A Robust Approach for Matching Mixed Case-sensitive and Case-insensitive Patterns

Hongbin Lu
Dept. of Computer Sci. & Tech.
Tsinghua University
Beijing, China
lu-hb02@mails.tsinghua.edu.cn

Kai Zheng
IBM China Research Lab
Beijing, China
zhengkai@cn.ibm.com

Bin Liu
Dept. of Computer Sci. & Tech.
Tsinghua University
Beijing, China
liub@tsinghua.edu.cn

Changhua Sun
Dept. of Computer Sci. & Tech.
Tsinghua University
Beijing, China
sch04@mails.tsinghua.edu.cn

Abstract— **As one of the key methods as well as a bottleneck for Network Intrusion Detection Systems (NIDSes) to detect and eliminate malicious traffic, pattern matching is increasingly gaining popularity while also faces threats from hackers' overloading attempts. The support of mixed case-sensitive and case-insensitive patterns, which is essential for NIDSes to detect possible attacks targeting different applications and operating systems, is currently a potential vulnerability since the widely used *Convert-Search-Verify* (CSV) approach encounters severe performance degradation in the worst-case scenarios. This paper firstly gives a thorough analysis on the reasons causing jams in the worst case, and then boosts up the performance by leveraging a novel mechanism named Convert-Search-incrementally-Verify (CSiV). CSiV differs from CSV in that it first merges possible case-sensitive matches to suspicious segments in the "Search" phase, and then leverages an Aho-Corasick like algorithm to verify them. The infeasibility of the simple *Double Search* (DS) approach is also explained by analyzing its low average-case throughput. Extensive experiments based on real pattern sets along with both collected and artificial traffic traces show that, the performance of the proposed approach outperforms the DS approach by a factor of 2 in the ordinary cases, and is better than the CSV approach up to 5 times under the worst-case scenario, indicating both its feasibility and robustness for a worst-case safe NIDS.**

Keywords-network security, intrusion detection, pattern matching, case insenstive, worst-case safe, incremental verification

I. INTRODUCTION

With billions of US dollars lost annually caused by the malicious attacks of/from the Internet [1], such as DDOS-attack, viruses/worms-spread, and spam, etc., network security is becoming a serious problem necessary to be solved with no time to delay. As one of the key methods for Network Intrusion Detection Systems (NIDSes) to detect and eliminate malicious traffic, Pattern Matching and analysis over network traffic, are increasingly gaining popularity nowadays.

Usually, NIDSes require inspecting on-the-fly large volume of Internet traffic against a huge rule set containing tens of thousands of patterns/virus-signatures [2, 3]; furthermore, to avoid crash or being blocked by unexpected/unwieldy workloads, e.g. generated by hackers intentionally, NIDSes are also required to cope with the worst-case scenarios, which is extremely headachy to handle.

Note that the support of mixed Case-Sensitive (CS) and Case-Insensitive (CI) patterns is one of the compulsory/necessary features enabling NIDSes to adapt to the various network contents coming from different parts of the Internet and different kinds of applications. For instance, commands and filenames on Microsoft Windows platforms are CI while those over UNIX/LINUX are CS, which means that NIDSes must support mixed CS and CI patterns to detect possible attacks targeting different platforms.

However, unfortunately, there is still no satisfactory solution for mixed CS and CI patterns up to now. The most straightforward solution is the *CI-to-CS Translation* approach, the idea of which is to convert all the CI patterns into equivalent CS ones and therefore turn the problem into a CS-only one. For example, CI pattern "ab" will be converted to 4 equivalent CS patterns "ab", "Ab", "aB", and "AB". However, apparently, such an approach suffers from the storage explosion issue, since any *n*-byte CI pattern has to be translated into 2^n CS ones, which makes it actually infeasible for ordinary occasions handling thousands of long patterns.

To avoid pattern replications, another approach, which is called the *Double Search* (*DS*) approach, is to treat CI and CS patterns separately that the patterns are pre-partitioned into a CI subset and a CS one, and then all the traffic data would be matched against these two subsets, respectively. As a consequence, there would be an around-50% deterministic performance degradation caused by the extra payload scan in the ordinary cases.

Different from the DS approach, the *Convert-Search-Verify* (*CSV*) approach, applied in a famous open-source NIDS *Snort* [2], realizes CI-CS-mixed pattern matching by sacrificing the worse-case performance instead of the average performance. Firstly, all patterns together with the network traffic are converted into a capitalized copy, which is used to find the CI matches and possible CS matches; whenever a possible CS match occurs, the matched string would be further verified with the original CS patterns for final judgment. Compared with the

This work is supported by NSFC (No. 60373007, 60573121 and 60625201), the Cultivation Fund of the Key Scientific and Technical Innovation Project, Ministry of Education of China (No. 705003), the Specialized Research Fund for the Doctoral Program of Higher Education of China (No. 20040003048 and 20060003058), China/Ireland Science and Technology Collaboration Research Fund (2006DFA11170), and the Tsinghua Basic Research Foundation (JCpy2005054).

DS approach, the CSV approach performs fairly well for ordinary case, when the CS matches seldom occur; however, for the extreme/worst case when most patterns are CS and matches occur for each and every sliding window of the traffic, the performance may even degrade by an order of magnitude.

The contributions of this paper are two-folds. On one hand, we look inside the problem of the CSV approach and find out that, actually, there are too many redundant transactions in its "Verify" phase, especially under the worst-case scenarios; on the other hand, based on this observation, we proposed an optimized *Convert-Search-incrementally-Verify* (*CSiV*) method which first merges possible case-sensitive matches to suspicious segments in the "Search" phase, and then employs an Aho-Corasick [4] (AC) like algorithm to verify them. Thus the "Verify" phase is speeded up by leveraging the temporary results of the previous matches, which in return reduces the number of transactions for verification. According to the experimental results based on real pattern sets and traffic traces collected from the real world or artificially generated for worst-case evaluation, the performance of the CSiV approach outperforms that of DS by a factor of 2 in the ordinary cases, and is better than the CSV approach up to 5 times under the worst-case scenario, with little average performance lost.

The remainder of this paper is organized in the following way. Section II first analyzes the drawbacks of the DS and CSV approaches, and then presents the idea, implementation, and analysis of CSiV in turn; Section III shows the experimental results as well as the comparisons with the existing schemes; Section IV concludes the paper and presents future work.

II. DESIGN CONCEPT

A. Definitions

Before describing the idea and implementation, we give a few definitions here to avoid clutter:

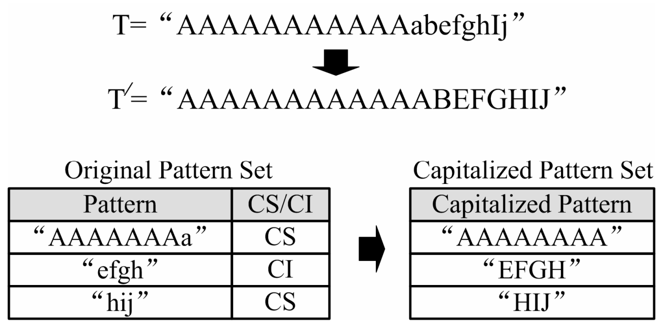
- *String*: a sequence of characters;
- *Pattern*: a sequence of characters, along with a property "CS/CI" specifying its case sensitivity;
- *Pattern Matching*: the operations to search the given string, determine whether there are occurrences of any pattern(s) in the given pattern set, and then report their positions if any. For example, given a string "AAAAAAAAAAAabefghIj" and a pattern set consisting a CI pattern <"efgh", CI> along with 2 CS patterns <"AAAAAAAa", CS> and <"hIj", CS>, the pattern matching engine must report 3 matches: <"AAAAAAAa", CS > at Byte 5~12, <"efgh", CI> at Byte 14~17, and <"hIj", CS> at Byte 17~19.

B. The Basic Idea

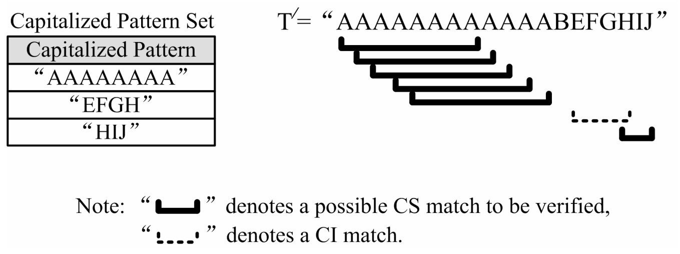
As described in Section I, the *DS* approach splits the pattern set into the CS and CI subsets, and deploys sequential searches within them, respectively, to realize the capability of mixed CS and CI pattern matching. Apparently, such approach would in return result in deterministic performance halved, and therefore we consider it not that feasible to base on. Note that for the ordinary scenarios, matches actually seldom occur and the addition scan usually results in vain.

On the counterpart side, the CSV approach suffers from the uncertainty of search time and sharp performance drop for the worst-case scenarios. The following example demonstrates the idea of the CSV approach as well as its drawbacks in nature. Given that the example in Section II.A is used here. Initially, in the "Convert "stage (Fig. 1(a)), all patterns are capitalized and formed a pattern set for filtering. And correspondingly, the input string *T* is capitalized as well.

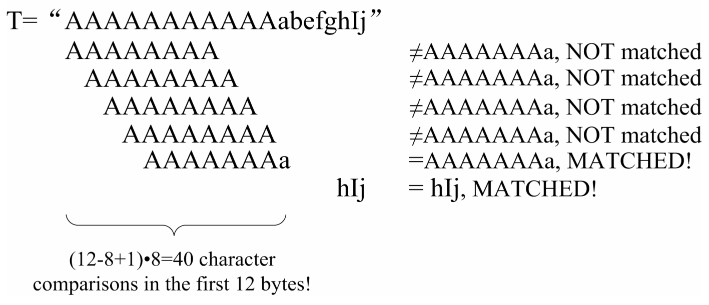
Then in the "Search" stage (Fig. 1(b)), string T' is matched against the capitalized pattern set for CI and potential CS matches. Whenever a match occurs, if it comes out to be a CI one, a final match is reported; otherwise, i.e. a CS-pattern is matched, the third stage, the "Verify" stage (Fig. 1(c)) is launched for a further check, and the corresponding suspicious sub-strings in *T* would be matched against the corresponding suspicious pattern in the original pattern set, character by character.



(a) Convert



(b) Search



T= "AAAAAAAAAAAabefghIj"
AAAAAAAA ≠AAAAAAAa, NOT matched
AAAAAAAA ≠AAAAAAAa, NOT matched
AAAAAAAA ≠AAAAAAAa, NOT matched
AAAAAAAA ≠AAAAAAAa, NOT matched
AAAAAAAa =AAAAAAAa, MATCHED!
hIj = hIj, MATCHED!
(12-8+1)•8=40 character comparisons in the first 12 bytes!

(c) Verify

Figure 1. An illustration of the CSV approach.

Apparently, it is the "Verify" stage that leads to the problem of the CSV approach. Note that the workload of the approach, or more precisely the workload of the "Verify" stage, heavily depends on the number of matches occurring in the "Search" stages. And consider the worst-case scenario when matches are found for each and every byte within the string (e.g. in Fig 1(b), the consecutive 'A's in T'), $(M - L_{Patt} + 1) \bullet L_{Patt}$ per-byte-verification are required, where M denotes the length of the string, and L_{Patt} denotes the length of the suspicious pattern.

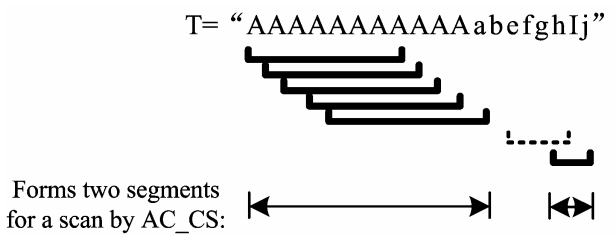
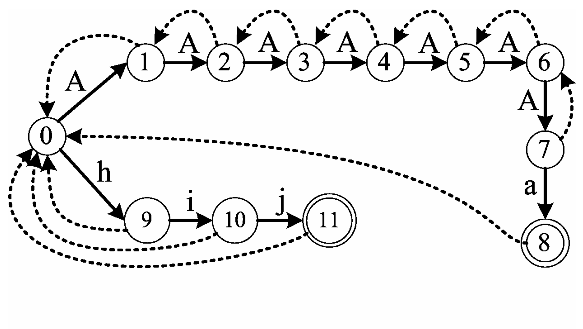


Figure 2. A worst-case scenario of the CSV approach.



Note: dot line denotes failure pointer.

Figure 3. The AC_CS state machine constructed for the pattern set in Fig. 1.

As we step into the details, we find that most of the per-byte-verifications, especially in the worst-case scenarios, overlap with each other, which are redundant and can be avoided. For instance, in the example shown in Fig.1(c), the verifications for Bytes 1~8 in T' against Pattern <"AAAAAAAa", CS> overlaps with that for Bytes 2~9, which means that the verifications for Byte 2~8 against Pattern <"AAAAAAAa", CS> would be redundantly launched; theoretically speaking, at most $(L_{Patt} - 1)$ redundant verifications would be launched for most bytes, which indeed results in the performance degradation for the CSV approach in the extreme case.

Based on the above observation, the idea of the proposed CSiV approach is to try erasing the overlapping verifications and therefore avoiding unnecessary overheads. For the "Search" stage, verification would NOT be launched immediately whenever CS-pattern match occurs, as that in the CSV approach. Instead, the CSiV approach introduces a novel concept called "suspicious segment" to represent a large suspicious part within the string, and process the whole segment in a batch way, using an AC-like algorithm instead of per-byte verification. More specifically, a "suspicious segment" is a consecutive sub-string of T, which is reported by the "Search" stage and contains possible CS patterns needed to be verified. A suspicious segment is denoted by a <starting pos, ending pos> pair that indicates its beginning and ending positions in T; it is actually a part within the string with overlapping bytes for verifications. For instance, Byte 1~12 in T, of the shown example, forms a suspicious segment which contains 5 possible overlapping CS-matches; the whole segment, i.e. "AAAAAAAAAAAa" would be reported by the "Search" stage and sent to the "Verify" stage to perform CS-verification with an AC-like state-machine. No redundant operation, therefore, would be launched under this framework.

C. Implementation and Analysis

1) The dedicated AC state machine for CS patterns

As mentioned in Section II.B, The CSiV approach differs from the CSV one in that it first finds out the suspicious segments and then leverages an AC-like algorithm to verify them. The constructing procedure of the AC state machine (named AC_CS in this paper) is detailed in [4]. For instance, the 3-pattern pattern set used in Fig.1 translates to an AC_CS state machine illustrated in Fig. 3.

Briefly, upon receiving an input character, the AC_CS state machine updates its state by first trying to find the corresponding transition from the current state, and if that fails then following the failure pointer. [4] proved that the AC algorithm has a deterministic performance: it costs at most $2M$ state transitions for an AC state machine to scan a string with length M, independently with the pattern set is used. Thus, if we perform the verifications in a batch fashion, the number of character comparisons will be bounded by $2M$.

2) Procedure of incremental verification

The procedure of incremental verification consists of two sorts of operations: forming suspicious segments and calling the AC_CS state machine for verification. Though it is more straightforward to form all the segments first and verify them one by one, extra memory allocations are required for storing all starting and ending positions of the segments. Instead of launching these operations in 2 separate runs, the AC_CS state machine is called immediately once the end of a segment is determined. Fig. 5 shows the pseudo code and Fig. 4 gives an illustration, whose explanation is as follows.

Generally speaking, the procedure tries to extend a suspicious segment incrementally. Whenever a possible CS match needs verifications, the coverage of this possible match will be checked to decide whether it overlaps with the current suspicious segment being extended. If so, the segment will be elongated by updating the ending position (e.g., when pos=8~12 in Fig.4); otherwise it means that the segment can not be extended more and should be sent for a verification by the AC_CS state machine; meanwhile, a new suspicious segment is generated to include this possible CS match (e.g. when pos=19 in Fig.4). Note that the AC_CS state machine also needs to be called for the last segment at the end of the whole pattern matching process. In this way, it is guaranteed that no redundant character comparisons are incurred, and no characters other than those inside the possible matches are verified.

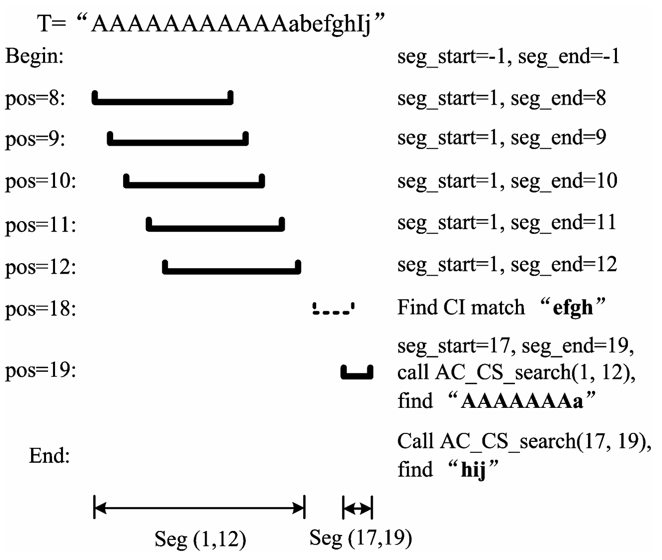


Figure 4. An illustration of the incremental verfication.

```
Initialize global variables before searching each T:
        seg_start = -1;
        seg_end = -1;

Function Verify(v_start, v_end):
        // Called in the Search phase when a possible
CS match need to be verified, with start and end position
specified by v_start and v_end
        If (v_start > seg_end) and (seg_end != -1) then
                // This possible CS match has no
overlapping with the current segment
                AC_CS_search(seg_start, seg_end);
                seg_start := v_start;
                seg_end := v_end;
        Else    // extend the current segment
                seg_end := v_end;

After searching each T:
        If seg_end != -1 then
                AC_CS_search(seg_start, seg_end);
```

Figure 5. Pseudo code of the incremental verfication.

The CSiV approach is designed to compete with the CSV one in ordinary cases, which outperforms the DS approach, and achieve a high accelerating factor against CSV in the worst-case scenarios.

On one hand, since the frequency of CS match occurrences is low in real-life traffic, the verification phase usually takes just a little proportion in the total processing time. Therefore in this case, the processing time of the CSiV approach is dominated by the one-time scan launched in the search phase, which is distinctly shorter than that needed by the DS approach to scan the payload twice.

On the other hand, for the worst-case scenarios, when verifications are required for each and every incoming byte, the whole payload will be regarded as a single large segment by CSiV to be verified. For the verification process, the AC_CS state machine requires performing $2M$ character comparisons. This number is apparently much lower than $(M - L_{Patt} + 1)\bullet L_{Patt}$, which is needed by the CSV approach, considering the fact that there are some patterns with $L_{Patt} \geq 16$ [i] in the current snort rule set.

III. Experimental Results

A. Experiment Setup

We evaluated the performance of CSiV against DS and CSV approaches using a set of real-life packet traces collected during the DARPA NIDS evaluation tests at MIT Lincoln Laboratory in year 2000 [5]. The pattern matching algorithm used for the search phases of the three approaches is Aho-Corasick [4], which is employed by Snort v2.6 as the default algorithm due to its high throughput and deterministic performance.

All the experiments were conducted on a machine equipped with a 2.40GHz Pentium 4 processor with 8KB L1 cache, 512KB L2 cache, and 1GB DDR main memory. The host operating system was Linux (Red Hat Fedora Core 4, kernel version 2.6.5). All codes are developed based on the Aho-Corasick CSV implementation in Snort v2.6 and compiled using GCC v3.3.3.

The rule set used is from the Snort official website [2] dated Sep.1, 2006, which is converted by Snort to a pattern matching database containing 16704 patterns organized in 196 port groups. Note that a rule "**alert tcp any any -> any any (ack:0; flags:SFU12; content: "AAAAAAAAAAAAAAAA"; depth:16;)**" exists in the rule set. As analyzed in Section II.B, with such a pattern "**AAAAAAAAAAAAAAAA**" specified, the worst-case scenario for CSV and CSiV could be generated when every byte within all incoming packet payloads is replaced with character 'A'.

The packet trace set used includes three traces named LL_DOS_1.0, LL_DOS_2.0.2 and NT, respectively by [5]. To generate both the random and the worst-case scenarios, we keep the packet headers in these traces and replace the payload with random bytes and all 'A' characters, respectively. Therefore, each trace derived 3 different versions marked as "Normal Payload", "Randomized Payload" and "All 'A' worst-case Payload" as depicted in Table I.

The values of processing time presented in Table I are measured by repeatedly running 5 times of the target algorithms, so as to smooth the incurred noises resulted from indeterminate factors such as OS scheduling and IO activities. The values of memory cost are measured by keeping track of the dynamic memory allocating and freeing operations.

[i] For example, Snort 2.x contains a pattern composed by 16 consecutive 'A's.

TABLE I. COMPARISONS OF THE 3 APPROACHES

Approach	Memory cost (MBytes)	Processing Time (seconds)								
		LL_DOS_1.0			LL_DOS_2.0.2			NT		
		Normal Payload	Random Payload	All-'A' worst-case Payload	Normal Payload	Random Payload	All-'A' worst-case Payload	Normal Payload	Random Payload	All-'A' worst-case Payload
CSV	165.211	3.639	3.133	61.041	1.604	1.329	23.912	8.379	7.759	160.095
DS	165.163	6.647	5.749	12.435	3.050	2.363	5.038	16.966	13.513	31.848
CSiV	279.372	3.171	2.473	11.977	1.577	1.101	4.771	8.389	5.916	32.556

B. Collected Data

Table I shows the memory costs and processing times. It can be seen that with the three different trace sets, CSV invariantly encounters severe performance degradation (about 95%) in the worst case, i.e. the "All-'A' worst-case Payload"; in contrast, DS and CSiV consumed only about 20% of time of CSV for the All-'A' worst-case Payload. Please notice that the reduced values of worst-case processing time of DS and CSiV are still 4~5 times higher than the processing time with the "Randomized Payload", which is due to the greatly increased number of "match()" function calls to report the final matches.

Compared with CSV and CSiV, although DS performs well with the All-'A' worst-case Payload, it almost halves the searching performance when processing either the "Randomized Payload" or the "Normal Payload", which is coherent with the previous analysis.

On the other hand, CSiV requires more memory than CSV and DS, due to the extra storage for the AC_CS state machine. However, since only part of the patterns, i.e. the CS ones need to be replicated to AC_CS state machine, apparently the storage overhead is less than the size of the original pattern set. For instance the overhead is about 114MB, i.e. 69% of the original set in our case. Considering the fact that the throughput issue is much more critical than the memory consumption issue for NIDSes, it is desirable to trade a tolerable memory cost (114MB) off for ~400% worst-case performance gain over CSV and ~100% average-case gain over DS.

IV. RELATED WORKS

Pattern matching has been well studied in the literature for the past three decades, consisting of both software algorithms [6-11] and hardware mechanisms [12-14].

As far as hardware solutions are concerned, Mixed CS and CI patterns are easily supported since the Double Search Approach can be applied in a straightforward fashion by copying and driving the inputs to separate CS and CI pattern matching circuits in parallel. The parallelism enables the Double Search approach to avoid launching two scans in sequence and thus achieves high performance in the average case. [13] proposed a Ternary Content Addressable Memory(TCAM)-based string matching engine which naturally supports mixed CS and CI patterns due to TCAM's ability to set a bit mask for each memory bit inside. Based on [14], [15] also proposed an extension to support mixed CS and CI patterns which need not split the pattern matching engines.

Although hardware solutions are usually with much higher performance, they suffer from long developing period/time-to-market and high manufacturing cost, as well as low flexibility. Also be aware of the rapid development of the multi-core processors which is becoming powerful and comparable with traditional hardware solutions, developing fast and robust software algorithms supporting mixed CS and CI patterns is also necessary and worthwhile. However, this issue was not well resolved in the past. [10] revealed the fact that Snort meets worst-case bottleneck when encountering all 'A' payload, but it did not propose a solution to narrow the gap between the average-case performance and the worst-case performance.

V. CONCLUSIONS AND FUTURE WORK

Pattern matching with the support of mixed case-sensitive and case-insensitive patterns is a key method for NIDSes to detect and prevent malicious attacks targeting different applications and operating systems. However, in conventional software-based NIDSes, the implementation of this feature is difficult to achieve high performance in both average-case and worst-case scenarios, failing to meet the demand of a robust and high throughput NIDS. In the widely-used CSV approach, a large number of redundant character comparisons are launched and caused severe performance loss when facing worst-case oriented attacks. To eliminate the redundant operations, CSiV employs a dedicated Aho-Corasick state machine for incrementally verifying the possible case-sensitive matches, achieving a much better lower-bounded worst-case performance which is independent of both the traffic trace and the pattern length. The infeasibility of DS approach is also explained by analyzing its low average-case throughput. Extensive experiments based on real pattern sets along with both collected and artificial traffic traces show that, the performance of the proposed approach outperforms the DS approach by a factor of 2 in the ordinary cases, and is better than the CSV approach up to 5 times under the worst-case scenario, indicating both its feasibility and robustness for a worst-case safe NIDS.

In terms of future work, though pattern matching algorithm generally determines the performance of an NIDS, there are many other modules such as flow/session reassembling and alert reporting which may become bottlenecks when attackers intentionally generate malicious traffic. Therefore, further research on system-wide improvement of robustness of NIDSes is a promising direction. Redundancy identification and mitigation, as we presents in this paper, are expected to be helpful for solving such problems.

REFERENCES

[1] "2005 FBI Computer Crime Survey," http://www.digitalriver.com/v2.0-img/operations/naievigi/site/media/pdf/FBIccs2005.pdf.

[2] Snort - the de facto standard for intrusion detection/prevention, http://www.snort.org.

[3] "Clam AntiVirus," http://www.clamav.net.

[4] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *communications of the ACM*, vol. 18, pp. 333-340, 1975.

[5] "MIT DARPA Intrusion Detection Data Sets," http://www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html.

[6] G. A. Stephen, "String Searching Algorithms," *Lecture Notes Series on Computing*, vol. 3, 1994.

[7] M. Fisk and G. Varghese, "An analysis of fast string matching applied to content-based forwarding and intrusion detection," Technical Report CS2001-0670 (updated version), University of California - San Diego 2002.

[8] E. P. Markatos, S. Antonatos, M. Polychronakis, and K. G. Anagnostakis, "EXB: exclusion-based signature matching for intrusion detection," presented at IASTED International Conference on Communication and Computer Network (CCN'02), 2002.

[9] R. T. Liu, N. F. Huang, C. H. Chen, and C. N. Kao, "A fast string-match algorithm for network processor-based network intrusion detection system," *ACM Trans. embedded Computing Systems*, vol. 3, pp. 614-633, 2004.

[10] S. Antonatos, M. Polychronakis, P. Akritidis, K. G. Anagnostakis, and E. P. Markatos, "Piranha: Memory-efficient String Matching for Intrusion Detection," presented at the 20th IFIP International Information Security Conference (SEC 2005), 2005.

[11] K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis, "E^2XB: a domain-specific string matching algorithm for intrusion detection," presented at the 18th IFIP International Information Security Conference (SEC2003), 2003.

[12] S. Fide and S. Jenks, "A Survey of String Matching Approaches in Hardware," TR SPDS 06-01, University of California - Irvine, 2006.

[13] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM " in *Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04) - Volume 00* IEEE Computer Society, 2004 pp. 174-183

[14] J. v. Lunteren, "High-Performance Pattern-Matching Engine for Intrusion Detection," presented at IEEE INFOCOM 2006, Barcelona, Spain, 2006.

[15] "Efficient mathing of mixed case-sensitive and case-insensitive patterns," IP.com PriorArtDatabase, Disclosed by IBM, http://www.priorartdatabase.com/IPCOM/000138568/, 2006.