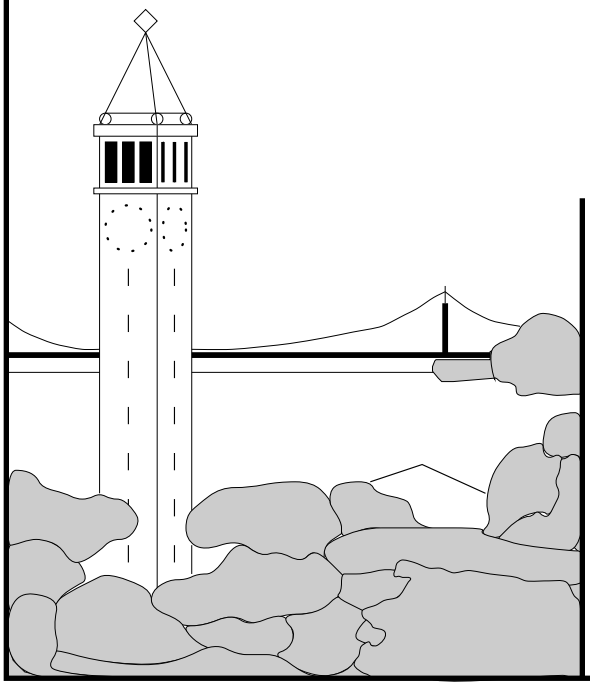
Gigabit Rate Packet Pattern-Matching Using TCAM

Fang Yu, Randy Katz, and T. V. Lakshman
{fyu, randy}@eecs.berkeley.edu lakshman@bell-labs.com



Report No. UCB/CSD-4-1341

July 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

This technical report is supported by UC Micro grant number 03-041 and 02-032 with matching support from NTT MCL, HP, Cisco, and Microsoft.

Gigabit Rate Packet Pattern-Matching Using TCAM

Fang Yu Randy H. Katz
EECS Department, UC Berkeley
{fyu, randy}@eecs.berkeley.edu

T. V. Lakshman
Bell Laboratories, Lucent Technologies
lakshman@bell-labs.com

Abstract

In today's Internet, worms and viruses cause service disruptions with enormous economic impact. Current attack prevention mechanisms rely on end-user cooperation to install new system patches or upgrade security software, yielding slow reaction time. However, malicious attacks spread much faster than users can respond, making effective attack prevention difficult. Network-based mechanisms, by avoiding end-user coordination, can respond rapidly to new attacks. Such mechanisms require the network to inspect the packet payload at line rates to detect and filter those packets containing worm signatures. These signature sets are large (e.g., thousands) and complex. Software-only implementations are unlikely to meet the performance goals. Therefore, making a network-based scheme practical requires efficient algorithms suitable for hardware implementations. This paper develops a Ternary Content Addressable Memory (TCAM) based multiple-pattern matching scheme. The scheme can handle complex patterns, such as arbitrarily long patterns, correlated patterns, and patterns with negation. For the ClamAv[1] virus database with 1768 patterns whose sizes vary from 6 bytes to 2189 bytes, the proposed scheme can operate at a 2 Gbps rate with a 240KB TCAM.

1. Introduction

In the current Internet, large number of malicious probes and worms spread every day. End-host based solutions rely on security service tools, traffic monitoring tools and anti-virus software. These approaches have drawbacks in being insufficiently fast to meet new virus threats, in needing coordinated actions by thousands of enterprises, and in incurring high costs due to duplicated prevention efforts. The inability to respond fast is increasingly being exploited by new worms that are designed to contaminate tens of thousands of hosts quickly (e.g., in less than an hour). It is hard to install security upgrades in large numbers of enterprise network clients within such a short time frame. A more effective approach is to use network based schemes that stop worm propagation in the network before they reach a significant number of end users.

Network Intrusion Detection Systems (NIDS) are well-suited for this purpose. They monitor packets in the network and scan packet payloads to detect malicious intrusions or Denial of Service (DOS) attacks. SNORT [2], a popular open source NIDS, has thousands of rules, each specifies an intrusion pattern to be used for packet payload scanning. However, most current network-based security devices can perform only layer 3 or 4 packet filtering with the packet header. Line speed filtering based on bit-patterns in packet payloads (content based or layer 7 filtering) is challenging especially when scanning for thousands of patterns.

Another difficulty in pattern matching is that virus signature databases often have correlated patterns to match. For example, Figure 1.a shows an MS-SQL worm detection rule, which requires matching 4 patterns sequentially. The rule in Figure 1.b is another example where the system seeks the first pattern "USER". If it does not detect a return key (|0a|) within the next 50 bytes, it will raise an intrusion alarm for overflow attack attempt. A large number of these complicated patterns make it hard for pure software-based pattern matching algorithms to keep up with line rate. The SNORT system, for example, implements pattern matching algorithms in software. It can handle link rates only up to 100Mbps [2] under normal traffic conditions and worst case performance is even less. These rates are not sufficient to meet the needs of even medium-speed access or edge networks. Since worms and viruses may possibly originate inside the network, NIDS are also required to scan packets inside the network, which is usually gigabit rates or higher.

```
content:"|04|"; depth:1;
content:"|81 F1 03 01 04
9B 81 F1 01|";
content:"sock";
content:"send"
```

1.a: MS-SQL Worm detection.

```
content:"USER"; nocase;
content:!"|0a|"; within:50;
```

1.b: POP3 User Overflow Attempt

Figure 1. Example patterns from SNORT rules.

To operate SNORT-like intrusion detection systems at multi-gigabit rates using hardware acceleration, one possibility is to use Ternary Content Addressable Memories (TCAM). TCAMs are widely used for IP header based processing such as longest prefix match. Because of their intrinsic parallel search capability, TCAMs can also be used effectively for the pattern matching functions needed in intrusion detection systems. However, TCAMs impose limitations on the pattern length that can be directly matched. Also, there is

no direct method to handle correlated patterns such as the example patterns shown in Figure 1. In this paper, we develop algorithms that use TCAMs to achieve high speeds while not being restricted to these limitations.

Our work is applicable to other layer 7 pattern matching problems. For example, applications like HTTP load balancing, email SPAM filtering, etc., require packet payload scanning. In this paper, we develop and test algorithms for intrusion detection and anti-virus signature sets because they are more complex than the signatures of the applications listed above.

The remainder of this paper is organized as follows: we review related work and summarize the relevant TCAM background in Section 2. We define a generalized pattern format based on the analysis of different signature sets in Section 3. Section 4 presents our scheme to map the multiple patterns into TCAM and efficiently scan packets at high speeds. Section 5 analyzes our scheme and discusses strategies against malicious attacks. Possibilities to speed up pattern matching to multi-gigabit rate are shown in Section 6. We present simulation studies on both the real world traces and synthesized worst case traffic in Section 7 and finally conclude the paper in Section 8.

2. Related Work

Pattern matching problems have been extensively studied. In this section, we only discuss approaches that are relevant to the intrusion detection problem. We first review representative software-based schemes and then discuss FPGA and Bloom Filter based schemes that are amenable to hardware implementation. Finally we introduce TCAMs and related work that uses TCAMs.

Algorithms for software-only schemes

The most influential software-only algorithms are: Knuth-Morris-Pratt(KMP), Boyer-Moore, Aho-Corasick, and Commentz-Walter [3] .

The KMP and Boyer-Moore algorithms are designed for single pattern searching. They build skip tables to avoid back tracking and to help shift forward. The search time for an m bytes pattern in a n bytes packet is $O(n+m)$. If there are k patterns, the search time is $O(k(n+m))$, which grows linearly to k.

The Aho-Corasick and Commentz-Walter algorithms match multiple patterns simultaneously. They both pre-process the patterns and build a finite automaton which can process the input packet in $O(n)$ time. Although both algorithms are fast, they suffer from an exponential state explosion. One of the network intrusion detection systems, Bro [4], uses a similar deterministic finite automaton based approach. Bro generates so many states that only a part of the automaton is kept in memory. The system dynamically extends the automaton based on runtime information. This degrades the system performance.

Recently, new pattern matching algorithms specifically for content-based packet handling have been proposed. The Aho-Corasick-Boyer-Moore (AC_BM) algorithm proposed by Silicon Defense [5] combines the Boyer-Moore and Aho-Corasick algorithms. Another new algorithm is the Setwise Boyer-Moore-Horspool algorithm by Fish et al. [6], whose average case performance is better than Aho-Corasick and Boyer-Moore. These algorithms greatly improve SNORT's pattern matching speed. However, it is still below the line rate needed for network deployment.

FPGA solutions

FPGAs can be programmed for fast pattern matching due to their exploitation of reconfigurable hardware capability and their ability for parallelism. To search for a regular expression of length n on an FPGA, one solution is to build a serial machine that requires $O(2^n)$ memory and takes $O(1)$ time per text character. Sidhu et al. proposed a Nondeterministic Finite Automaton (NFA) approach using FPGAs [7]. Their approach requires only $O(n^2)$ space and is still able to process each text character in $O(1)$ time.

The above two approaches are optimized for single keyword searching and do not scale well for multiple patterns. The recent work by Sidhu et al. uses a modified KMP algorithm [8]. Each pattern is still treated independently; however, multiple (100 reported in the paper) patterns can be pipelined at gigabit rates. The main concern is that patterns are searched sequentially, so the overall latency increases proportionally with the number of patterns.

Bloom filter solutions

Dharmapurikar et al. proposed a multiple-pattern matching solution using parallel bloom filters [9]. Their approach can handle thousands of patterns. The proposed scheme builds a bloom filter for each possible pattern length. This could impose parallelism limits in some virus databases because pattern lengths vary from tens to thousands of bytes and there are hundreds of possible patterns lengths.

TCAM solutions

Ternary Content Addressable Memory (TCAM) is a type of memory that can perform parallel search at high speeds. A TCAM consists of a set of entries. The top entry of the TCAM has the smallest index and the bottom entry has the largest. Each entry is a bit vector of cells, where every cell can store one bit. Therefore, a TCAM entry can be used to store a string.

A TCAM works as follows: given an input string, it compares this string against all entries in its memory

in parallel, and reports one entry that matches the input. The lookup time (e.g., 4 ns [12]) is deterministic for any input. Unlike a binary CAM, which only has two states: 0 or 1, each cell in a TCAM can take one of the three states: 0, 1, or '?' (do not care). With the 'do not care' state, TCAMs can be used for matching variable prefix CIDR IP addresses and thus can be used in high-speed IP lookups [10, 11]. Also because of the 'do not care' state, one input may match multiple TCAM entries. In this paper, we assume the use of the widely-adopted first-match TCAM, which gives out the lowest index match of the input string if there are multiple matches as shown in Figure 2.

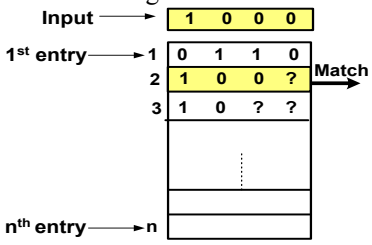


Figure 2. TCAM.

Single-chip densities of TCAMs are approaching 2MBs [12]. The width of each entry can be configured according to user requirements. For example, a 1M TCAM can be programmed as 64K entries with 16 bytes per entry, or 1K entry with 1K bytes per entry etc.

Binary CAM is proposed as a pattern matching solution [13]. The space needed for k patterns each with w bytes is just kw bytes of CAM space. To search a packet of length n, it can provide an answer in a deterministic time of $O(n)$ CAM lookups. The proposed approach is for simple patterns of length equal to CAM width. In this paper, we will present algorithms for arbitrary long patterns and other complex patterns.

3. Problem definition

We are given a set of k patterns, $\{P_1, P_2, ..., P_k\}$. When $k=1$, we have a *single-pattern matching* problem. When $k>1$, we have a *multiple-pattern matching* problem. In this paper, k is usually a large number (e.g., thousands). Given a packet of length n, our goal is to report all the matching patterns in the packet. In this paper, we will concentrate on two categories of patterns, namely simple patterns and composite patterns.

3.1. Simple patterns

A simple pattern P of m bytes can be written as $P = b^1 b^2 ... b^m$, where each b^i represents a byte. The pattern length m can be different for each pattern. b^i can be of two forms: a deterministic form or a non-deterministic form. For the deterministic form, every bit of b^i is either 0 or 1. It is one **specific** value of the 2^8=256 possible values. For example, b^i= 0100 0001 (letter a). For the non-deterministic form, b^i can be **any** value in a domain. Following are two kinds of domains:

1. Case insensitive alphabets: $b^i = \{a, A\}$,…., $b^i = \{z, Z\}$.
2. Wildcard byte (*): b^i could be any 2^8 possible values.

3.2. Composite patterns

Simple patterns can be extended to form composite patterns like those in Figure 1. From several virus and worm signature databases, especially SNORT [2], we identify the following two types of composite patterns:

1. Negation (!). *!P* denotes no appearance of pattern P.
2. Correlated patterns. If P_1 and P_2 are two patterns, $P_3 = P_1 * P_2$ is a correlated pattern. This pattern requires matching P_1 first, then some arbitrary content *, and finally matching pattern P_2. Note that * can have infinite length, but usually we will put a length limitation on it, e.g., equal to four bytes, less than four bytes etc.

3.3. Comparison with regular expressions

Simple and composite patterns can be mapped into Perl Compatible Regular Expressions (PCRE) [14]. The case insensitive can be mapped to the "/i" modifier. The wildcard byte corresponds to "." meta-character. For composite patterns: the negation corresponds to "!" syntax and the correlated patterns can be expressed with ".*" meta-character between patterns.

Our pattern definition is a subset of PCRE. For example, we do not directly support matching *"or"* relationship as a regular expression {A|B} has to be split into two separate patterns in our pattern definition: pattern "A" or pattern "B". We made this restriction because patterns used in packet processing are a small subset of regular expressions. Therefore, we extract only commonly used pattern formats from the regular expression. By doing this, the pattern matching process can be much simpler and faster. We will show later in Section 7 that the pattern expressions we extracted can express all the patterns in our pattern database.

4. Multiple-Pattern Matching with TCAM

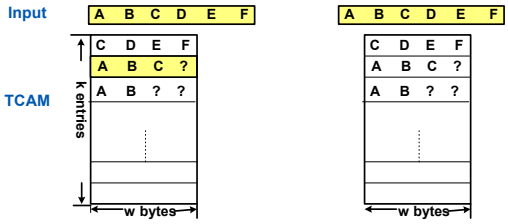
In this section, we present our TCAM-based pattern-matching algorithm. We begin with a simple case where the patterns are shorter than the TCAM width w. Then in Section 4.2, we describe our solutions for handling long patterns. In Section 4.3, we extend our algorithms to handle correlated patterns.

4.1. Solution for simple patterns

Suppose the width of the TCAM is *w* bytes. Let us first look at the simple case where all the patterns are simple deterministic patterns, with length shorter or equal to *w* bytes. Our solution is simply putting the patterns into TCAM, each occupying one entry. If a pattern is shorter than *w* bytes, then we pad it with "?" (do not care) bits. For ease of explanation, in the sequel, we use alphabet characters rather than binary forms as pattern examples. We assume that each character is one byte.

Patterns should be organized according to their lengths in descending order. This is because a TCAM only reports the first matching result if there are multiple matches and we want to identify all matching patterns. For example, if a pattern "ABC" is put in a lower index (top end) in TCAM, matching of the "ABC" includes matching of a shorter pattern "AB". If we place patterns in the other order, we can not infer the matching of the longer pattern from matching the shorter pattern. Thus we may miss out some matching results.

The process of finding patterns in a packet is as follows: The first *w* bytes in the packet are mapped into TCAM (Figure 3.a). If there is a hit, then report the matched pattern. Next, shift one byte and check TCAM again as shown in Figure 3.b. This process is repeated until we have read the whole packet. Note that when we are at the end of the packet and the remaining packet size *t* is less than the TCAM width, we can pad it with all 0s and look it up in TCAM. However, only patterns less than *t* bytes should be reported as matches.



3.a First Position **3.b Second Position**

Figure 3. Scanning Process.

One TCAM lookup is needed for every byte position in the packet. Assuming the TCAM lookup time is 4 ns, it can support a deterministic scan rate of 8 bits/4 ns =2Gbps against thousands of patterns and is able to report all the matching patterns.

4.2. Long Patterns

We can configure the TCAM width to be larger or equal to the longest pattern length. However this will waste TCAM resources as most of the short patterns have to be padded with the 'do not care' states to reach the TCAM width. As TCAMs are relatively small, wasting resources by padding is not a good approach.

A better solution is to set the TCAM width smaller and cut some of the long patterns into shorter patterns to save TCAM space. At this point, let us assume that we have identified a good TCAM width *w* for the given signature set. Table 1 shows a pattern set with four patterns and the TCAM width is set to be four bytes. For long patterns (Patterns 1-3) that are cut into shorter patterns, we call the first *w* bytes *prefix pattern*s and the remaining part *suffix pattern*s. Patterns shorter than *w* bytes (Patten 4) remain intact. The selection of *w* and the tradeoffs between a single cycle "long" match and many cycles of "short" matches will be discussed in Section 5.

Table 1. Long Pattern Examples.

Pattern Index	Pattern Contents	Prefix Patterns	Suffix Patterns
1	ABCDABCD	ABCD	ABCD
2	DEFGABCDL	DEFG	ABCDL
3	DEFGDEF	DEFG	DEF
4	DEF	-	-

4.2.1. Mapping Long Patterns into TCAM

Prefix patterns can be fit into TCAM directly since they are *w* bytes. After matching a prefix pattern, we can use software to compare the suffix patterns. However, there may be multiple suffix patterns sharing the same prefix pattern like pattern 2 and 3 in Table 1. In such cases, the computation costs for software comparisons are still high.

We choose to put the suffix patterns into the same TCAM as well. If a suffix pattern is longer than *w* bytes, we need to cut it into multiple sub-patterns of *w* bytes each. The suffix patterns can be less than *w* bytes, or not exactly a multiple of *w* bytes. They will generate very short sub-patterns. We can pad those short patterns with 'do not care' states to make them *w* bytes. The problem with this approach is that small patterns that are only one or two bytes greatly increase the probability of TCAM hits, thus demand a lot of processing.

To overcome this problem, we pad it in the front by the tail of previous pattern. For example, the suffix pattern of "DEFGDEF" is "DEF". We pad it with the tail of previous prefix pattern ("G") to make it exactly *w* bytes ("GDEF"). Another example is "DEFGABCDL", the suffix pattern "ABCDL" is divided into two suffix patterns of *w* bytes each: "ABCD" and "BCDL".

We can order the unique prefix patterns and suffix patterns in any order because they all have the length *w*, so none of them covers another unless they are identical. For patterns shorter than *w* bytes (e.g., pattern 4 in Table 1), similar as before, we will pad them with 'do not care' at the end and sort them according to the descending order of lengths. Table 2 shows TCAM layout for patterns in Table 1.

The overall process of matching long patterns is as follows: if we match a prefix pattern at the i^{th} position

of the packet, we record it in memory. Later, if we match a suffix pattern at position *i+ j (0 < j <=w),* check whether the concatenation of this suffix pattern and the previous prefix pattern forms a long pattern. We will discuss this process in detail in Section 4.2.3.

4.2.2. Data Structures in Memory

There are three data structures to be stored in memory (e.g., SRAM) for matching long patterns.

A. Pattern Table

All the patterns (simple, prefix, and suffix patterns) are put into a single TCAM. So, when matching an entry in TCAM, we need to check what kind of matching it is. Pattern table (Table 3) records such information. Each line in the table is correlated with one TCAM entry.

The second column records whether it is a matching of a simple pattern. For example, from a hit of "DEFG", we can infer a matching of "DEF".

The third column shows prefix pattern information. Positive number illustrates a valid pattern and "-1" indicates otherwise. Since not every entry in TCAM is related to a prefix pattern, we use a compressed index to store the index of prefix patterns. At the end of this section, we will show how this compressed index can help reduce memory consumption. Column four stores the compressed index for suffix patterns. Note that the compress index for suffix patterns is separately built and thus is independent of the prefix pattern index.

B. Partial Hit List (PHL)

When matching a prefix pattern, we need to record it in memory. We call the data structure a partial hit list as shown in Table 4. For example, when matching pattern "ABCD" at the first four bytes of the packet, we record the compressed index (1 in this example) and the starting position of the pattern in the packet (1st byte in this example) in the PHL.

C. Matching Table

Having identified prefix and suffix patterns with TCAM hits, next we assemble the prefix with the corresponding suffix patterns to recover valid long patterns. Matching table (Table 5) stores all the valid combinations. The combination of prefix pattern "DEFG" and suffix pattern "ABCD" yields a new prefix pattern (annotated by 3*). We give the new prefix pattern ("DEFGABCD") index 3 as the compressed prefix pattern index and store it back to the PHL. Later, when we match suffix pattern "BCDL" at the next position, we can lookup the matching table and conclude that we have matched pattern "DEGFHIJKL".

The lookup process appears to be complicated as we need to search through the mapping table to check whether one combination is valid. In a real system, we can trade space for speed. The first three columns can be used as indices of a three-dimensional array and we do not need to store those columns. Only matched long

Table 2. Patterns in the TCAM.

TCAM Index	Content
1	ABCD
2	DEFG
3	BCDL
4	GDEF
5	DEF?

Table 3. Combined Pattern Table.

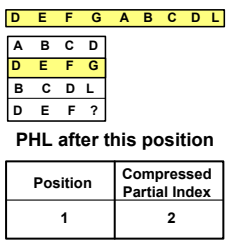
Index (Content)	Simple Pattern Index	Prefix Index	Suffix Index
1(ABCD)	-1	1	1
2(DEFG)	4(DEF)	2	-1
3(BCDL)	-1	-1	2
4(GDEF)	-1	-1	3
5(DEF ?)	4(DEF)	-1	-1

Table 4. Partial Hit List.

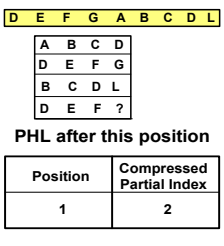
Compressed Index	Position
1	1

Table 5. Matching Table.

Prefix Index	Suffix Index	Distance	Matched Long Pattern Index
1(ABCD)	1(ABCD)	4	1(ABCDABCD)
2(DEFG)	1(ABCD)	4	3*(DEGFABCD)
2(DEFG)	3(GDEF)	3	3(DEGFDEF)
3(DEGFABCD)	1(ABCD)	4	1(ABCDABCD)
3(DEGFABCD)	2(BCDL)	1	2(DEFDABCDL)

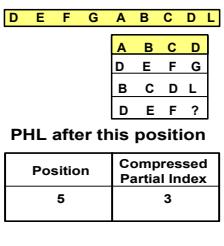


Position 1: Match "DEGF".
Report short pattern "DEF"
It is a suffix pattern. But PHL was empty, so no long pattern is found at this position.
It is also a prefix pattern with compressed index 2, so insert such information to the PHL.

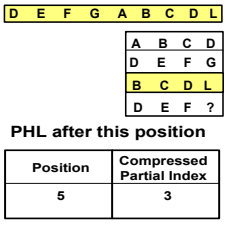


Position 2: No match.

No match for position 3 and 4 either. So, these two positions are omitted in the figure.



Position 5: Match "ABCD". No short pattern.
It is a suffix pattern. Combined with prefix pattern 2 in the PHL yields another prefix pattern "DEFGABCD" (compressed prefix index is 3 as shown in the mapping table with *). Insert it into PHL. The old item (1, 2) can now be deleted since it is *w* position away from the next position.
It is prefix pattern "ABCD", but it is included by "DEFGABCD". We will not insert it into PHL.



Position 6: Match "BCDL".
Imply no short pattern.
This is a suffix pattern. Combining with "DEFGABCD" in the PHL, report a long pattern "DEGFABCDL".
This is not a prefix item.

Figure 4. An Example of Matching Long Patterns in an Input String "DEFGABCDL".

patterns at the corresponding indexed space are stored and 'no match' (-1) is placed at all the other invalid combination places. In this manner, we can decide whether a combination is valid or not with only one memory lookup. The total memory consumption is *a*b*w*, where *a* is the compressed prefix index size, *b* is the compressed suffix pattern index size. Here we see that the compressed index can help save memory consumption for the matching table.

4.2.3 Algorithms for Long Patterns

We use an example (Figure 4) to walk through the algorithm. Suppose the input packet is "DEFGABCDL" and we want to search for patterns in Table 1.

The initial partial hit list (PHL) is set to be empty. The algorithm looks up the first *w* bytes of the packet payload in TCAM and then shifts one byte at a time. At each position, if it matches a TCAM entry (e.g., "DEFG" in the first position and "ABCD" in the fifth position), it will consult the combined pattern table and do the following three steps. First, it will check whether the matched entry implies simple patterns (e.g., pattern "DEF" in the first position). Second, if it is a suffix pattern, it needs to check whether the combination with any prefix pattern in PHL forms a valid long pattern through consulting the mapping table (e.g., in position 6, "BCDL" combined with prefix pattern in PHL generates a long pattern "DEFGABCDL"). In the case that they form another prefix pattern, we need to add the new prefix pattern back to the PHL (e.g., in position 5, "ABCD" combined with previously matched "DEFG" forms another prefix pattern "DEFGABCD"). Third, if it is a prefix pattern, the algorithm inserts it into the PHL if it is not inserted by the previous step (e.g., "DEFG" in the first position). Note that, in position 5, "ABCD" is not inserted because "DEGFABCD" is inserted in the previous step, which implies "ABCD".

No matter what packet position we are at, the size of PHL is bounded by the TCAM width *w*. This is because at each position, we only insert one item into PHL. In addition, suffix patterns must immediately follow the prefix patterns to form long patterns, so we can delete the prefix patterns that are *w* bytes away.

4.3. Solution to Composite Patterns

Simple patterns are not sufficient for intrusion detection systems such as SNORT. In this section, we extend our algorithm to handle composite patterns.

4.3.1. Correlated Patterns

Correlated patterns denote series of patterns, i.e., patterns followed by other patterns like "ABCD" followed by pattern "DEFG" within 4 bytes from the end of the first pattern. We call the patterns in the pattern series sub-patterns (e.g., "ABCD" and "DEFG"). Matching a correlated pattern is similar to a long pattern: a long pattern is just several sub-patterns and the distance of these patterns must be exactly *w*. Hence, the scheme for long patterns can be extended to correlated patterns. The matched sub-patterns are also recorded in the PHL. The only difference is that the partial hit record for sub-patterns cannot be removed after *w* positions because the distance between two sub-patterns can be larger than *w*.

4.3.2. Patterns with Negations

Negation of a pattern stands for no occurrence of the patterns. This is usually the second sub-pattern correlated with another pattern. For example, *content : "USER" ; nocase ; content : !"|0a|" ; within: 50.* This pattern shows that if we see pattern "USER", then we want to find "|0a|" (return) in the next 50 bytes. Otherwise it is an abnormal packet.

The identification of negation of a pattern is similar to correlated patterns. After matching the first sub-pattern "USER", put it in the PHL. In the next 50 bytes, inspect whether there is a hit for "|0a|". If yes, those two matches will generate a good match and we will remove the index for "USER" from the PHL. Otherwise, after 50 bytes, we will remove the "USER" from the PHL and report a hit of pattern "USER" and !"|0a|".

4.3.3. Patterns with Wildcards

Some signature databases specify patterns with "no case" keyword. This means that either a upper case or a lower case of the pattern is considered valid. Thanks to the coding of ASCII, the distance between a lower case character and its corresponding upper case character is 32. It is a power of 2, so TCAMs can support it easily with a 'do not care' bit. For example, the ASCII code for letter "A" is 65 (binary form 0100 0001) and letter "a" is 97 (binary form 0110 0001). So we can represent case insensitive letter *a* in the TCAM as (01?0 0001). If there is a requirement for a fixed width wildcard for any characters, then we can just put 'do not care' states in all their corresponding positions in the TCAM.

5. Analysis of the Scheme

The scheme of Section 4 covers the key pattern formats for building anti-virus and intrusion detection systems. In this section, we will analyze the performance of the proposed scheme using two metrics. The first metric is the memory consumption. We want to minimize the memory consumption for SRAM and especially TCAM as it is expensive with current manufacturing technologies. The second metric is pattern scanning rate. It is affected by the number of TCAM hits since each TCAM hit requires one memory lookup of the pattern

table. The size of PHL also influences the scan rate because we need to access mapping table once for each item in PHL for matched suffix pattern.

Our analysis is three folds: (1) the impact of the TCAM width on the scheme, (2) the impact of memory lookups on the system scan rate, and (3) how to avoid malicious packets that are aimed at slowing down the system.

5.1. Analysis of the TCAM Width

As we mentioned in Section 2, the width (*w*) of TCAM is configurable. Here we analyze the impacts of the TCAM width on matching long patterns. We will perform analysis of correlated pattern in Section 5.3.

Effects on TCAM Space

Suppose we have a total of *k* patterns, each with m_i bytes. If we set the TCAM width to *w,* then each pattern will be cut into $\lceil m_i / w \rceil$ prefix or suffix patterns, where $\lceil\ \rceil$ denotes rounding up. The total TCAM space required to accommodate all these patterns is $w*\sum\lceil m_i / w \rceil$. It increases as *w* increases because short patterns and suffix patterns need to be padded to the TCAM width.

Effects on Memory Space for Mapping Table

The compressed prefix pattern index is $\sum_i(\lceil m_i / w \rceil - 1)$, which can be proved through two cases. Case 1: If a pattern is shorter than *w* bytes, it has no prefix pattern. This agrees with $\lceil m_i / w \rceil - 1 = 0$. Case 2: For a pattern longer than *w,* although there is one prefix pattern, it will generate new prefix patterns after intermediate suffix patterns are matched. Hence, we also need $\lceil m_i / w \rceil - 1$ indices. So, independent of the pattern length, the number of prefix pattern indices is $\lceil m_i / w \rceil - 1$ per pattern and the total is $\sum_i(\lceil m_i / w \rceil - 1)$. Similarly, the size of compressed suffix pattern index is also $\sum_i(\lceil m_i / w \rceil - 1)$. Hence, the memory space for the mapping table is $w*(\sum_i(\lceil m_i / w \rceil - 1))^2$, which decreases as *w* increases.

Chances of TCAM Hits

We distinguish two types of TCAM hits. One is a "real" hit, which can report a matched pattern. For a pattern longer than *w*, we define the matching of the last suffix pattern as a real hit. The other type of hit is an "associate" hit—an intermediate hit that may lead to a real hit, i.e., each prefix pattern hit is an associate hit. Associate hits incur extra computation, so next we want to analyze the probability of associate hits. Assume packet contents are random, for any *w* bytes in the packet, there are $(2^8)^w$ possible values and the chance of matching one particular pattern of length *w* is $1/(2^8)^w$. As we have analyzed before, there are $\sum_i(\lceil m_i / w \rceil - 1)$ prefix patterns of *w* bytes each and assume they are independent, the chance of having an "associated" hit at each position is $\frac{\sum_i(\lceil m_i / w \rceil - 1)}{(2^8)^w}$, which decreases dramatically when *w* increases. For example, suppose we have 2000 patterns of 200 bytes each. Setting *w* to be 4 bytes, the associate hit rate at each position is 2.2e-5. If w is 8, it is 2.6e-15, which is very low.

Effects on PHL size

As discussed in Section 4.2.3, PHL has an upper bound of *w* for pattern sets that contain long patterns only. If we have a matching at position *i,* it imposes extra constraint on matching at position *i+j (j<w)* because the last *w-j* bytes of the first position must overlap with the first *w-j* bytes of position *i+j*. Assume that patterns are independent, the number of overlapped pattern pair is small. If we relax this constraint and assume these positions are independent, the expected prefix pattern list size: $w*\frac{\sum_i(\lceil m_i / w \rceil - 1)}{(2^8)^w}$, which decreases to zero quickly as *w* grows. For example, suppose there are 2000 patterns with 200 bytes each. If *w* is set to be 4 bytes, the expected size of PHL at each position is 8.8e-5. If w is 8, it is 2e-14, which is well below 1. We will analyze the PHL for correlated patterns in the Section 5.3.

Summary on the impacts of TCAM width

The above analyses show that a small *w* can save TCAM space. However, a small *w* also generates many prefix and suffix patterns, which results in a large mapping table. In addition, since each entry in the TCAM is small, it will report many matches, create a large PHL and require many matching table lookups. Therefore, if there is enough TCAM space, we should set *w* larger than most of the pattern sizes and allow only a very small number of patterns to be cut into prefix and suffix patterns.

5.2. Analysis of Memory Lookups

Memory lookups are usually the bottleneck of packet processing systems. There are two types of memory lookups in our scheme: TCAM lookups and regular memory (e.g., SRAM) lookups in the combined pattern table and matching table. The process of TCAM lookups and the memory lookups can be pipelined. We can perform the memory lookups for current position while consulting TCAM with the data at next position. Suppose we have a packet of *n* bytes and the TCAM lookup

time is *a* for each lookup, we will have a deterministic TCAM processing time of *n*a.*

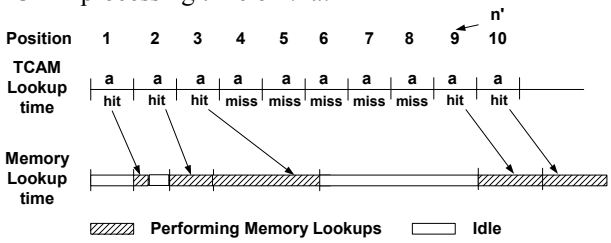


Figure 5. Memory Lookup Process.

If the TCAM reports a miss, no extra memory lookup will be initiated in this position *i* and the memory lookup process is idle. Otherwise, the proposed scheme will first perform one memory lookup in the combined pattern table. If the matched pattern is a valid suffix pattern and there are j_i items in the current PHL, we need another j_i memory lookups in the matching table. Hence, a maximum of j_i+1 memory lookups will be issued for a TCAM hit. The memory lookup time may be shorter or longer than the TCAM lookup time, thus the memory lookup process may be backlogged. For example, in Figure 5, positions 1, 2, and 3 all have TCAM hits, so the memory lookup process is kept busy for a while. Later when there are some TCAM misses, the memory lookup process can catch up with the TCAM lookup speed. Therefore, only the last memory backlog position (*n'*) is important. The overall packet scan time is the sum of the time needed for the TCAM accesses up to this position (*n'*a*) and the memory lookup time after this position ($\sum_{i=n'}^{n}(j_i+1)*h_i$). j_i is usually a very small number (<1) as we analyzed in Section 5.1. If the TCAM hit rate (h_i) is low, the second term $\sum_{i=n'}^{n}(j_i+1)*h_i$ is small. In such a case, the speed of the pattern matching is dominated by the TCAM lookup time. Assuming TCAM lookup time is 4ns, the total time to scan an *n* bytes packet is *4n* ns. This yields a matching speed of *8*n/4n = 2* Gbps if we have a small TCAM hit rate and PHL size.

5.3. Protection against Malicious Attacks

For long patterns, we can discard the partial hit results that are *w* positions away. This assumption does not hold for correlated patterns, as the distance between two sub-patterns can be larger than *w*. In addition, each sub-patterns can be smaller than *w* bytes, which generates higher TCAM hit rate than longer patterns. Intruders may intentionally send packets that cause a lot of partial hits for the correlated pattern to create a long PHL. Later, when a suffix pattern is matched, a large number of memory lookups have to be issued and the system performance degrades dramatically.

To deal with this kind of attack, we study the size of PHL for correlated patterns. First we answer the question: if we match a pattern of length *m* at position *i,* what is the chance that we can construct an input to match another pattern at position *i+j?* If *j* is one, it means matching two patterns one byte apart. Such probability is low because it requires the first *m-1* bytes of the second pattern are the same as the last *m-1* bytes of the first pattern. It can be proved that given *k* independent patterns, the probability is $1-((2^8)^{m-1})!/(((2^8)^{m-1}-k)!*((2^8)^{m-1})^k)$. For example, *k = 1000* and *m=4*, it is 0.029, which is low.

When $j = m$ and two patterns do not overlap, intruders can pack the sub-patterns consecutively to form an *n* bytes packet. This packet generates matches at every *n/m* positions, where *m* is the shortest sub-pattern length. Thus the PHL can have *n/m* items or more. To limit the PHL size, we recommend limiting the max distance between two sub-patterns to be considered as correlated. This recommendation is reasonable because in practice, patterns very far apart are unlikely to be considered correlated.

6. Speedup to Multi-gigabit Rates

Speeds higher than 2Gbps can be achieved in multiple ways. A brute force solution is processing packets in parallel by using multiple TCAMs. A better option uses only one TCAM but exams two possible positions and shifts two bytes at a time. The longest pattern that can fit into a TCAM is no longer *w* bytes, but *w-1* bytes. We illustrate this scheme with a simple example as shown in Figure 6. Suppose *w=5* and there are three patterns: "ABCD", "BCDE", and "HIJK". We add 'do not care' after these three patterns and put them into the TCAM to match the packet payload from position *i* to *i+w-2*. In addition, we add 'do not care' before these patterns to match the packet payload from position *i+1, i+w-1*. If two patterns overlap by one byte (e.g. "ABCD" and "BCDE"), we create a new pattern that covers both patterns ("ABCDE"). This pattern should be placed at the top of the TCAM. In this case, matching "ABCDE" means matching "ABCD" from position *i* to *i+w-2* and "BCDE" from position *i* to *i+w-1*. Not that the probability of an overlapped pattern is small when *w* is large.

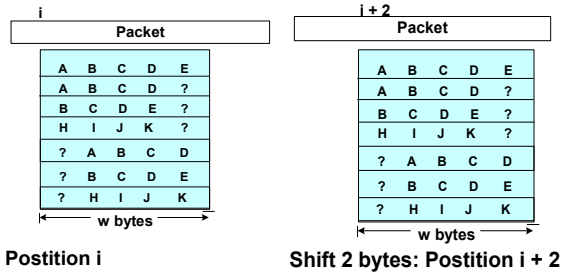


Figure 6. Speedup to Multi-Gigabit Rate.

This scheme can be generalized to get a speedup that is l times faster or even more. If there is no overlapped pattern, we need l times as many TCAM entries for a speed up l. The maximum pattern length is decreased by l bytes. The size of mapping table increases approximately l^2 times as both the number of prefix patterns and suffix patterns are increased approximately l times.

7. Simulation Results

7.1. Methodology

We select two complex pattern sets. The first is a virus signature set from ClamAV [1], which contains simple patterns only. The second is from the SNORT [2] intrusion detection system with many correlated patterns.

We use two sets of real packet traces and a synthesized data trace for pattern scanning. The first real trace set is the intrusion detection evaluation data set from an MIT DARPA project [15] that has more than a million packets. The second real trace is from the Berkeley research group's local traffic, with more than six million packets. The synthetic data trace is generated by randomly inserting patterns into the packet payload.

For all the test traces we record the average and maximum PHL size for each packet. We used three metrics for the PHL size over the whole trace data. *Avg* is the mean of the average PHL size over all packets. *AvgMax* denotes the mean of the maximum PHL sizes. *Max* records the maximum size over all packets, which denotes the maximum number of entries in memory for extreme cases.

7.2. Results on ClamAV Pattern Set

ClamAV (version 0.15) has 1768 simple patterns. The average pattern length is 55 bytes. Figure 7 plots the distribution of the pattern length. It varies from 6 bytes to 2189 bytes.

With such a large variance in pattern lengths, selection of the TCAM width w is critical. Figure 8 shows the total TCAM space needed to accommodate all the patterns under different w settings. As w increases, TCAM space requirement increases too because of the padding for short and suffix patterns. This agrees with our analysis in Section 5.1. The size of mapping table to be kept in memory, however, is negatively correlated to TCAM space. When w is small, it generates many prefix and suffix patterns. Therefore, the number of prefix and suffix indices grows, which results in high memory consumption for the mapping table. We recommend setting w as 128 bytes and using a 240KB TCAM.

7.2.1. Test Results on Real Data

Table 6 shows the PHL size for both the MIT and Berkeley traces. Since these two trances don't contain many viruses, the PHL size is extremely low when the window size is reasonably large. When the size of PHL is small, the memory lookup process is mostly idle and the system performance is bounded by the TCAM access rate only. So, we can achieve 2Gbps rate with a 240KB TCAM.

7.2.2. Test on Synthesized "Worst-case" Packets

We generated four sets of synthesized data, each with 1, 10, and 100 randomly inserted virus patterns per packet

TCAM Width	MIT Dump			Berkeley Dump		
	Avg	*AvgMax*	*Max*	*Avg*	*AvgMax*	*Max*
4	0.042	0.27	4	0.03	0.48	4
8	4.8e-6	5.6e-4	8	1.e-6	1.9e-5	7
16	0	0	0	4.3e-7	5.8e-6	3
32	0	0	0	0	0	0
64	0	0	0	0	0	0
128	0	0	0	0	0	0

Table 6. PHL Size for ClamAV Signature Set.

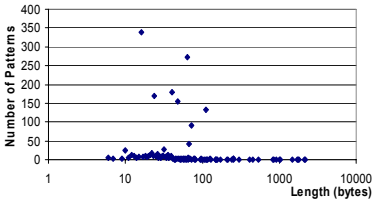


Figure 7. Distribution of Pattern Length

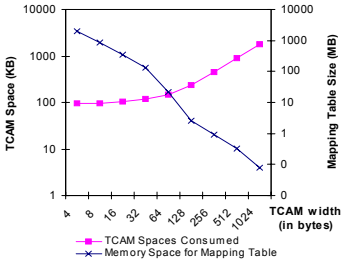


Figure 8. Impact of TCAM Width

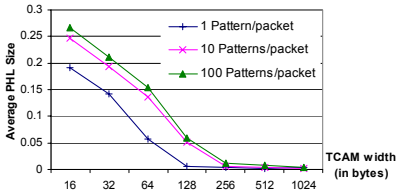


Figure 9. *Avg* of PHL Size.

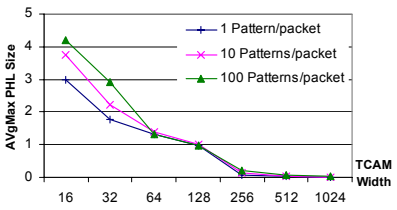


Figure 10. *AvgMax* PHL Size.

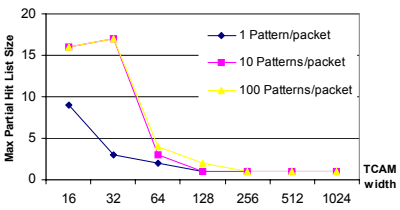


Figure 11. *Max* PHL Size.

respectively. Compared with the real world traces, synthesized data sets yield a larger PHL size. Figure 9 shows the *Avg* PHL size. It decreases quickly as we increase the TCAM width *w*. In addition, having multiple patterns in the packets does not increase the PHL size dramatically. This is because we can delete the prefix patterns that are *w* bytes ahead, so the number of patterns in a *w* bytes window may not increase proportionally to the increase of the total number of patterns per packet.

The *AvgMax* PHL size per packet is notably larger than *Avg* as plotted in Figure 10. This shows that some contents in the packets cause backlogs in the memory lookup process. The effect of the TCAM width again has great impact—if we set *w* to be 128 bytes or longer, the *AvgMax* PHL size is around one per packet. This means that even with this "worst-case" data, the memory lookup process can still finish within one or two cycles after the TCAM lookup process finishes. This has the same effect as having the packet one or two bytes longer. Given the fact that packets consist of at least tens of bytes and we do not need to perform pattern matching on the packet header, this impact of slightly increasing the "effective packet length" for matching purposes is negligible. Hence the packet scan rate is still 2Gbps over this set of synthesized data.

Figure 11 illustrates the *Max* PHL size over all packets. When *w* is small (e.g., 16), the maximum is small because the *max* PHL size is bounded by *w*. When *w* gets larger, the PHL size increases and then drops quickly because the probability of a TCAM hit becomes small for a large *w*. There is a big difference in the PHL sizes between packets containing 1 virus pattern and 10 patterns. However, as we increase the number of viruses per packets, the growth of max PHL size slows down.

7.3. Results on SNORT Pattern Set

The current version of SNORT system (v2.1.2) contains 1991 rules, of which 1836 contain string patterns. The lengths of patterns are much shorter than the ClamAV signature set: mostly from 10 bytes to 100 bytes. In addition, there is a noticeable amount of short patterns of one or four bytes. Among the patterns, 1039 are simple patterns and 527 are correlated patterns with up to seven sub-patterns in one correlated pattern.

We set the TCAM width to 128 bytes as it covers most of the SNORT patterns. We successfully expressed all SNORT patterns using our pattern definition in Section 3. For patterns containing PCRE [14], we converted most of them into correlated patterns. For those of the PCRE patterns requiring matching word boundary, we add all the combination of '\t', '\n', and ' ' before and after the pattern. After conversion, all the patterns can be mapped into a TCAM size of 295KB.

Since SNORT has correlated patterns, we first tested the impact of different window sizes from 20 to 200 bytes. Compared to ClamAV, the PHL size is much larger as is shown in Table 7. We believe this is because the SNORT signature set contains a lot of short patterns. In addition, the size of PHL increases when the window size increases because it needs to keep the partial hit information longer. However, as the window size grows larger, growth of PHL is slower.

Table 7. PHL Size for SNORT Signature Set

Window Size	MIT Dump			Berkeley Dump		
	Avg	AvgMax	Max	Avg	AvgMax	Max
20	0.5523	2.7683	8	0.4702	1.5765	12
40	0.9881	3.5376	14	0.6500	1.8661	18
60	1.3151	3.9960	14	0.7313	1.9652	23
80	1.5491	4.2158	16	0.7587	2.0373	24
100	1.6867	4.3485	18	0.7661	2.0740	25
120	1.7725	4.4475	18	0.7669	2.0768	25
140	1.8308	4.5722	19	0.7669	2.0768	25
160	1.8800	4.6643	19	0.7669	2.0768	25
180	1.9244	4.7386	19	0.7669	2.0768	25
200	1.9662	4.8079	20	0.7669	2.0768	25

A large PHL is problematic since it requires many memory lookups and slows down the system. Therefore, we studied the total scanning time (including memory lookups) vs. the time spent on TCAM lookups only. We call this ***scan ratio***. This ratio is important because TCAMs have a fixed access rate (e.g., 4ns) and generate a constant processing rate (2Gbps). Therefore, we can use them as bases for speed calculation. For example, if the scan ratio is 2, overall system scan rate is 2/2=1Gbps. Figure 12 plots this information. Since memory (e.g., SRAM) access is usually slightly faster than TCAM access rates, we simulated scenarios with different ratios of memory to TCAM access times, (which we call the ***memory ratio***). For example, 1 means memory access speed is equal to TCAM access speed. 0.2 denotes that memory access speed is 5 times the TCAM access rate. Figure 12 shows the impact of memory ratio on the scan ratio, with each curve standing for one memory ratio setup. Value of 1 (y axis) at 60 percent (x axis) stands for that 60% of the packets have a scan ratio of 1. Simulation results show that the scan ratio is less than 1.2 for most of the packet (80%) under all settings. The TCAM access speed is the bottleneck for these packets. For the remaining around 20% of the packets, the memory access process is backlogged and therefore the overall system performance is lower than the TCAM rate. Nevertheless, the max scan

ratio is less than 2 for all setups, which means that we can have a pattern scan rate of at least 1Gbps.

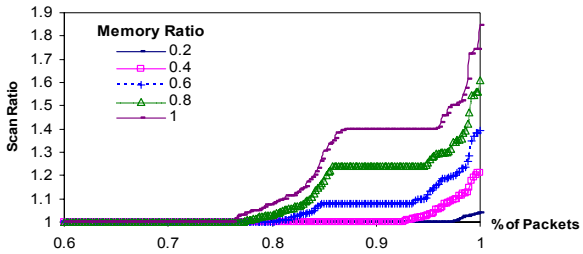


Figure 12. Effects of Memory Ratio on Scan Rate

8. Conclusions

With the increasing importance of network protection from cyber-attacks, it is essential to develop mechanisms for building effective defenses against virus, worm, and denial of service attacks. The rapid rise in link bandwidths implies that network protection mechanisms must be capable of operating at multi-gigabit rates. A key operation for network protection is pattern-matching to check for virus and worm signatures. In this paper, we developed a TCAM based scheme to solve the packet pattern-matching problem. Our proposed scheme can scan thousands of patterns simultaneously at gigabit rates. By evaluating its performance using multiple real-network traces we showed that it is indeed suitable for multi-gigabit operation. The scheme can also be extended to achieve even higher rates with larger TCAMs [15].

Acknolegements

We would like to extend our gratitude to SNORT system developers for implementing the powerful tool and making it open source. We would also like to thank Guozheng Ge, Li Yin, Weidong Cui, Animesh Nandi, Ling Huang, and Yanlei Diao for proof reading. Finally, we want to thank anonymous reviewers and committee members for valuable comments and suggestions.

References

[1] Clam anti virus signature database, www.clamav.net.
[2] SNORT system, www.snort.org .
[3] G. A. Stephen, "String Searching Algorithms," *Lectures Notes Series on Computing*, Vol. 3, 1994
[4] V. Paxon, "Bro: A System for Detecting Network Intruders in Real-Time," Computer Networks, 1999.
[5] 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.
[6] M. Fish and G. Varghese, "Fast Content-Based Packet Handling for Intrusion Detection," *UCSD technical report CS2001-0670,* 2001.
[7] R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching using FPGAs," *In FCCM,* 2001.
[8] Z.K. Barker and V.K. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs," *In FPGA*, 2004.
[9] S. Dharmapurikar, et al., "Implementation of a Deep Packet Inspection Circuit using Parallel Bloom Filters in Reconfigurable Hardware," *In Hot Interconnects*, 2003.
[10] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," *in ICNP*, 2003.
[11] P. Gupta, and N. McKeown, "Algorithms for Packet Classification," *IEEE Network,* 2001.
[12] 5512GLQ TCAM from NetLogic Microsystems
[13] M. Gokhake, et. al, "Granidt: Towards Gigabit Rate Network Intrusion Detection," i*n the international conference on Field-Programmable Logic and its Applications (FPL),* 2002.
[14] PCRE - Perl Compatible Regular Expressions, www.pcre.org.
[15] MIT DARPA Intrusion Detection Data Sets, www.ll.mit.edu/IST/ideval/data/2000/2000_data_index.html