Scalable NIDS via Negative Pattern Matching and Exclusive Pattern Matching

Kai Zheng①, Xin Zhang②, Zhiping Cai③[i], Zhijun Wang④, and Baohua Yang①
①IBM China Research Lab, Beijing, P.R.China,
②Carnegie Mellon University, Pittsburgh, PA, USA
③School of Computer, National University of Defense Technology, ChangSha, P.R.China,
④Dep. of Computing, Hong Kong Polytechnic University, Hong Kong, P.R.China
zhengkai@cn.ibm.com

ABSTRACT

In this paper, we identify the unique challenges in deploying parallelism on TCAM-based pattern matching for *Network Intrusion Detection Systems* (NIDSes). We resolve two critical issues when designing scalable parallelism specifically for pattern matching modules: 1) how to enable *fine-grained* parallelism in pursuit of effective load balancing and desirable speedup simultaneously; and 2) how to reconcile the tension between parallel processing speedup and prohibitive TCAM power consumption. To this end, we first propose the novel concept of *Negative Pattern Matching* to partition flows, by which the number of TCAM lookups can be significantly reduced, and the resulting (fine-grained) flow segments can be inspected in parallel without incurring false negatives. Then we propose the notion of *Exclusive Pattern Matching* to divide the entire pattern set into multiple subsets which can later be matched against selectively and independently without affecting the correctness. We show that Exclusive Pattern Matching enables the adoption of smaller and faster TCAM blocks and improves both the pattern matching speed and scalability. Finally, our theoretical and experimental results validate that the above two concepts are inherently complementary, enabling our integrated scheme to provide performance gain in any scenario (with either clean or dirty traffic).

KEYWORDS

Negative Pattern, Exclusive Matching, Pattern Matching, Intrusion Detection

1. INTRODUCTION

Over the past decade, researchers have been advocating a clean-slate design of the Next-Generation Internet in order to intrinsically eradicate network security vulnerabilities. While endeavors from this architectural perspective make provision for the long term, in the current practice, *Pattern Matching* (PM) in *Network Intrusion Detection Systems* (NIDSes) remains one of the de facto resorts for mitigating immediate network security plagues. While not a panacea, NIDSes prevent, to a large extent, malicious intrusions that could potentially incur losses of hundreds of millions of dollars every year [1]. Nowadays, ever-increasing line rates require a *fast* and *scalable* PM module. However, PM is inherently resource-intensive with regards to both computation and communication (i.e., I/O), when given a considerable traffic volume and a sizable pattern set with length-varying patterns.

We observe that, *content pre-filtering* and *parallelism* are two promising means to dramatically increase PM throughput. First, content pre-filtering refers to the pre-processing of flows or pattern sets to reduce the overhead of the actual PM operations. It embodies the philosophy of trading additional *cheap* operations for fewer *expensive* resource expenditures. Second, exploiting parallelism has been widely proven to be an effective solution for achieving greater system efficiency/performance. Unfortunately, traditional pre-filtering schemes require sequential processing of the input flows and are difficult to be parallelized. In addition, parallelism must be deployed while maintaining PM scalability and the correctness of the PM results (though parallel processing tends to increase power consumption).

We propose to partition both individual flows and the pattern set to approach the above two goals. From a high level, first, an incoming flow is divided into consecutive segments, which can be inspected in parallel. Second, the entire pattern set is partitioned into multiple subsets, which can be looked up individually and selectively (as opposed to checking the entire pattern set every time), to provide extra dimensions for parallelism and to reduce the power consumption due to hardware accesses. Implementing parallelism via such a two-level partitioning, however, creates two fundamental challenges. First, partitioning the flows is non-trivial, since a sophisticated attacker can split a malicious pattern across multiple packets within a flow. Consequently, to capture such cross-packet dependencies (thus avoiding false negatives), flows are usually re-assembled and the packets in each flow are inspected sequentially [11]. Second, the partition of the pattern set should not incur false negatives, either. And it is challenging to minimize the power consumed by hardware accesses and to effectively balance the workload among the pattern subsets.

In light of the above observations, in this paper we develop an efficient parallel PM module based on TCAM coprocessors, which significantly reduces the number of TCAM accesses. We first introduce the novel concept of *Negative Patterns* to address the challenge of partitioning flows. In a nutshell, a negative pattern a string chosen such that any part of it (i.e. any substring) will not match any other pattern in the given pattern set. Hence, by partitioning a flow only within negative patterns appearing in that flow, we incur no false negatives. Second, we propose the idea of *Exclusive Subsets* to efficiently partition the pattern set as follows. We identify all pairs of two patterns that will not be matched simultaneously

[i] The 3rd author is supported by the NSFC(NO.60603062 and No. 60903040) and HUNAN NSFC(No. 06JJ3035)

within one TCAM lookup input, and separate the two *exclusive patterns* in such a pair into *exclusive subsets*. Intuitively, each exclusive subset can later be looked up independently, since one TCAM lookup input can find matches in at most one of the exclusive subsets. Finally we will show that the techniques of negative patterns and exclusive subsets are inherently complementary, enabling our integrated scheme to outperform previous approaches in any scenario (with either clean or dirty traffic).

2. RELATED WORKS

Some recently developed PM schemes [6-9] use heuristics to filter the strings that cannot be matched by any suspicious patterns. These schemes either require a large amount of memory for software implementations, or they fail to work with large pattern sets. For example, in [2, 6], space-efficient prefix Bloom Filters are proposed to detect multiple intrusion patterns with fixed lengths. However, due to the variable length of the patterns, a large number of Bloom Filters with different lengths must be constructed, making them un-scalable.

FPGAs can provide high speed PM due to their exploitation of reconfigurable hardware capability and their ability in developing parallelism. Some of these approaches [4,6,10] are claimed to be able to support 10Gbps wire-speed by exploiting parallelism. But the intrusion patterns in the hardware need to be re-compiled and "downloaded" to the hardware when the patterns are updated/added/deleted, and hence do not support incremental updates. They also have huge implementation overheads in terms of both cost and development time.

Ternary Content Addressable Memories (TCAMs) are fully associative memory. TCAM has been widely used for packet classification and PM in high-speed routers and NIDS. In [5], a gigabit rate intrusion detection system using a single-TCAM coprocessor is proposed. The scheme can handle both *simple* and *composite*[ii] [5] pattern matching. In this scheme, a long pattern is divided into several sub-patterns (simple patterns) to fit within the TCAM's slot size, and a composite pattern is treated as a combination of multiple simple patterns. For a pattern with length less than the TCAM slot size, some wildcarded bytes are attached to fit the size. Then, with a *partial hit list* mechanism to verify the long or composite patterns, all the patterns can be handled using a unified scheme. It is claimed that the system can handle up to OC-48 (i.e. 2.5Gbps) line rates by using state-of-the-art TCAM chips. However, due to the need to inspect a data stream by shifting one byte at a time, the system requires a high throughput TCAM lookup, which is limited by the "*Frequency Wall*" of VLSI and also results in relatively huge power consumption.

3. DESIGN CONCEPTS

In this paper, we do not intend to take part in the debate on "*Which category of PM schemes is the best?*", which has been an open topic in the literature for years. Each category has its own strengths and weaknesses, better or worse depends heavily on the requirements of the specific NIDS under development.

In this paper, we base our work on [5] which uses TCAMs as a key building block. Our focus is on how to exploit parallelism while also improving power efficiency. We will not address how to handle long patterns and composite patterns, since these have been handled very well in [5] (and as will be mentioned in Section 4, in the proposed NIDS prototype, a dedicated mechanism will be used to handle general regular expression matching). Before delving into implementation details, in this section we first sketch the high-level ideas of our key approaches.

3.1 Content Pre-Filtering vs. Flow Segmentation

One major challenge of traditional PM and/or content pre-filtering mechanisms comes from the fact that usually a specific flow stream has to be inspected sequentially (e.g., one byte examined at a time), in order not to miss consecutive sub-strings/patterns. Therefore, individual packets need to be first buffered and then put together in the original order to reconstruct the flow. To boost PM throughput, it is intuitive to deploy parallelism at the flow level, such that the packets within one flow can still be inspected sequentially. This straightforward measure nevertheless turns out to be ineffective, because the network packet flows[iii] tend to vary significantly in size and duration, and large flows with up to several million bytes are not uncommon [12], depriving the flow-level parallelism of effective load balancing.

Flow segmentation, which partitions an individual flow into successive fragments, enables parallelism at a finer granularity and facilitates better load-balancing for PM. The key challenge lies in that, when segmenting the flows, we must guarantee that any suspicious pattern does not span more than one segment, so that each segment can be inspected individually (Section 3.2).

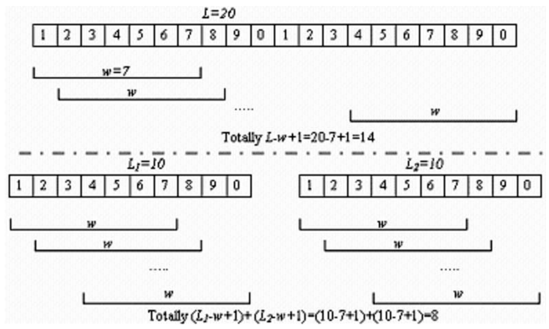

Figure 1. Number of lookups reduces when the flow is partitioned.

Meanwhile, we also observe that, besides the load balancing advantage, flow segmentation can also provide a PM speedup, especially for TCAM-based schemes.

[ii] As defined in [5], a *simple pattern* refers to a continuous bit string while a composite pattern refers to a composition of multiple simple patterns with some wildcards and/or negations inside.

[iii] Note that a malicious sender can intentionally separate the attacking "fingerprints" across multiple packets to avoid being detected at each individual packet. Therefore, to combat such surreptitious attackers, sessions/flows must be reconstructed. For example, *SNORT* [11] uses several preprocessor plug-ins to defrag IP fragments and to rebuild the TCP flows/sessions before deploying PM against the network data.

Consider the example shown in Fig.1. Given that the TCAM slot size is *w* bytes (*w*=7 in the example), an input flow with 20 bytes requires 14 TCAM lookups. However, if we partition the flow into two 10-byte segments, then the inspections can be performed on the two segments in parallel, and the total number of TCAM lookups is reduced to only 4+4=8. Generally speaking, given a flow Text *T* with length *L*, and the TCAM slot size *w*, the number of TCAM lookups required to inspect *T* is *L*-*w*+1. If we partition *T* into several small consecutive segments $\{t_1, t_2, ..., t_n\}$($\sum_{i=1,...,n} l_i = L$, where l_i is the length of the *i*th segment), then the number of TCAM lookups is reduced to,

$$\sum_{i=1,...,n} Max(l_i - w + 1, 1) \qquad (3\text{-}1)$$

We can express the minimal number of TCAM lookups as,

$$\begin{aligned} &Min \sum_{i=1,...,n} Max(l_i - w + 1, 1) \\ &= n + Min \sum_{i=1,...,n} Max(l_i - w, 0) = n = \lceil L / w \rceil, iff \forall i, l_i = w \end{aligned} \qquad (3\text{-}2)$$

(3-2) indicates that if we could partition the flow content into segments with sizes as close to the TCAM slot size as possible, then the number of TCAM lookups would be minimized. In addition, note that the result of

$$L - w + 1 - \sum_{i=1,...,n} Max(l_i - w + 1, 1)$$

reflects the number of TCAM lookups that can be saved by using segmentation, compared to that without flow segmentation. The following calculation provides us with a lucid estimation of such a reduction. Considering the case where we guarantee that $\forall i, l_i \geq w$ (i.e. not to partition the flow into segments shorter than *w* bytes), we further have,

$$\begin{aligned} n_{\text{save}} &= L - w + 1 - \sum_{i=1,...,n} Max(l_i - w + 1, 1) \\ &= L - w + 1 + nw - n - \sum_{i=1,...,n} l_i = (n-1)(w-1) \end{aligned} \qquad (3\text{-}3)$$

(3-3) implies that the more segments we can partition the flow into, the fewer TCAM inspections will be needed (which is independent of the total flow length *L)*; the larger the window size *w* is, the more significant the benefit can be. In this sense, flow segmentation can also be treated as a form of content pre-filtering for the traditional PM, since it tends to trade simple operations for expensive ones; the more positions we could find to partition the flow (without false negatives), the more TCAM inspections we can save, as long as the segment sizes are no smaller than *w*. We further identify the challenge for such inner-flow segmentation as two-fold: firstly, how can we guarantee that no pattern will span across multiple segments? Secondly, how can we make the segmentation process efficient? The subsequent subsection sketches the *negative pattern* technique as the solution.

3.2Negative Pattern Matching (NPM)

Traditional content pre-filtering methods try to find the occurrence of certain sub-strings from a given pattern set in an input stream, which is called "positive pattern filtering". Here we propose the notion of *Negative Pattern* (NP) matching/filtering. Instead of finding the positive occurrence of the sub-strings in a given pattern set, we aim to unearth "negative patterns" that do not appear in the given set and that allow the flow content to be partitioned without false negatives. As aforementioned, flow segmentation based on negative pattern matching serves as a form of content pre-filtering. Let us first formally define a negative pattern as follows.

Definition 1**: *Negative Pattern*: Given a pattern set *S*={*P*$_1$, *P*$_2$,…, *P*$_n$}, pattern *np* is a Negative Pattern of *S* *iff* any *k*-byte (*k*>1) suffix of *np* is not a sub-string of any $P_i \in S$, $i = 1,2,...,n$.**

For illustration, given that the pattern set *S* includes two patterns {*pattern, attack*}; the "positive" sub-string set of *S* is, *SS*={*pa, at, tt, te, er, rn, ta, ac, ck, pat, att, tte, ter, ern, tta, tac, ack, patt, atte, tter, tern, atta, ttac, tack, patte, atter, ttern, attac, ttack, patter, attern, attack, pattern*}. According to Definition 1, we say pattern "*negative*" is a negative pattern of *S,* since none of its *k*-byte (*k*>1) suffixes, i.e. any string in {*ve, ive, tive, ative, gative, egative, negative*}, is included in *SS*.

Theorem 1**: If we find a Negative Pattern *np* in the flow to be inspected, then it guarantees that no pattern(s) in *S* appear(s) across the adjacent segments after the segmentation, if only we cut the flow between the last two bytes of *np*.**

Proof: We prove Theorem 1 by contradiction. Suppose that there exists a pattern *p*∈ *S* across the partitioning point. Hence *p* has at least 2 bytes, i.e. the last 2 bytes of *np*, and therefore *p* must contain a 2-*byte* suffix of *np*. However, according to the definition, any 2-byte suffix of *np* must not be a sub-string of *p*. Therefore such a pattern *p*∈ *S* must not exist □

For instance, let us continue to use the pattern set *S*={*pattern, attack*}. Given that the byte stream to inspect is *T*="*…thisisapatternindicatingattacks…*"; according to the definition of a negative pattern, we know that {*isi, sap,…erni,…inga,…*} are NPs of *S*, since any *k-byte* ($k > 1$) suffix of them is not included in *SS*. Therefore, according to Theorem 1, we say that, {"*…this*", "*isa*", "*pattern*", "*indicating*", "*attacks…*"} is a valid partition of *T*, which will not lead to false negatives if the partitions are inspected in parallel.

Definition 1 and Theorem 1 answer the first question in Section 3.1: how to partition the flow while guaranteeing that no potential pattern in the flow will be missed if each segment is to be examined separately. By Theorem 1, whenever an NP is found, the flow can be partitioned at a position within the NP (i.e., between the last 2 bytes).

So far, there are still problems remaining unsolved, i.e. how to build up an NP set and how to make the segmentation process efficient enough to achieve an overall performance gain. We will delve into the details of our NP matching implementation in Section 4; below, we first present the key ideas of pattern set partitioning with exclusive pattern matching.

3.3 Using Multi-block TCAM

The full associativity of all TCAM entries gets in the way of making TCAM/Tagged-memory faster and scales better (with respect to both power consumption and timing criteria). Note that traditionally, the TCAM is actually a single block fully associated tagged memory. Reducing the capacity of the TCAM blocks will make it possible to deliver significantly higher working frequency, although this might reduce the number of entries that can be compared at a time. It is intuitive for us to think about using a TCAM with a set of small blocks rather than one with a single huge fully associative block. So that we can try to partition the pattern set into several subsets and distribute them among the blocks on the TCAM to increase the system frequency. With the aid of mature pipeline techniques (where the scan in the subsets are treated as the stages in the pipeline), system throughput can be improved accordingly. In this paper, our concepts and algorithms are instantiated on multi-block TCAMs which operate at a higher frequency; however, they are not limited to using the pipelining processing model. Furthermore, we argue that the performance gain by using the multi-block TCAM can pay off the implementation complexity introduced.

3.4 Exclusive Subsets & Exclusive Matching (EM)

There might be numerous ways of partitioning the pattern set (e.g., so as to fit the subsets into the TCAM blocks, respectively). It might be interesting to ask "Are there partitioning schemes which make it possible to avoid going through all of the subsets for each specific lookup?" One might think of Hash-based schemes (i.e., partitioning the pattern set based on the hash of certain fixed bits of each pattern) at first glance, which sound intuitive and straightforward. However, unfortunately, because of the following considerations, the hash-based pattern grouping schemes are not practical for PM.

1) Wildcards appear frequently in arbitrary locations in the patterns (also note that the presentation of the short patterns in TCAM always include several wildcard bits, for example as was the case in [5]). Replication of the patterns might be required when the hash keys of the patterns include wildcards. For instance, given the pattern set {1*0*, *1*0, 1*00, 11*0} and assuming the first 2 bits are chosen as the hash key, then the 4 patterns are grouped into 2^2=4 subsets, i.e., $\{\Phi\}_{00}$, $\{*1*0\}_{01}$, $\{1*0*\}_{10}$, $\{1*0*, *1*0, 1*00, 11*0\}_{11}$. We can see that 2 out of the 4 patterns are replicated, resulting in an expansion ratio of 50%. What's more, the last subset includes all the patterns, which means the partition actually failed.

2) Load balancing tends to be a tough challenge for the Hash-based partitioning scheme to solve. We note that the occurrence of contents (i.e., characters) in the traffic is usually indefinitely and unevenly distributed. Different pattern groups may therefore have unbalanced and fluctuating load ratio, and some pattern subsets may become access bottlenecks.

Now, we will go through a very simple example to gain some intuition in our search for an alternative to hashing. Consider a pattern set with two patterns {ax*c, x*yz}, and assume the width of the TCAM (i.e. the scan window) is *w*=7 Bytes. We can say that the two patterns are matched exclusively for any traffic content/text in any cases, no matter how the patterns are presented in the TCAM, for example "ax*c***" , "*ac*c**", … or "***ac*c"... The observation is that none of the prefixes of the patterns is also a suffix of the other, and they are not sub-strings of each other either. Therefore, any content that matches both patterns at the same time must be at least $|ax*c|+|x*yz|$=8 bytes, which is longer than the scan window size *w*=7.

Definition 2**: *Minimum Combine Length (MinCLen)*: Let $\Omega(P)$ denote the sub-string set of an arbitrary string *P*. The "*MinCLen* of Pattern *A* and Pattern *B*", *i.e. MinCLen(A,B),* is defined as the minimum length of any string *P*, where** $P\in\{P_s \mid A,B\in\Omega(P_s)\}$ **.**

For example, *MinCLen*("123ab", "ab123")=7, since pattern "ab123ab" is (one of) the shortest pattern which has both "ab123" and "123ab" as sub-strings.

Definition 3**:** $A\nabla_w B$ **: Given pattern *A* and *B*, and the scan window size *w*, symbol** $A\nabla_w B$ **denotes the case when A and B must be matched exclusively, i.e. there does not exist such a string *P* satisfying both** $A,B\in\Omega(P)$ **and** $|P|\le w$ **.**

Theorem 2**:** $A\nabla_w B$ ***when MinCLen(A, B)>w.***

Proof: Assume that there exists a string *S* (|*S*|=*w*) which matches both Pattern *A* and *B*, and *MinCLen(A, B)>w.* It is obvious that *A* and *B* must be sub-strings of *S*, i.e. *S* satisfies $A,B\in\Omega(S)$. However, according to the definition of "*Minimum Combine Length*", any string *S* satisfies $A,B\in\Omega(S)$ should be longer than *w*, which conflicts with the assumption. Therefore such a string *S* does not exist, in another word, A and B are impossible to be matched at the same time □

Theorem 2 explains why "*ax*c*" and "*x*yz*" are matched exclusively given the scan window size *w*=7, since *MinCLen*("ax*c", "x*yz")=8>*w*=7.

Definition 4**:** $S1\Delta_w S2$ **: Given the scan window size *w*, we say Pattern subset *S1* and *S2* are matched exclusively with each other, (i.e.** $S1\Delta_w S2$ **) *if and only if*** $\forall A\in S1,\forall B\in S2, A\nabla_w B$ **.**

Definition 4 poses a formal criterion on when to leverage the "Exclusive" property to speedup the lookup, regardless of the traffic content within the *w*-byte scan window. Given the "exclusive" pattern subsets, $S_1,S_2,...,S_n,\forall 1\le i,j\le n, S_i\Delta_w S_j$, since for any *w*-byte string (or those shorter than *w*-byte), match can be found in no more than 1 subset, all the subsets need not to be walked through all the time. The lookup process terminates whenever match is found; only when no match is found is it necessary to go through all the subsets.

Theorem 3**: If** $S1\Delta_w S3$ **and** $S2\Delta_w S3$ **, then** $(S1\cup S2)\Delta_w S3$ **.**

Proof: Obvious, according to Definition 4.□

p	*a*	*t*	*t*	*e*	*r*	*n*
a	*t*	*t*	*a*	*c*	*k*	

Original Pattern Set

		NP?
…	…	
t	*a*	0
t	*b*	1
…	…	
p	*a*	0
…	…	
x	*y*	1
…	…	

2-Byte NP Table

Figure 3. The data structure used by the proposed NP pre-filter.

In Section 4, we will discuss in details how to partition the pattern set into exclusive subsets, make use of the idea and make it actually practical.

3.5 The Complementarity between NP Matching & Exclusive Matching

A very interesting fact of "Exclusive Matching" is that the "dirtier" the traffic content is, the greater the expected speedup, since dirty traffic leads to higher chance of getting a "hit". The earlier a hit is found, the more TCAM accesses can be saved. This advantage of "Exclusive Matching" provides a very good countermeasure for the worst-case scenario in which the attackers generate dirty traffic/attacks to exhaust the NIPS/NIDS. On the other hand, in ordinary usage, when the traffic is typically very clean, attack patterns are seldom found. In this case, NP matching, i.e., the idea introduced in Sections 3.1 and 3.2, in turn results in performance gains. More theoretical analysis on a system leveraging the complementarity between NP matching and exclusive matching will be discussed in Section 5.

4. IMPLEMENTATION AND PROTOTYPE

4.1 Overall Architecture & Workflow

As depicted in Fig. 2, the proposed PM system consists of an *NP Analyzer (NPA)*, an *Exclusive Set Partitioner (ESP)*, a hardcore accelerator, a multi-block TCAM module, and a general-purpose multicore processor.

The workflow comprises an *off-line preprocessing* stage and the subsequent *real-time PM* stage. In the off-line preprocessing stage, i) NPA scans the pattern set to obtain the NPs and put them (there might be multiple instances for parallelism) into the accelerator ; ii) in addition, ESP inspects the pattern set, partitions it into exclusive subsets and stores them independently in the TCAM blocks.

In the real-time PM stage, the incoming flows will be dispatched into the hardcore accelerator for content pre-filtering (i.e. flow segmentation) and be inspected for simple patterns in parallel on the exclusive subsets populated in the TCAM blocks. Then the matching results (e.g. which simple pattern has been matched at which position in the flow, etc.) will be delivered to the general-purpose processors for further post-processing such as a general regular expression matching etc. Recall that in this paper we focus on simple PM in the hardcore accelerator, which lies in the critical data path. In what follows we delineate each key component in the hardcore accelerator, as well as the overall workflow.

4.2 The Preprocessing Stage

4.2.1 Construction of the NP Table

Fig.3 depicts the key data structure, i.e. the NP table, used by the proposed content pre-filter. Given the pattern set (in this example the set contains two patterns {pattern, attack}), the corresponding NP table enumerates all the strings that are NPs, according to Theorem 1. To avoid too much overhead of table lookup, we use 2-byte NP size in our prototype, which is proven to yield good enough filtering effects according to our experimental results. For instance in Fig. 3, sub-string "*ta*" appears in pattern "at<u>ta</u>ck", so "*ta*" is not an NP. However, none suffix (which >2 bytes) of "*tb*" appears in any pattern in the pattern set, so "*tb*" is an NP, and so forth. The algorithm for enumerating all the 2-byte NPs is straightforward and we omit its pseudo-code in this paper. Since the NP table

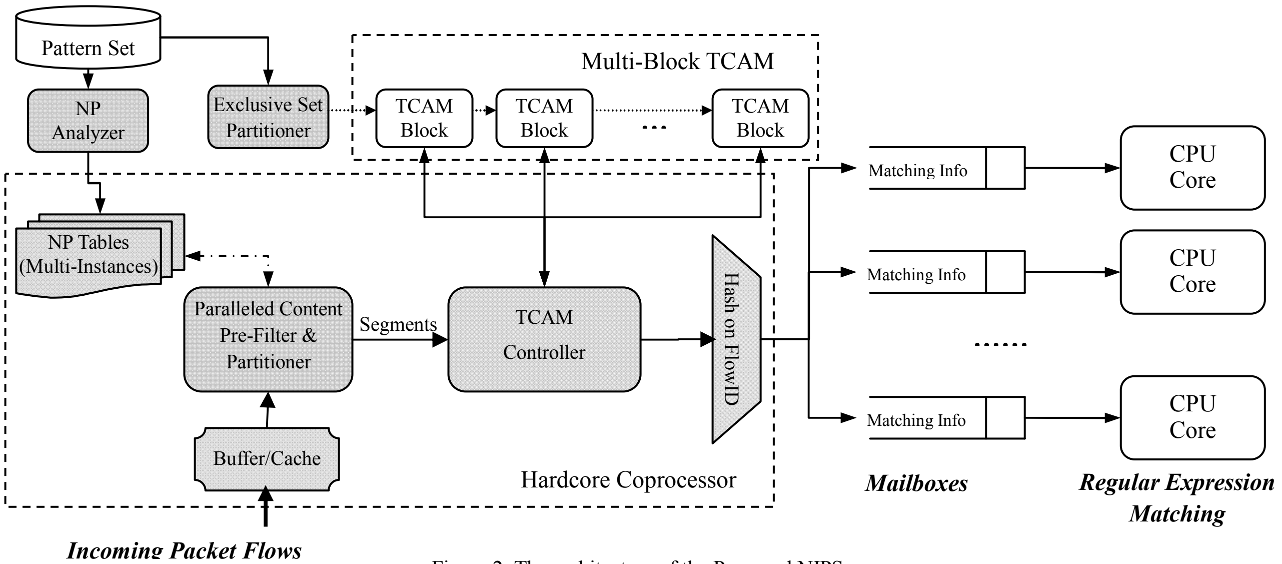

Figure 2. The architecture of the Proposed NIPS.

is tiny enough (e.g. 2^{2x8}=64K bits when using 2-byte NPs) and each 2-byte string in the flow can be inspected independently, fast ASIC and multiple NP tables can be employed for parallel (and fast) NP table lookups.

4.2.2 Partitioning the Pattern Set

In Section 3.4, we briefed the idea of partitioning the pattern set into exclusive subsets, so that inspection may be performed on selective subsets to improve system throughput and reduce power consumption. To practically implement this idea, we bear the following goals in mind when designing the partitioning algorithm: 1) the number of "exclusive" subsets should be large enough to produce significant parallel speedup and power reduction; 2) the sizes of the subsets should be balanced; and 3) the partitioning process should take reasonable time to finish. Though 1) and 2) inevitably depend on the characteristics of a given pattern set, a smart partitioning algorithm can still push the results toward the optimum on a given pattern set.

```
int ExclPart(PatternSet *PS, PatternSet *ExSubsets[], int n){
Stage I:
    Init_all_subsets(ExSubSets[]);
    for each pattern in PS do {
        cs = next_empty_subset_in(ExSubsets[]);
        cs->add_pattern(pattern);
        for each non-empty subset in ExSubSets[] except cs do {
            if IsExclusive(cs, subset)==FALSE {
                Merge(cs, subset);      //Merge subset into cs
                Clear(subset);
            }
        }
    }
Stage II:
    Sort the sets in ExSubsets in the decreasing order of the patterns number
    for each non-empty subset in ExSubsets[] do {
        if subset is the n largest ones
            continue;
        else
            merge subset into the smallest one of the n largest subsets.
    }
}
```

Figure 4. Pseudo-code for partitioning the pattern set.

We propose a heuristic partitioning algorithm running within polynomial time. The algorithm contains two high-level stages. Figure 4 shows the pseudo code of the proposed partitioning algorithm. In the first stage, the patterns are allocated one by one to either an existing subset or a newly created one based on the following principles.

A. If the current pattern is exclusive with all previously allocated patterns, then a new subset will be created for the pattern;
B. If the current pattern is not exclusive with some of the previously allocated patterns which are in the same subset, then the current pattern will be allocated to the subset;
C. If the current pattern is not exclusive with some of the previously allocated patterns which belong to different subsets, then all the related subsets should be merged and the current pattern will be allocated to the merged subset.

In the second stage, the subsets with unbalanced sizes will be reformed into new ones with balanced sizes (correctness is guaranteed by Theorem 3). A classic greedy algorithm for the "knapsack problem" is used, where the subsets resulting from Stage 1 are regarded as the items and the target subsets are treated as the knapsacks.

4.3 Real-Time Pattern Matching

4.3.1 Flow Segmentation Algorithm

According to the analysis in Section 3.1, the sizes of the segments should be as close to the TCAM window size (say, *w* bytes) as possible. In other words, a flow does not need to be partitioned at every NP in the flow; hence a brute-force approach to find out all NPs in a flow can incur unnecessary overhead.

```
NPSeg(byte *Content, int nSegLen, int w, int nRound, int N){
        if (nRound > w-1 or nSegLen < 2* w)
            TCAM_Lookup (Content,niSegLen);

        for m from 1 to M parallel_do {
            if IsNP(Content[nRound+m*w-1,N])
                NP_MAP[m]=TRUE;
        }
        lastNP=0;
        for i from 1 to M do {
        if (NP_MAP[i]==TRUE) {
          NpSeg (Content+lastNP*w, (i-lastNP)*w, w, nRound+1, N);
           lastNP=i;
        }
}

Hierarchical_NP(byte *C, int Length, int w, int N){
        NpSeg (C, Length, w, 0, N);
}
```

Figure 5. Pseudo-code for the Hierarchical NP Matching scheme.

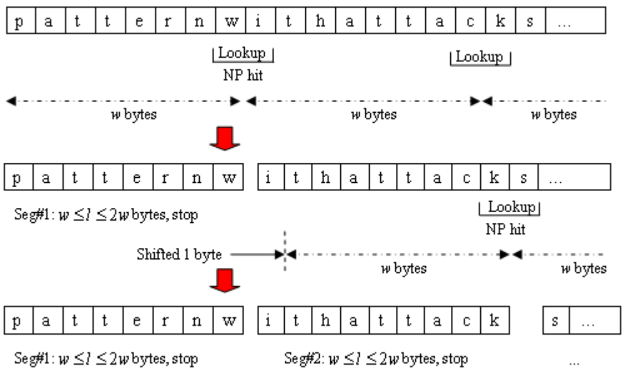
p a t t e r n w i t h a t t a c k s ...
Lookup
NP hit
Lookup
w bytes
w bytes
w bytes
Seg#1: w ≤ l ≤ 2w bytes, stop
Lookup
NP hit
Shifted 1 byte
w bytes
w bytes
Seg#1: w ≤ l ≤ 2w bytes, stop
Seg#2: w ≤ l ≤ 2w bytes, stop
...

Figure 6. A demonstration of the Recursive NP lookup scheme.

We propose in this paper an iterative flow segmentation algorithm (see Fig. 5). Let *M* be the number of parallel PM processing units (or the number of TCAM blocks) the system can provide. Then at each stage of the iterative algorithm, *M* locations (the length between any two adjacent locations is greater than *w*) in the flow are inspected in parallel; then based on the NP matching results of the *M* locations, the flow will be partitioned into a certain number of (i.e. 1 to at most *M*+1) segments, which will be handled independently in parallel at the subsequent stages. This operation will be performed iteratively at each stage until either no segment is longer than 2*w* bytes or certain preset constraints are reached. At each stage, the new locations to be examined are selected by shifting 1 byte from each of the previous locations, respectively. Fig. 6 gives an example of the operations of the above algorithm. In this example, we still use the pattern set *{"pattern", "attack"}* for illustration, and assume that the flow is "*patternwithattacks…*" and *w*=8. At the first stage, NP lookup is performed at the byte offsets (*w*:*w*+1), (2*w*:2*w*+1),…, (k*w*:k*w*+1) , simultaneously. NP hit is found at (*w*:*w*+1), and hence the flow is split to two segments as shown in Fig.6. At the next stage, no further NP lookup is needed for Seg#1, since its length is no longer than 2*w* bytes; for the remaining text, since it is still significantly longer than 2*w* bytes, further NP lookups will be performed at the positions (*w*+1:*w*+2), (2*w*+1:2*w*+2],….

4.3.2 Pattern Matching in the Exclusive Subsets

After the above pre-filtering process, the flow is partitioned into segments and ready for parallel inspections in the multi-block TCAM. Suppose that the pattern set has been divided into *K* exclusive subsets which are allocated to *K* blocks in the TCAM (one subset in each block). Generally speaking, for a flow segment with l ($l \geq w$) bytes, (l-w+1) input windows (each with *w* bytes) need to be inspected against 1 up to *K* exclusive subsets (i.e. TCAM blocks). Note that for a *w*-byte sub-string of the segment (see Fig.7), the lookup time in the multi-block TCAM is nondeterministic according to the "done-on-hit" principle introduced in Section 3. In the worst case (l-w+1)x*K* TCAM lookups are required for the segment. We name these lookups the *Potential Lookup Tasklets* (PLTs) of the segment.

A *TCAM controller* is further deployed to dispatch and schedule the PLTs. In each TCAM cycle, the TCAM controller tries to dispatch as many pending PLTs as possible to the TCAM blocks concurrently. To maximize both the TCAM utilization and throughput: 1) only the PLTs corresponding to different TCAM blocks can be dispatched concurrently; and 2) one sub-string should be dispatched into multiple TCAM block/subsets (if necessary) only in a *sequential* manner, so that an early match in one pattern subset can prevent the lookups in other subsets according to the exclusive PM property. We observe that the above formulation is essentially one instantiation of the chessboard problem which frequently arises in the traditional switch fabric context. In this paper, a simple but effective round-robin-based scheduling scheme is used.

As depicted in Fig. 8, at the beginning, *K w*-byte (*K*=4, *w*=7 in this example) text sub-strings are extracted from the segment pool by the scheduler and loaded to the sub-string latches within the TCAM Controller. During each TCAM cycle, the contents in the *K* latches will be matched against the *K* TCAM blocks, respectively. Whenever a match hit is observed, the matched sub-string will leave the system immediately as it needs no more TCAM lookups according to the exclusive pattern matching property. The scheduler will fetch another sub-string from the segment pool to replace the content in the corresponding latch. Meanwhile, the *K* round-robin switches will shift a number so that the unmatched substrings can be inspected in other pattern subsets in turn. If a substring remains unmatched when the corresponding switch shifts *K*-1 times, it will leave the system with a *matching miss*. Finally, recall that only the matched sub-strings will be further delivered to the general-purpose processors for regular expression matching.

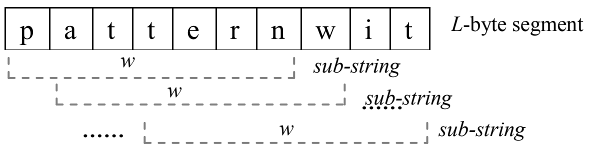

Figure 7. Term Definitions: Text segment and its *w*-byte substrings.

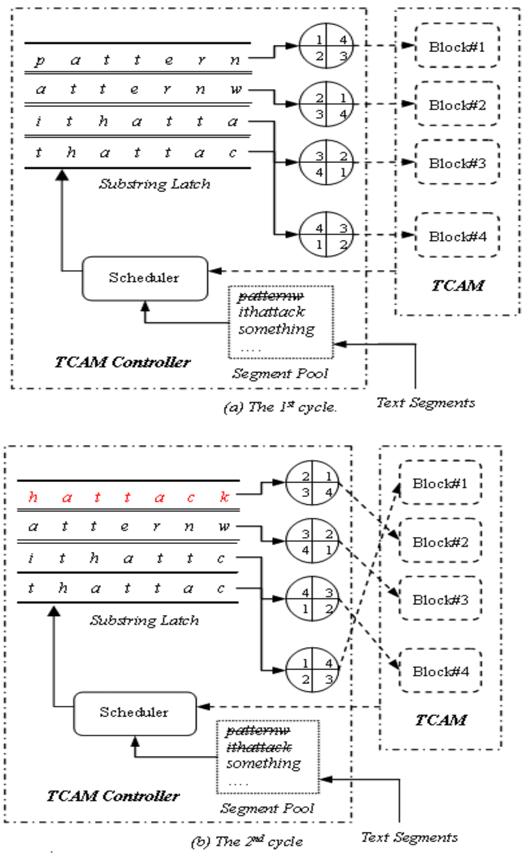

Figure 8. An example of the TCAM Controller. In this case, the content "*pattern*" in the 1st latch is reported as "*matched in TCAM Block#1*" in the 1st cycle. So in the next cycle, the latch will be reloaded, and the next (7-byte) sub-string to load should be "*hattack*" (in Segment "*ithattack*"). In the 2nd cycle, the *K* Round-Robin switches all shift a number.

5. PERFORMANCE EVALUATION

5.1 Theoretical Analysis

The primary goal of the theoretical analysis is to show that the proposed scheme can achieve supiorior results with the aid of both NPM and EM, *regardless of* how dirty the incoming traffic is. Hence, the key input to the following analysis is a parameter that characterizes the degree of the dirtiness of the incoming traffic. To make the paramter most amenable to our specific TCAM-based mechanism, we devise the *dirty traffic ratio* (R_d) as the indicator of traffic dirtiness. Formally, R_d is defined as the number of inspection windows with PM hits (matches) to the total number of inspection windows required. For example, for a *L*-byte flow, we need *L*-*w*+1 inspections; if we get *m* PM hits during the *L*-*w*+1 inspections, the dirty traffic ratio is given by *m/*(*L*-*w*+1).

5.1.1 Additional Overhead for NP Matching

In the ideal case of hierarchical NP lookup (Fig. 5), an *NP hit* is found on each of the M positions being inspected, then only $\lceil L/w \rceil$ NP matchings are required. For the general cases, let $R_h\,(0 \le R_h < 1)$ denote the hit rate of the NP matching, and let D_r (also defined as *nRound* in Fig. 5) denote the number of stages/iterations in the "Hierarchical NP lookup" algorithm. Then the expectation of NP lookup times is given by,

$$n_{NP} = \sum_{i=0}^{D_r-1} (1-R_h)^i \times L/w = \frac{[1-(1-R_h)^{D_r}]\times L}{w \times R_h} \qquad (5\text{-}1)$$

From (5-1), we can see that, the NP lookup overhead is disproportional to the window size *w*, and inceases when R_h decreases. In the worst case when $R_h \to 0$, we have,

$$\lim_{R_h \to 0} \frac{[1-(1-R_h)^{D_r}]\times L}{w \times R_h} = \frac{L \times D_r}{w} \qquad (5\text{-}2)$$

5.1.2 TCAM Lookup Savings

Note that, two factors collectively give rise to TCAM lookup overhead or power consumption: the number of input windows for inspection, and the number of TCAM entries triggered for each lookup. To capture the effects of both aspects, we invent a metric N_{pro} as the *product* of the number of input windows (denoted by N_{win}), and the *average* number of TCAM entries triggered for each window (denoted by N_{ent}). Suppose the original pattern set occupies $|S_P|$ TCAM entries (*w* bytes each) in total. Now we first evaluate N_{pro} for the strawman approach where the flow is not segmented using NP, nor is the pattern set partitioned. Given an *L*-byte flow, *L*-*w*+1 windows need to be inspected, and each window needs to match against all the $|S_P|$ entries: $N_{pro} = (L-w+1) \times |S_P|$

Now, with NP-based flow segmentation, we first reduce the number of lookups (in terms of number of inspection windows) as presented in Section 3.1:

$$N_{Win} = \sum_{i=1}^{n} (l_i - w + 1) = L - n(w-1) \qquad (5\text{-}3)$$

where *n* denotes the number of segments the flow content is partitioned into. According to (5-1), we have,

$$n = n_{NP} \times R_h = [1-(1-R_h)^{D_r}] \times L/w \qquad (5\text{-}4)$$

and according to both (5-3) and (5-4), we have,

$$N_{Win} = L \times \{1 - [1-(1-R_h)^{D_r}] \times (w-1)/w\} \qquad (5\text{-}5)$$

Then suppose with EM we partition the parttern set into *K* subsets, each taking up $|S_P|$ /*K* TCAM entries, and assume that each subset has equal probability to match/hit the input string. Given a dirty traffic ratio R_d, we further have,

$$N_{ent} = \frac{|S_P|}{K}[\sum_{i=1}^{K-1} i \cdot \frac{R_d}{K} + K \cdot (1 - \frac{K-1}{K} R_d) = |S_P| \frac{2K-(K-1)R_d}{2K} \qquad (5\text{-}6)$$

Therefore, the percentage of TCAM lookups that can be saved is estimated by,

$$\begin{aligned} N_{save\%} &= 1 - N_{Win} \times N_{ent} / N_{pro} \\ &= 1 - \{1 - [1-(1-R_h)^{D_r}]\frac{w-1}{w}\} \frac{2K-(K-1)R_d}{2K} \frac{L|S_P|}{(L-w+1)|S_P|} \end{aligned} \qquad (5\text{-}7)$$

From (5-7), we can see the favorible fact that $\partial N_{save\%} / \partial R_h \ge 0$ (5-8) and $\partial N_{save\%} / \partial R_d \ge 0$ (5-9); since it is quite straighforward that $dR_h / dR_d \le 0$ (common sense), (5-8) and (5-9) indicate that though the lookup reduction rate tends to decrease as the traffic become cleaner (i.e. $R_d \to 1$), however, since the NP hit ratio R_h will increase as the traffic becomes cleaner, we can still expect performance gain with the proposed scheme.

5.2 Experimental Analysis

5.2.1 Experiment Setup

Performance may differ significantly across rule sets and traffic with different characteristics. Only real-life data sets can shed light on the degree of such impacts. The pattern set from SNORT [11] (dated March 2006) is used in our experiments. There are 62,674 2-byte negative patterns in this pattern set. The long patterns are pre-partitioned into *w*-byte sub-patterns to fit in the TCAM, using the method proposed in [5]. For example in the case where *w*=16bytes, 3005 sub-patterns are generated from the SNORT pattern set, and then the sub-patterns are group into 4 exclusive subsets which contain 955, 710, 670 and 670 sub-patterns, respectively.

The traces of the 1st week (attack-free) and 5th week (with slight attacks) from [3] are used in the experiments. In order to evaluate our system performance under heavy dirty flow contents, we also generate synthetic dirty traffic trace[iv], and the dirty traffic ratio, i.e. R_d, of the synthetic traffic is about 51%.

5.2.2 Experimental results

Fig.9(a) depicts the ratio of TCAM lookups saved, indicating the overall gain by deploying the proposed approaches. From the figure, we can see that around 38%~92% TCAM lookups can be saved in different experiments and configurations. The dominating factor of the performance gain is the inspection window size, *w.* It differs slightly among the cases with traffic dirty ratio (R_d) ranging from 0% to 51%, benefiting from the complementarity between NPM and EM.

Fig.9(b) depicts the speedup ratio by using the proposed scheme. Speedup ratio is defined as $K \times N_{TCAM\_lookup\_before} / N_{TCAM\_lookup\_after}$, where $N_{TCAM\_lookup\_before}$ and $N_{TCAM\_lookup\_after}$ denote the number of inspection windows to lookup in the TCAM before and after using the proposed approach, respectively[v]; *K* denotes the number of TCAM blocks or the number of exclusive pattern subsets. Note that

[iv] Several attack patterns from SNORT pattern set are randomly chosen and inserted into each packet in clear trace from [3].

[v] It is assumed that the working frequency of the multi-blocked TCAM is the same as the commodity ones. And actually it can be higher and result in additional performance gain.

parallelism is introduced when the TCAM is partitioned into blocks and inspections are performed in parallel (K=4 in the deployed experiments). Speedup ratio indicates the performance gain in terms of PM throughput. We can see that performance is improved significantly.

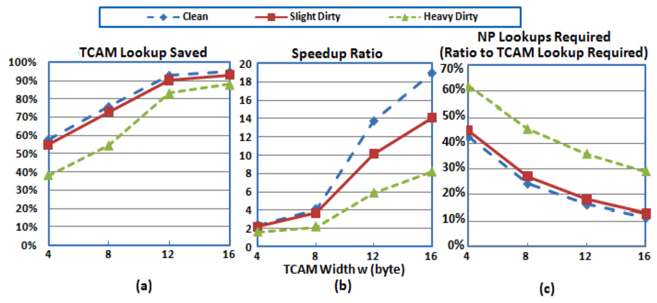

Figure 9. Experimental Results.

Further more, according to Fig.9(c), we found an obvious trend of the overheads (in terms of the ratio to TCAM lookups required) that it decreases significantly as the inspection window size (i.e. TCAM width) increases. This is also coherent with the estimation given by formula (5-2).Correlating the data of both TCAM lookup saved and overheads in terms of NP lookups required, taking the case of Trace#1 when w=16 for instance, we can see that the proposed scheme approximately trades 11% 2-byte NP table lookups for 92% 16-byte TCAM lookups, which obviously results in overall net performance gain. Note that the 2-byte NP lookups can be performed efficiently on the accelerator chip in parallel.

6. Conclusions

In this paper, two novel ideas are studied in order to exploit parallelism and speed up the PM operations using TCAM. First, starting with a different angle from the traditional content pre-filters, we proved that segmentation on the flow content to be inspected can significantly save the number of TCAM lookups. Second, we propose a distributed parallel PM scheme which makes use of the "Exclusive Matching" principle to speedup the PM process. We proved that any two patterns whose "*Minimum Combine Length*" is larger than the inspection window size will not be matched by the same TCAM lookup input. So that whenever match is found for one of the "exclusive" patterns given a TCAM lookup input, there is no need to further inspect other subsets, saving both memory access bandwidth and power. Finally, we demonstrate both theoretically and experimentally the complementarity between the two approaches. We argue that the combination of the content pre-filter based on NP Matching and the parallel PM mechanism based on Exclusive PM can cover each other for both clean and dirty traffic, making the proposed solution practical and attractive.

References

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

[2] N. S. Artan and H. J. Chao, "Design and analysis of a multipacket signature detection system", International Journal of Security and Network, v2(1), pp122-136, 2007.

[3] "MIT DARPA Intrusion Detection Data Sets," http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1999data.html

[4] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04), 2004.

[5] 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

[6] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," *IEEE Micro*, vol. 24, 2004.

[7] R. Ramaswamy, L. Kencl, and G. Iannaccone, "Approximate fingerprinting to accelerate pattern matching", ACM Internet Measurement Conference, 2006.

[8] C. J. Coit, S. Staniford, and J. McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *DARPA Information Survivability Conference* and Exposition (DISCEX II'01), 2001.

[9] C. Kruegel and F. Valeur, "Stateful Intrusion Detection for High-Speed Networks," IEEE Symposium on Research on Security and Privacy, 2002.

[10] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu, "A Memory-Efficient Parallel String Matching Architecture for High-Speed Intrusion Detection," *IEEE Journal on Selected Areas in Communications*, vol. 24, pp. pp. 1793-1804, 2006.

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

[12] N. Dukkipati and N. McKeown, "Why Flow-Completion Time is the Right Metric for Congestion Control," ACM SIGCOMM Computer Communication Review, vol. 36, pp. 59-62, 2006.