, which further reduces the per unit capital expense significantly.

Despite this vision being promising, implementing multi-string pattern matching on the programmable switch (i.e., Protocol Independent Switch Architecture (PISA)) is non-trivial. A recent work, PPS [4], has already demonstrated the feasibility of achieving fast string searching on PISA at ~Tbps. However, we identify two essential questions which remain unanswered. First, current security applications usually require a large set of rules (e.g., the latest community ruleset of Snort has ~4000 rules), which also indicates that the translated deterministic finite automaton (DFA) would also be large. Simply mapping the DFA into the match action tables as PPS [4] requires an extremely large number of match action entries, which are challenging to be stored in the limited memory of the programmable switch. Second, many security applications also require correlated strings to express attack signatures (rules). For example, Figure 1 shows two simplified snort rules (sid: 25355~25336) where only "content" fields are saved. Rule #1 requires "POST", "<NAME>" both exist while "</NAME>" does not appear in the packet, and rule #2 requires "POST", "<URL>" both exist while "</URL>" does not appear. This requires that the programmable switch should support relational operations for multiple strings, which is challenging to be efficiently achieved on the restricted computational model of the programmable switch.

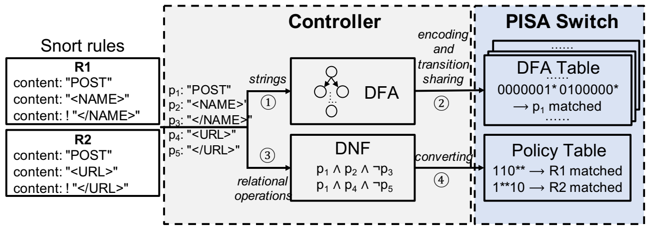


Figure 1: Bolt architecture and workflow.

To address these problems, we propose Bolt, a fast multi-string pattern matcher with the programmable switch. We utilize the *transition sharing* to compress pattern entries to fit them into the limited memory on the programmable switch (§2.1). And we design a *policy table* at the end of the switch pipeline to express relational operations among multiple strings (§2.2). Finally, we give a preliminary implementation and evaluation to demonstrate the effectiveness of Bolt (§3).

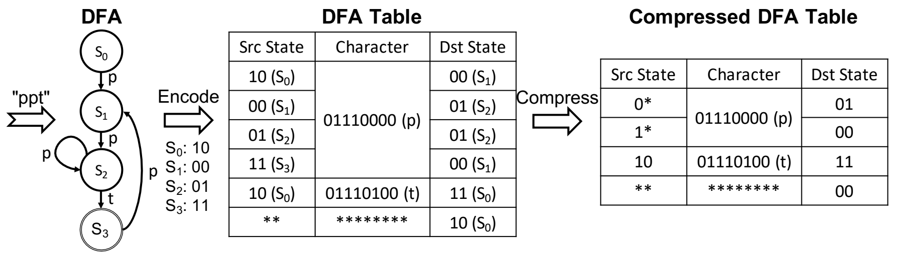
2 BOLT DESIGN

The overall architecture and workflow of Bolt are shown in Figure 1. Bolt controller extracts the strings and the relational operations among strings from the given pattern rules separately, and translates them into different match action tables. And the data plane of Bolt conducts pattern matching with *DFA* match action tables and *policy* match action table. Details to achieve the compact DFA tables and the effective policy table are elaborated in §2.1 and §2.2.

2.1 Minimizing the entry number

Bolt employs Aho-Corasick algorithm to construct an equivalent DFA for given strings, and then transforms the DFA into match action tables, similar to PPS [4]. During the mapping from the DFA

*Supported by National Key R&D Program of China (2018YFB1800405 and 2017YFB0801701) and National Science Foundation of China (No.61872426 and No.61772307).



DFA Table

Src State	Character	Dst State
10 (S_0)	01110000 (p)	00 (S_1)
00 (S_1)		01 (S_2)
01 (S_2)		01 (S_2)
11 (S_3)		00 (S_1)
10 (S_0)	01110100 (t)	11 (S_0)
**	*********	10 (S_0)

Compressed DFA Table

Src State	Character	Dst State
0*	01110000 (p)	01
1*		00
10	01110100 (t)	11
**	********	00

Figure 2: Bolt compression procedure.

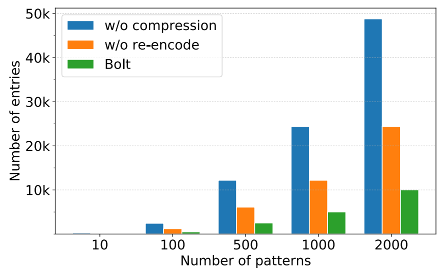


Figure 3: Compression efficiency.

to the match action entries, we observe that many transition edges have the same input character and destination state, and they only differ in their source states. This implies that we can share some transitions to reduce the transition number and save the switch memory.

However, directly merging multiple transitions is not always feasible. Take the DFA in Figure 2 as an example, although the state S_0 and S_3 have the same transitions with "p" to S_1, they can not be merged if we choose *"00"* and *"11"* to encode them separately. But if we encode S_0 with *"10"* instead of *"00"*, then the state S_0 and S_3 can be represented by state *"1*"*, and the transitions (*"10"*, "p", S_1) and (*"11"*, "p", S_1) can be merged into a single transition (*"1*"*, "p", S_1). To achieve an efficient state encoding, we define two transitions with the same input character and the destination state as a *redundancy pair.* To re-encode each state, we first construct a complete graph for all states where the weight of each edge is the number of redundancy pairs between the edge's two vertices. Then we calculate the maximum spanning tree for this complete graph, so a node in this tree is likely to have many redundancy pairs with its father/children nodes. Finally, we use codes with the same prefix to re-encode nodes with descendent/ancestor relations. After the state encoding, the next step is to seek out an effective compressing solution to produce the minimum number of transitions. We observe that sharing transitions always have the same input character, so we group re-encoded DFA match action entries according to the input character, and then employ the classic dynamic programming method in routing table compression [5] to obtain an effective compression for each group. We do not claim our scheme will get the optimal compression rate, only that they are empirically effective and can get a reasonable compression result. We will seek the optimal algorithm in future.

2.2 Correlated multi-string matching

To achieve multi-string matching with complex relational operations, our high level idea is to transform those operations in one rule into an entry (entries) in the policy table, as shown in Figure 1. First, Bolt represent each rule in Boolean algebra as a Boolean expression, in which each Boolean variable indicates whether the corresponding string should be matched for this rule. Second, each obtained Boolean expression is normalized to a more compact Disjunctive Normal Form (DNF), which is a disjunction of multiple conjunctions. To accommodate these DNFs, we put them in the policy table whose match fields represents the Boolean variables of the DNFs: each entry in the policy table represents one conjunction in the DNF, and the relationship between different entries also corresponds to the disjunctive relations of different conjunctions. Therefore, the DNF is finally transformed into one entry (entries), the number of which is the same as that of conjunctions in the DNF. Since any Boolean expression can be converted to a DNF, all pattern rules containing AND, OR, and NOT relations can be eventually converted into entries of the policy table on the programmable switch.

At runtime, for each packet, Bolt maintains a bit vector to store the Boolean variables, which indicates whether the corresponding strings are matched. Once a single string is found in the payload of the packet, Bolt sets the corresponding bit in its bit vector. When completing the detection for the whole payload of the packet, Bolt match the policy table with the full bit vector, to see whether the packet contains some rules. On the programmable switch, the bit vector of the packet is stored in the metadata.

3 EVALUATION AND FUTURE WORK

We implement an open-source Bolt prototype with 400 lines of P4 code for the data plane and 500 lines of Python code for the controller[1]. We select "content" patterns from snort rules and join them with relational operations as the original rule. As depicted in Figure 3, comparing with PPS [4], Bolt reduces the number of entries by ~80%, which makes it possible to fit 2000 patterns within 10K match-action entries in one hardware switch. Besides, Bolt can function normally under correlated multiple strings in one rule, which also demonstrates the effectiveness of our policy table. In future, we plan to further optimize the compression algorithm, implement Bolt on the hardware Tofino switch and conduct more evaluations in real-world scenarios.

REFERENCES

[1] Byungkwon Choi and et al. 2016. {DFC}: Accelerating String Pattern Matching for Network Applications. In *NSDI*. 551–565.
[2] Cisco. 2019. Snort. https://www.snort.org/.
[3] Muhammad Asim Jamshed and et al. 2012. Kargus: a highly-scalable software-based intrusion detection system. In *CCS*. ACM, 317–328.
[4] Theo Jepsen and et al. 2019. Fast String Searching on PISA. In *SOSR*.
[5] Alex X Liu and et al. 2010. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *TON* 18 (2010).
[6] ntop. 2019. nDPI. https://www.ntop.org.
[7] Reetinder Sidhu and Viktor K Prasanna. 2001. Fast regular expression matching using FPGAs. In *FCCM*. IEEE, 227–238.
[8] Trustwave. 2019. ModSecurity. https://modsecurity.org/.

[1]https://github.com/LamperougeWang/P4PatternMatch