

Small Subset Queries and Bloom Filters Using Ternary Associative Memories, with Applications

Ashish Goel
Stanford University
Stanford, California
ashishg@stanford.edu

Pankaj Gupta
Twitter, Inc.
San Francisco, California
pankaj@twitter.com

ABSTRACT

Associative memories offer high levels of parallelism in matching a query against stored entries. We design and analyze an architecture which uses a *single* lookup into a Ternary Content Addressable Memory (TCAM) to solve the subset query problem for small sets, i.e., to check whether a given set (the query) contains (or alternately, is contained in) any one of a large collection of sets in a database. We use each TCAM entry as a small Ternary Bloom Filter (each ‘bit’ of which is one of $\{0,1,*\}$) to store one of the sets in the collection. Like Bloom filters, our architecture is susceptible to false positives. Since each TCAM entry is quite small, asymptotic analyses of Bloom filters do not directly apply. Surprisingly, we are able to show that the asymptotic false positive probability formula can be safely used if we penalize the small Bloom filter by taking away just one bit of storage and adding just half an extra set element before applying the formula. We believe that this analysis is independently interesting.

The subset query problem has applications in databases, network intrusion detection, packet classification in Internet routers, and Information Retrieval. We demonstrate our architecture on one illustrative streaming application – intrusion detection in network traffic. By shingling the strings in the database, we can perform a single subset query, and hence a single TCAM search, to skip many bytes in the stream. We evaluate our scheme on the open source CLAM anti-virus database, for *worst-case* as well as random streams. Our architecture appears to be at least one order of magnitude faster than previous approaches. Since the individual Bloom filters must fit in a single TCAM entry (currently 72 to 576 bits), our solution applies only when each set is of a small cardinality. However, this is sufficient for many typical applications. Also, recent algorithms for the subset-query problem use a small-set version as a subroutine.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$10.00.

1. INTRODUCTION

The subset (superset) query problem is defined as follows: Given a database consisting of N sets $D_i, i = 1...N$, each of which is a subset of a Universe U , determine if there is any set D_q that is a subset (superset) of a given query set $Q \subset U$. The superset query problem is also referred to as the containment query problem. Another closely related problem is that of partial matching, where each D_i has a set of attributes and Q specifies a subset (or superset) of those attributes. The problems of subset query, partial matching and superset query are equivalent as they can be reduced to one another [10]. Collectively, we refer to them as set query problems.

Set query problems are of fundamental importance in many applications. For example, database researchers have studied set query problems in order to facilitate efficient querying on set-valued attributes and in the context of data mining tasks [25, 24, 26, 9]. They also occur in many information retrieval problems – in one, we are required to find a set of documents containing the set of query words, and in another, we are required to find if a document is a subset of the set of documents in the corpus. The partial matching problem occurs in packet classification (based on header fields) in Internet routers where it is applied to security, flow recognition and other applications [19, 36].

In this paper we restrict ourselves to set queries in which all sets involved (Q and all D_i 's) are small¹. We design and analyze an architecture which uses a *single* lookup into a Ternary Content Addressable Memory (TCAM) to solve small set query problems. TCAMs are already in wide use in networking (for routing lookup and packet classification) and, as explained in the next section, are becoming increasingly practical as a primitive for general computing problems. We use each TCAM entry as a small Ternary Bloom Filter (each ‘bit’ is one of $\{0,1,*\}$) to store one of the sets in the collection. The “*” matches either a 0 or a 1, and makes TCAMs much more powerful than a binary CAM. Like Bloom filters, our architecture is susceptible to false positives. Since each TCAM entry is quite small (currently 72 to 576 bits), asymptotic analyses of Bloom filters do not directly apply. Surprisingly, we are able to show that the asymptotic false positive probability formula can be safely used if we penalize the small Bloom filter by taking away just one bit of storage and adding just half an extra element before applying the formula. We believe that this analysis

¹This is made more precise later.

of Ternary Bloom Filters is independently interesting. We also sketch an efficient algorithm for computing the false positive probability (fpp) exactly for small Bloom filters (in Appendix C). While the use of TCAMs has recently been proposed in many algorithmic settings, to the best of our knowledge, ours is the first work that employs TCAMs to implement randomized data structures.

Since each Bloom filter is stored in a TCAM entry, each set in the database must be small as well. For example, with a typical 288 bit wide TCAM, the corresponding Bloom filter can store close to 20 elements while having a false-positive probability of around 0.1%. This is quite adequate for many applications, including packet classification and intrusion detection. We further illustrate our architecture by applying it to the problem of high speed multiple string matching which involves finding occurrences of any of a set of strings in an incoming stream of bytes. This problem has diverse real world applications such as data de-duplication in storage systems, DNA/protein sequence alignment tools, and network intrusion detection. The first two applications discover commonalities between a new element and existing elements of a corpus. Solutions typically involve extracting strings from every document/sequence in the corpus and finding whether any of these is present in the input document/sequence.

Similarly, network intrusion detection systems look for the presence of any of a set of pre-determined signatures in high-speed streaming data. These signatures consist of strings or regular expressions; in the latter case, strings are typically extracted and matched in a first stage filter before doing regular expression matching. The key to achieving high performance in multiple string matching is to somehow lookup many traffic bytes at once. This is challenging, and perhaps the best-known algorithm for this problem is the Boyer-Moore algorithm [6] and its variants which search for small fragments of the pattern in the data stream. This algorithm can skip a small number of bytes when there is no match. In contrast, we shingle the strings in the database, and perform a single subset query using a single TCAM search to skip many bytes in the stream. We evaluate our scheme on the open source ClamAV database, for *worst-case* as well as as random traffic. Our results indicate that this architecture can plausibly scale to more than 15Gbps, which is more than one order of magnitude faster than previous work. In our evaluation, we formally define the notion of the “worst-case” stream that a malicious adversary can choose to maximize the false positive probability of our architecture, assuming the adversary knows the database of strings and all the parameters of our architecture, but not the random numbers used by our hash functions. We choose the parameters of our architecture so that we can simultaneously (a) Have a small false positive probability against this worst-case traffic stream, allowing a slower second stage algorithm (using standard data structures) to keep up with the desired speeds, and (b) Process many byte-positions in the traffic stream using a single TCAM lookup, giving the desired speed if we plug in standard TCAM performance numbers. We do not conduct experiments on actual traffic traces; however, it is important to note that real traffic traces can only yield fewer false positives than worst-case traffic.

In the experimental evaluation, we focus on network intrusion detection primarily for illustrative reasons, since that is an application with widely accepted benchmarks (in the form of anti-virus rules) and significant previous work for comparison purposes. Note that the subset query problem itself is applicable broadly within static and streaming databases (eg. for data mining, data de-duplication, and genomics/proteomics, as mentioned before), and is not restricted to networking applications.

Set query problems are very hard to solve efficiently using Random Access Memory. State-of-the-art algorithms such as [10] are still not practical for large sets and a large universe. Interestingly, this state-of-the-art algorithm uses a small set problem as a (one of many) sub-routine. It would be an important open problem to design fully general set query algorithms where the only inefficient steps are small set queries, for which our architecture can then be used.

Organization. Section 2 provides an overview of TCAMs and Bloom filters, and then describes related work in set queries and multiple string matching. Section 3 describes how a TCAM can be used to implement small set operations and small Bloom filters. Section 4 presents our analysis of small Bloom filters and extends that to set queries. Section 5 presents an application of this architecture to multiple-string-matching, along with experimental results. We end with brief concluding remarks. Some of the proofs in this paper are deferred to the Appendix, as is the dynamic programming approach to compute the false positive probabilities exactly for small Bloom filters.

2. BACKGROUND AND RELATED WORK

2.1 Overview of TCAMs

A CAM (Content Addressable Memory) is simply an associative lookup memory containing a set of fixed-width entries that can hold arbitrary data bits. A CAM takes a *search key* as a query, and returns the address of the entry that contains the key, if any. In this sense, it performs an inverse operation of that of a random access memory (RAM), which returns the data stored at a given address. The word Ternary in TCAM denotes that each data “bit” is capable of storing one of three states: 0, 1, or “*”, where a “*” is a wildcard that matches both 0 and 1. We will use the term “ternion ” to denote a 0, 1, or “*”. Some TCAMs allow “*” ’s in both the query and the stored data, whereas others restrict them to just the stored data. The presence of wildcards in TCAM entries implies that more than one entry could match a search key. When this happens, the index of the ‘highest’ matching entry (i.e., appearing at the lowest physical address) is typically returned.

TCAMs have now been used in switches and routers for almost a decade, primarily for the purposes of route lookup (more specifically, for longest prefix matching) [27, 20, 31] and packet classification [27, 20, 21]. Despite being a very powerful primitive, TCAMs have been mostly limited to use in high-performance routers, primarily because they have historically consumed very high power. However, in recent years, TCAMs have been optimized significantly in both performance and power usage. See Table 1 comparing the characteristics of the state-of-the-art TCAM and RAM technolo-

Memory	Single-chip Capacity	\$/chip	\$/MByte	Access speed (ns)	Watts/chip	Watts/MByte
DRAM	128MB	\$10-\$20	\$0.08-\$0.16	40-80	1-2	0.008-0.016
SRAM	9MB	\$50-\$70	\$5.5-\$7.8	3-5	1.5-3	0.17-0.33
TCAM	4.5MB	\$200-\$300	\$44.5-\$66.7	4-5	15-20	3.33-4.44

Table 1: Comparison of high-speed memory technologies in terms of capacity, speed and power.

gies. Measured on a per-bit basis, a TCAM is still about 8x more costly than an SRAM and has about 20x more power usage. We believe that there may be important applications where the benefits of the extra efficiency provided by TCAMs would outweigh the disadvantage of high power consumption; for example, in packet classification, TCAMs typically provide a 50x speedup so that the net power consumption is actually less than the much slower RAM based approach [33, 18, 35].

The state of the art TCAMs are available in densities of 512K entries, each entry being 72b wide (for a total of 36Mb). These densities have doubled approximately every 2 years in the last decade. The width of the entries can be typically configured to be 72, 144, 288, 576 bits with correspondingly proportionate reduction in the number of entries and search speed. Thus a 144b (and a 72b) ternary search can be made in modern TCAMs at the rate of more than 250 million searches per second (close to the highest performance SRAMs available today) while a 288b ternary search can be done at the rate of 125 million searches per second. To summarize, a TCAM should be thought of as yet another device available in the memory hierarchy with its own tradeoffs, much like DRAM and SRAM memory technologies. Several algorithms that use TCAMs as an underlying primitive have been proposed in the past few years including efficient packet classification [39], power efficient forwarding engines [41], supporting classifiers with range based searches [22, 21], sorting and searching [32] and fast data stream algorithms [3]. However, to the best of our knowledge, ours is the first work that employs TCAMs to implement a randomized data structure.

2.2 Overview of Bloom filters

A Bloom filter [5] is a randomized data structure that compactly represents a set using a bitmap. Each element of a set is hashed using k independent hash functions, each of which chooses a randomly chosen bit in the bitmap and sets it to 1. The resulting bitmap can then be queried for set membership queries by checking each of the k bit positions that the query element hashes to using the same k hash functions. This data structure can lead to false positives with the false positive probability depending on the size of the bitmap and the cardinality of the set, but can never lead to false negatives. Bloom filters have been applied in different applications such as routing table lookup and packet classification [12, 13], traffic measurement [16], approximate set reconciliation [8] in peer-to-peer networks, generating perfect hash functions [23] and so on. A survey of various such applications is presented in [7]. The asymptotic false positive probabilities of Bloom filters is now well understood; we extend this to small Bloom filters in this paper.

2.3 Related Work

Owing to their significance, set query problems have been well researched, both in theory and in the specific contexts of their applications. In this subsection, we look at previous work of both kinds.

Simple heuristics. Two trivial solutions to the set query problems are: (1) linear search and (2) pre-computing answers for all subsets of U . Clearly, both are infeasible for any large application.

Another approach to achieving small subset query problem could be to break up the query set into individual elements, solve the set-membership problem on all the query elements and then intersect these intermediate results to form the final result. Indeed, this approach is quite common in web search engines where the query set is typically very small (average size is anecdotally said to be around 3 or 4 words [4]). However, this approach needs as many set membership queries as the cardinality of the query set, and is infeasible when the intermediate results are very large. Even if the intermediate results are small, we will illustrate how in order to achieve the same speed in a generic application as our proposed architecture, this approach would require a large amount of parallelism and would consume no less storage.

Formal bounds. Rivest [28, 29] proposed the first non-trivial solutions to the set query problems. He showed that when $|U| \leq 2 \log N$, the trivial space bound of $2^{|U|}$ can be somewhat improved. In a significant improvement, Charikar et al [10] proposed two algorithms with the following tradeoffs (here $u = |U|$): (a) $N2^{O(u \log^2 u \sqrt{c/\log N})}$ space and $O(N/2^c)$ time, for any c , and (b) Nu^c space and $O(uN/c)$ query time, for any $c \leq N$. For the kinds of applications we are interested in, $|U|$ is quite large compared to N . For example, in our shingling architecture of network intrusion detection, U is the set of all strings of a certain size, say 6 bytes leading to $u = 2^{48}$, while $N \approx 2^{17}$. Even if we take $u = O(N)$, the formulae are prohibitively expensive to be practical for high speed applications. As mentioned earlier, this algorithm uses a small-set problem as a subroutine.

Set queries in databases. In this context, one commonly used approach has been to employ a bitmap called a ‘signature’ that is similar to a Bloom filter but typically does not explore the use of multiple hash functions [9, 26, 25]. Also, without access to a TCAM, one has to resort to linear search of all the bitmaps, which is clearly prohibitively expensive. An extension of this approach is to build an index in order to avoid linear search on the bitmap [24], but it is difficult to prove any bounds on performance. In contrast,

we explore the use of small Bloom filters with multiple hash functions as they can be used in a TCAM and analyze the false positive probabilities in both theoretical and practical settings.

Multiple string matching. The Multiple String Matching (MSM) problem has been recently studied in the context of network intrusion detection, where high performance is paramount because every byte of incoming network traffic needs to be looked up against a large set of signatures. The largest bottleneck in achieving high performance in MSM is the number of memory accesses required for every byte of the incoming traffic. As an example, 1 byte arrives every 0.8ns at 10Gbps network speed, and based on the memory access speeds shown in Table 1, one can not support this speed with just one memory bank. Further, it can be easily seen that the extent of replication needed, even with the state-of-the-art algorithms, for many tens of thousands of strings is impractical. The most common MSM algorithm is by Aho and Corasick [1]. The basic algorithm does two memory accesses per byte in the worst case and also builds a large data structure so that fast SRAM can not be used in practical settings, and one has to resort to slower DRAM to store the data structure. As DRAM access times are no better than 40ns, one instance of the Aho-Corasick algorithm runs at $1/2 * 8/40 = 0.1$ Gbps. The extent of replication needed to achieve 10Gbps is therefore 100, which seems prohibitively complex and expensive. Tuck et al [37] compress the data structure size needed for Snort signatures, and are able to bring it down to approximately 1MByte for 1500 strings. However, for 95,000 ClamAV signatures, even with their compressed algorithm, one would need more than 60MBytes of SRAM and still be only able to achieve 1.33Gb/s (with SRAM access time of 3ns). Again, to achieve a speed of 10Gbps using replication, we would need close to 450MBytes of SRAM memory which is impractical as it would involve 50 SRAM chips (see Table 1) consuming too much space, power and money.

In [17], setwise algorithms for pattern matching were used in Snort to improve its performance. This was improved upon in [2] which proposed the E²XB algorithm that works by creating a front end filter to the above well known algorithms. Essentially, substrings of the original strings are first searched in the network traffic, and when a substring is located, a backend algorithm such as Aho-Corasick [1] is employed. The idea is that if a string s contains at least one character that is not in an input string I (consisting of the bytes of a packet), then s is not a sub-string of I . The E²XB algorithm[2] does two passes on the input-string I , first to record which characters appear in I and to then sequentially iterate over each string of the database to verify if all the characters of the string do appear in the record. This approach has two problems: (1) Two passes are needed – often not possible in a high speed streaming environment, (2) The amount of time spent is linear with the number of strings in the database. Studies of the E²XB algorithm have reported up to 200Mbps of throughput in software using various traces and 1661 strings. We use the similar idea of using a two stage architecture in this paper, where the first stage is a filter that can quickly skip past many bytes in the traffic stream reducing the work for the second stage

by two orders of magnitude.

TCAMs have also been previously applied to the problem of pattern matching in intrusion detection by [40] and [12]. In [40], the patterns are mapped on the TCAM device, and the traffic is searched against the TCAM to find matches. On 1768 string patterns of average length 55 bytes, [40] achieved 2Gbps scan rate using 240 KB of TCAM. On Snort v2.1.2, which had 1836 string patterns, using 295KB of TCAM, [40] was able to achieve 1Gbps. In [12], a FPGA based implementation that uses multiple parallel Bloom filters to perform string matching. They support 10,000 strings at 2.4 Gbps using this approach. Again, the number of strings supported is an order of magnitude lower than ClamAV type of signatures.

Previous work done directly on ClamAV signatures has achieved speeds of around 50Mbps using modified Boyer-Moore algorithm [42] and 400Mbps using 512KB memory for standard Bloom filters [15]. The latter scheme could perhaps scale to higher performance if done in dedicated hardware — extrapolating from that paper’s Figure 4 (which seems to reflect an increase in performance by less than 100Mb/s for each doubling of Bloom filter size), the performance could reach 800Mb/s with 8MB of on-chip memory (which is very expensive, but conceivable). However, it would still be more than one order of magnitude less than our goal of 10Gbps network intrusion detection and anti-virus scanning.

3. SET QUERIES AND BLOOM FILTERS USING TCAMS

3.1 Definitions

The subset query problem is defined as follows: Given a database D consisting of N sets $D_i, i = 1 \dots N$, each of which is a subset of a Universe U , determine if there is any database set that is a subset of a given query set Q , where $Q \subset U$. We analogously define the superset query problem, sometimes called the containment query problem [10]

Our approach is to represent each D_i as a *separate* Ternary Bloom filter (TBF) F_i and store F_i in the i^{th} TCAM entry. A TBF is a vector of ternions where a ternion is a “bit” that can have a value of “*” in addition to 0 or 1. We also refer to “*” as a “wildcard”.

Given two ternions x and y , we will say they match each other if they are both equal, or if at least one of them is a “*” ; we will denote this as $x =_T y$. Observe that this definition holds if either x or y are constrained to be only binary (i.e., not having a wildcard value). We will extend the notion of ternary match to ternion vectors (or equivalently, ternion strings) p and q by saying that $p =_T q$ if they have the same dimension and for every position $i, p_i =_T q_i$.

We also define the total order \prec over the set $\{0, *, 1\}$ as $0 \prec * \prec 1$ (i.e., 0 is smaller than * which is smaller than 1). We will use \max/\min of a multi-set of values from $\{0, *, 1\}$ to denote the element which is the largest/smallest under the order \prec .

We will assume that we are given a hash function $f : U \rightarrow \{0, 1\}^M$ which maps U (the universe of elements) into an M -bit binary vector. This hash function could be viewed as the function that maps an element to the Bloom filter vector resulting from adding this element to an initially empty

Bloom filter 0^M . We will assume that $f(x)$ is equivalent to picking k bits uniformly at random (with replacement) out of the M bits, and setting them to 1 while the rest of the M bits are set to 0. The parameter k refers to the number of hash functions used in a standard Bloom filter. We will further assume that f is perfectly unbiased, in that $f(x)$ is independent of all other $f(y)$'s. We will refer to the i -th bit of $f(x)$ as $f_i(x)$.

We will define two other functions g and h which map the universe of elements into $\{0, *, 1\}^M$. Let $g_i(x)$ and $h_i(x)$ denote the i -th ternions of g, h respectively. We define $g_i(x) = 0$ if $f_i(x) = 0$ and $g_i(x) = *$ otherwise. We will set $h_i(x) = *$ if $f_i(x) = 0$ and $h_i(x) = 1$ otherwise. Hence, g results in a $\{0, *\}^M$ vector while h results in a $\{1, *\}^M$ vector.

3.2 Small set operations using a TCAM

We now show how a TCAM can be used to implement a bank of N Bloom filters, each of size M bits, where M is small enough to fit into a single TCAM entry. Recall from Section 2 that typical values for N and M in today's technology are 128,000 and 288.

Given a set of elements S , we define multiple ways of mapping them into a TCAM entry:

1. $f_Q(S)$ is an M bit vector, with the i -th bit set to the maximum of $\{f_i(x) : x \in S\}$. In other words, $f_Q(S)$ is the resulting standard Bloom filter after adding all elements of S .
2. $f_{Sub}(S)$ is an M ternion vector, with the i -th bit set to the maximum of $\{g_i(x) : x \in S\}$. In other words, the i^{th} bit is set to $*$ if $f_i(x) = 1$ for any x and 0 otherwise. Note that $f_{Sub}(S)$ is a $\{0, *\}^M$ vector, i.e., it does not have '1's.
3. $f_{Sup}(S)$ is an M ternion vector, with the i -th bit set to the minimum of $\{h_i(x) : x \in S\}$. In other words, the i^{th} bit is set to 1 if $f_i(x) = 1$ for any x , and $*$ otherwise. Note that $f_{Sup}(S)$ is a $\{1, *\}^M$ vector, i.e., it does not have '0's.

Informally, f_Q is the standard method of hashing a set into a Bloom filter, f_{Sub} turns all the "1"s in f_Q into " $*$ ", and f_{Sup} turns all the "0"s into " $*$ ".

Recall that we are given a database D of N "small" sets D_1, \dots, D_N and a "small" query set Q . Here by a small set, we mean a set which can be hashed into a Bloom filter of size M and give reasonably small false positive probabilities. As we will see later, the exact bound on the size of the set depends on the desired false positive probabilities. Given a query set Q , we produce a bit vector, $f_Q(Q)$.

Subset Query: During the preprocessing phase, we store $f_{Sup}(D_i)$ in the i -th entry of the TCAM. Then, issuing the query, $f_Q(Q)$ to the TCAM results in a match if the set Q is a superset of one or more of the sets D_1, D_2, \dots, D_N . Further, the index of one of the matched sets is returned (If we store sets D_i in decreasing order of their sizes, the index returned will be that of a largest subset of Q).

Superset Query: During the preprocessing phase, we store $f_{Sub}(D_i)$ in the i -th entry of the TCAM. Then, issuing the query, $f_Q(Q)$ to the TCAM results in a

match if the set Q is a subset of one or more of the sets D_1, D_2, \dots, D_N . Further, the index of one of the matched sets is returned (If we store sets D_i in increasing order of their sizes, the index returned will be that of a smallest superset of Q).

We defer the analysis to the next section, but note several points about these queries:

1. There can not be false-negatives, but false-positives are possible.
2. By letting each D_i be a singleton set, we can implement an *inverted Bloom filter*, where the query corresponds to a (small) Bloom filter, and all the N elements (i.e., singleton sets) in the TCAM are tested for membership against the query Bloom filter, in a single TCAM lookup operation.
3. If some of the D_i 's are encoded using f_{Sub} and some others using f_{Sup} , then we can simultaneously test whether Q is a subset of any of the D_i 's encoded using f_{Sub} or the superset of any of the D_i 's encoded using f_{Sup} .
4. Some TCAMs also allow the query to contain " $*$ "s. If the D_i 's are encoded using f_Q then we can simulate a superset or a subset query by encoding the query string Q using f_{Sub}, f_{Sup} respectively; this may be advantageous in increasing the capacity of TCAMs since we would not need TCAM entries to have wildcards.
5. The following straightforward technique can be used for subset and superset queries when the universe U is very small, e.g., when $|U| \leq M$: Represent any subset, S of U as a bitmap whose bit i is set if S contains the i^{th} element. It follows that the superset or subset queries can be solved exactly using the f_{Sub}, f_{Sup} encodings as above.

4. ANALYSIS

We will now bound the false positive probability of our Bloom filter structure. It is not clear a priori that the asymptotic Bloom filter formula for calculating false positives is a reasonable estimate for Bloom filters small enough to fit into a single TCAM entry.

4.1 Analysis of small Bloom filters: Half an extra element and one fewer bit

We have been unable to obtain a simple closed form formula for the false positive probability of small Bloom filters, and will establish an interesting upper bound instead.

Let $f_S(M, E, k, q)$ denote the false positive probability for a q -way set query (i.e., $|Q| = q$) to a Bloom filter with M bits, E elements stored in the Bloom filter, k hash functions used per element, *assuming that none of the elements in the query set are stored in the Bloom filter*. The case $q = 1$ gives the false positive probability for the standard set membership queries with Bloom filters.

Let $f_A(M, E, k, q)$ be the false positive probability that results from using the asymptotic Bloom filter formula, i.e.

$(1 - e^{-Ek/M})^{kq}$. The core of the upper bound is the following technical lemma, which also provides general insight for other situations where small Bloom filters may be used.

LEMMA 4.1. *The false positive probability, $f_S(M, E, k, q)$ is at most*

$$\prod_{j=1}^{kq} \left(1 - \left(1 - \frac{1}{M} \right)^{Ek+j-1} \right).$$

PROOF. The proof is via a balls-and-bins argument. Consider a Bloom filter with M total bits, to which E elements have been hashed, each element choosing k random bits. We will think of each bit position as a bin and each choice of position as a ball being thrown into a bin at random. Then, constructing the Bloom filter is equivalent to throwing Ek “red” balls uniformly at random into M initially empty bins. We will refer to those bins which get at least one red ball as “red bins”; these bins correspond to those bits which are set to 1 in the Bloom filter, or equivalently, to a “*” in the corresponding TCAM entry if we are using f_{Sub} . Constructing the query is equivalent to then throwing kq “green” balls uniformly at random into the same M bins. A false positive is obtained if each green ball lands into a red bin. This is a complicated process, since the event that the first green ball falls into a red bin and the event that the second green ball falls into a red bin are not independent; the first event makes it more likely that there are more red bins than initially expected, which increases the likelihood of the second event being true. If the correlation were in the other direction, we could still have pretended that the events are independent to obtain a valid upper bound on the fpp. The rest of the proof exhibits another process which has this anti-correlation and can also be coupled with the process we are interested in.

Define variables $Z_j, j = 1, \dots, kq$ and set $Z_j = 1$ if the j -th green ball falls into a red bin and $Z_j = 0$ otherwise. As mentioned above, the false positive probability is merely the probability that all the Z_j 's are 1. We also define variables $X_j, j = 1, \dots, kq$ and set $X_j = 1$ if the j -th green ball falls into a non-empty bin (i.e., in a bin which already had at least one red or green ball), and $X_j = 0$ otherwise. The event $Z_j = 1$ implies $X_j = 1$, but the reverse is not necessarily true. However, the variables X and Z exhibit an interesting coupling. If all the X_j 's are 1, then there can not be a non-empty bin with only green balls since the first of these must have fallen into an empty bin which would contradict all X_j 's being 1. Hence, all the Z_j 's must be 1 as well. Thus, the false positive probability is the same as the probability of all the X_j 's being 1, and using Bayes' rule, we get:

$$f_S(M, E, k, q) = \prod_{j=1}^{kq} \Pr[X_j = 1 \mid X_r = 1 \text{ for } r = 1, \dots, j-1]. \quad (1)$$

Consider the probability that $X_j = 1$ conditioned on the event that $X_r = 1$ for all $r < j$. This is the probability that the j -th green ball falls into a non-empty bin, conditioned on the fact that all the previous green balls also fell into a non-empty bin. Intuition suggests that the probability of $X_j = 1$ should only go up if we remove the conditioning, since the $j-1$ previously thrown green balls would be allowed to fall in empty bins as well, increasing the number of non-empty

bins and hence the probability of X_j being 1. This intuition is indeed true, and in Appendix A we formally prove the following “anti-correlation” inequality:

$$\Pr[X_j = 1 \mid X_r = 1 \text{ for } r = 1, \dots, j-1] \leq \Pr[X_j = 1]. \quad (2)$$

Now observe that $X_j = 0$ if all the previous $Ek + j - 1$ balls fall into bins other than the one chosen for the j -th green ball. Hence,

$$\Pr[X_j = 1] = 1 - (1 - 1/M)^{Ek+j-1}. \quad (3)$$

The proof of the lemma follows by combining equations 1, 2, and 3. \square

By replacing each term in the product with its largest value which is achieved at $j = kq$, it is immediate that $f_S(M, E, k, q)$ is at most $\left(1 - (1 - \frac{1}{M})^{(Ek+kq-1)} \right)^{kq}$. In fact, the following stronger statement is also true:

COROLLARY 4.2. *The false positive probability $f_S(M, E, k, q)$ is at most*

$$\left(1 - \left(1 - \frac{1}{M} \right)^{Ek+(qk-1)/2} \right)^{kq}.$$

We defer the proof of this corollary to Appendix B and focus on the implications instead. There are two sources of space overhead compared to the asymptotic formula of $(1 - e^{-Ek/M})^{kq}$. First, we can not assume that $(1 - 1/M)^M$ is $1/e$. This issue is easy to deal with: for $M \geq 1$, $(1 - 1/(M+1))^M > 1/e$; this corresponds merely to *having one fewer bit in the Bloom filter*. Second, we have an exponent of $(Ek + (qk - 1)/2)/M$ instead of Ek/M , which corresponds to having at most $q/2$ extra elements in the Bloom filter. For the common case $q = 1$, this corresponds to having half an extra element.

We are now ready to state our main theorem:

THEOREM 4.3. $f_S(M, E, k, q) < f_A(M - 1, E + q/2, k, q)$.

4.2 Analysis for small set operations

The finite Bloom filter analysis above tells us that we do not lose much in terms of false positive probability by encoding a small Bloom filter into a TCAM entry. However, it is important to remember that we are now simultaneously performing N Bloom-filter-like operations, and that the probability that the TBF of Q matches the TBF of a set D_i depends on the number of elements in $Q - D_i$ as only the elements in $Q - D_i$ need to collide with the other elements in the TBF of D_i . Consider the case where we are storing N sets in a TCAM with entries of width M , each D_i is of the same size d , and Q is disjoint from all the D_i 's. Then, by theorem 4.3, $f_S(M, |D_i|, k, q)$ is at most $f_A(M - 1, d + q/2, k, q)$. If we choose k to minimize $f_A(M - 1, d + q/2, k, q)$, i.e., $k = ((M - 1)/(d + q/2)) \ln 2$, then the formula for overall false-positive probability is:

$$\sum_{i=1}^N f_S(M, |D_i|, k, q) < N \cdot \left((1/2)^{\frac{\ln 2 * (M-1)}{d+q/2}} \right)^q \quad (4)$$

Based on this formula, Table 2 tells us the maximum size of the sets D_i for three false positive probabilities and typical

q	FPP < 1%	FPP < 0.1%	FPP < 0.01%
1	6	5	5
2	14	12	11
4	30	26	23
8	62	54	47
16	125	109	96

Table 2: Maximum sizes of sets D_i for given values of q and false positive probabilities. Here $N = 2^{17}$, $M = 288$.

TCAM parameters of $N = 2^{17}$, $M = 288$ for various values of $q = |Q|$.

The above bounds can easily be adapted to the case where $|Q - D_i| = j$ simply by substituting j for q , i.e., by using the expression $f_S(M, |D_i|, k, j)$. However, given a value of j , it is also easy to compute the optimum k and the corresponding false positive probabilities using a dynamic program. This is the approach taken for the intrusion detection application in the next section. The basic ideas behind the dynamic program are sketched in Appendix C. In figure 1, we compare the false positive probabilities resulting from the asymptotic upper bound in theorem 4.3, the dynamic program, and the asymptotic lower bound $f_A(M, E, k, q)$. It is clear that all these fpps are very close to each other, specially the dynamic program and the lower bound, empirically illustrating what we already proved formally: that we do not lose much by using small Bloom filters. The fpps are plotted in log-scale; the difference between them is even less discernible in linear scale.

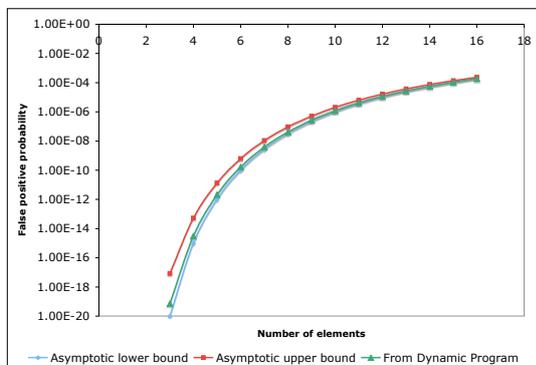


Figure 1: Comparison of the false positive probabilities resulting from the asymptotic upper bound in theorem 4.3, the dynamic program, and the asymptotic lower bound $f_A(M, E, k, q)$. Each of these three fpps is computed using the k that is optimum for that quantity, q is set to 1, M to 288, and E varies from 3 to 16. The y -axis is in logscale.

Of course assuming that all the D_i 's are disjoint from Q is a very strong assumption. As we illustrate in the next

section, even when $|Q - D_i|$ is 1, this architecture results in small false positive probabilities for realistic applications.

5. MULTIPLE STRING MATCHING

In this section, we show how to use Ternary Bloom filters to solve the problem of high performance multiple string matching (MSM). In this problem, we are given a database, D , of N strings of varying lengths. The MSM problem is to report all occurrences of *any* of these strings in an incoming “traffic” stream of bytes.

In this experimental section, we focus on network intrusion detection primarily for illustrative reasons, since this application has widely accepted benchmarks in the form of open-source anti-virus rules and significant previous work for comparison purposes. The MSM problem in particular and the subset query problem in general are applicable broadly within static and streaming databases (eg. for data mining, data de-duplication, and genomics/proteomics, as mentioned before), and are not restricted to networking applications.

We evaluate our MSM architecture on the signatures of an open-source intrusion detection/anti-virus system called Clam AntiVirus or ClamAV [11] which contains more than 95,000 signatures² at the time of writing this paper. We show how, using TCAMs and commodity DRAMs (low cost memories used in PCs), we can support higher than 10Gbps of network bandwidth while scanning for ClamAV signatures in every byte of network traffic. This is more than one order-of-magnitude performance/capacity improvement over state-of-the-art solutions. In our evaluation, we formally define the notion of a “worst-case” traffic pattern, and choose the parameters of our architecture so that we can simultaneously (a) Have a small false positive probability against worst-case traffic, allowing a second stage standard data structure to keep up with the desired speeds, and (b) Process many byte-positions in the traffic stream using a single TCAM lookup, giving the desired speed if we plug in standard TCAM performance numbers.

5.1 High performance MSM architecture using Ternary Bloom Filters and Shingling

The key to achieving high speed is to process more than one byte in the input traffic stream, T , at once. Since a string in the database D can be anywhere in T , the problem is inherently byte-aligned, and indeed most previous solutions work on each byte in T . In our architecture, we process multiple bytes using Ternary Bloom filters. The basic idea is to quickly determine whether there is a *possibility* of any string in the database being present in a sliding window on the bytes in the traffic stream, T . Assume for a moment that every string in the database is of length l . Then, if it has been determined that no string can be inside a window T_1, \dots, T_w ; we can safely slide the window forward by $w - l + 1$ bytes and next consider the window $T_{w-l+1}, \dots, T_{2w-l}$. In this way, we achieve a speed of $w - l + 1$ bytes per unit time. In our architecture, we pick fixed size substrings of each string in the database and use subset

²The total number of signatures exceeds 300,000 but the content-based signatures, i.e., those that need to be searched for in the contents of the file, number 95,988 in Nov 2008.

query operations using Ternary Bloom filters to determine whether a substring is present in the query window.

More specifically, our MSM architecture consists of two levels of data structures. The first level can be viewed as a *filter* that works at the speed of traffic and reduces the work needed to be done by the second level by quickly determining if a query window can potentially match one of the strings, and for the second level to then complete the actual match. The second level is a standard algorithmic implementation of an MSM algorithm, typically the Aho-Corasick MSM algorithm [1]. Network intrusion detection is one of those applications where we do not expect too many matches, as we expect rare intrusions rather than frequent intrusions. Hence, we need to only worry about the false-positive rate and assume that true positives are rare³. In such a scenario, our MSM filter can filter out more than 99% of the traffic, allowing the second level to work up to 100x slower than the first level. The MSM filter uses ternary Bloom filters stored in TCAM. The filter produces either a ‘No’ or a ‘May be’ for every window of bytes. If the answer is a ‘May be’, the second level data structure needs to further process this window of bytes. In this section we look at the operation of the first level MSM filter in more detail.

We preprocess all strings of the database so that we store S bytes of each string in the first level MSM filter. In other words, we first form the set of S -byte unique substrings (such as prefixes) from the set of strings. Then, for each such substring, $P = (P_1, P_2, \dots, P_S)$, we form p shingles consisting of consecutive bytes (sometimes also called n -grams) from P starting from P_1, P_2, \dots, P_p . Hence, each shingle is a substring of P of size $S - p + 1$ bytes. These p shingles are then stored in a Ternary Bloom filter of size M which is stored in one TCAM entry (recall that M ‘bits’ is the width of each TCAM entry). The Bloom filter is constructed using the f_{sup} function defined in Section 3.2. Hence, the number of TCAM entries needed is the same as the set of unique S -byte picked substrings of the strings in the database. The query Bloom filter, Q , is constructed from the traffic shingles in a similar manner using f_Q . If a TCAM lookup done on Q returns “No”, we can skip multiple bytes in the query at the same time. If we desire to skip W bytes, we need to ensure that Q has $n = W + p - 1$ shingles so that the database substring that could potentially begin from the last byte of the W byte window is not missed.

We now analyze the false positive probabilities of using our architecture. Note that we are checking whether any of N sets, each of size $|D_i| = p$, is a subset of the set of query shingles of size $n = W + p - 1$.

We can write the false positive probability bound as

$$\text{FPP} = \sum_{j=1}^p N_j * f_S(M, n, k, j) \quad (5)$$

where for any j , N_j = the number of strings in the database that have j different shingles than those in the query (as only

³This is a general feature of Network Intrusion Detection rather than our architecture and is typically ensured by isolating flows that trigger many true matches. In any case, as we explain later in 5.3.2, our first stage filter is more selective than the standard approach of using the Boyer-Moore algorithm as a first stage filter.

the different shingles contribute to the false positives). Remember that we are checking whether the set stored in the TCAM entry is a subset of the query, and hence, the role of the query and the set are reversed in the Bloom Filter formula. Given estimates of N_j (either in the worst case or for random traffic, as explained later), we tune the Bloom filter design parameters (p , W , and k) such that the overall false positive probability is no more than 1%, implying that the second level data structure needs to work at approximately 1% of the traffic speed on average. The resulting maximum W for this false positive probability value of 1% then gives us the speed of our architecture to be W bytes per TCAM lookup. We will use the dynamic program outlined in Appendix C to tune the Bloom filter parameters.

5.2 Evaluation on ClamAV

The signatures of the open-source antivirus system called ClamAV [11] number 95,000 at the time of writing this paper and have been increasing at more than 10,000 signatures per year. We show how our architecture achieves a speed of 15Gbps (under worst-case traffic conditions and reasonable assumptions about the other components of the system) and more than 20Gbps under average traffic conditions. First, we choose $M = 288$ so that we can fit up to $2^{17} = 128K$ strings in a single 36Mb TCAM (TCAMs of this capacity have been commercially available for at least the last couple of years) and can thus fit the current ClamAV database in a single TCAM with room for further growth. For these values of $N = 95000$ and M , we tune the design parameters to have the false positive probability of the ternary Bloom filter to be less than 1%, so that the second level data structure needs to run at a speed of less than 0.2Gbps. This speed can be achieved by the well-known Aho-Corasick MSM algorithm [1] running with commodity DRAM memory. In this algorithm, one has to make two memory accesses per byte in the worst-case. At DRAM bank access speed of $t_{RC} = 40ns$ (see Table 1), the standard implementation of Aho-Corasick algorithm achieves a speed of 0.1Gbps with one bank. Therefore two DRAM banks can easily support the desired speed. We therefore focus on details of the first level MSM filter in the remainder of this section.

We have written a program that uses the approach outlined in Appendix C to compute the optimal Bloom filter design parameters p and k for the ClamAV database and $M = 288$. We first extract all substrings of length at least 10 bytes (as in [42]) and for a given p , we use a simple diversity heuristic in which we pick substrings greedily such that the total frequency of top shingle counts is minimized. Given these counts and assumptions on traffic, we estimate the values of N_j (see below for details) needed in equation 5. We then compute the optimal value of k that minimizes the false positive probability for the given value of p and W . We then pick that p which maximizes W such that the false positive probability is still less than 1%. The values of W thus obtained depend on the values of N_j which in turn depend on assumptions on traffic conditions. These are summarized in Table 3 and explained below. Note that this is all part of the pre-processing phase.

5.2.1 Worst-case traffic

Worst-case traffic is one where every query causes the

Traffic type	FPP < 1.0%			FPP < 0.1%		
	W_{max}	p_{opt}	k_{opt}	W_{max}	p_{opt}	k_{opt}
Worst-case	15	3	11	12	2	13
Random	45	6	4	39	6	4
Random (parameters as in worst-case)	22	3	11	13	2	13
Worst-case (9,10)	27	6	5	21	6	5

Table 3: Maximum window sizes W_{max} (i.e., the number of bytes skipped per TCAM lookup) and optimal number of shingles of a string (p) and number of hash functions (k) for false positive probabilities of 1% and 0.1%.

highest false positive probability. We will provide bounds on this worst-case false positive probability, by estimating an upper bound on N_j for every j and plugging that into equation 5. We upper bound N_j using the following heuristic: we first compute the frequency distribution of shingles in the database strings, so that m_i is the number of database strings that contain the i^{th} most popular shingle. Then, we upper bound N_j as $\frac{1}{p-j} \sum_{i=1}^{i=n} m_i$ since for j shingles to be different in each of N_j strings, each such string should have $p-j$ shingles in common with the query, leading to a total of $N_j(p-j)$ shingle counts which should obviously be less than the sum of the counts of the topmost n occurring shingles. Recall that n is the number of query shingles and $n = W + p - 1$.

Doing this for the ClamAV database gives us the maximum value of W to be 15 (with $p = 3, k = 11$ and therefore shingle-size=8 for $S = 10$) in order to achieve a false positive probability of less than 1%. This implies that we can skip 15 bytes in every 288b TCAM lookup under worst-case conditions of traffic (i.e., such that the traffic is constructed so as to maximize N_j). At a speed of 125 million lookups/sec for 288b lookups, this can support network bandwidth of 15Gbps. The maximum window size for 0.1% false positive probability is 12, corresponding to 12Gbps network bandwidth.

As mentioned earlier, we do not conduct experiments with real TCAMs on actual traffic traces; however, it is important to note that real traffic traces can only yield fewer false positives than worst-case traffic.

5.2.2 Random traffic

For random traffic, we show that the probability that a window of traffic has even one shingle in common with that of any of the picked database substrings is negligible for a reasonably large shingle size. For example, for 10 byte strings, $p = 6$ implies $10 - 6 + 1 = 5$ byte shingles. Therefore, this probability is $N \cdot p \cdot 2^{-8 \cdot 5} < 2^{17} \cdot 6 \cdot 2^{-40} < 2^{-20}$. Therefore, we assume $N_j = 0 \forall j \neq p$ and $N_p = N$ for our calculations. With the Bloom filter parameters tuned for these values of N_j , we get a window size of 45 for 1% false positive probability. Even if we use the Bloom filter design parameters optimized as above for worst-case traffic (i.e., $p = 3, k = 11$, shingle size = 8), we get maximum window sizes of 22 and 13 respectively for false positive probabilities of 1% and 0.1% respectively.

5.2.3 Tradeoff between filtering precision and system throughput

As mentioned above, the output of our MSM filter is fed

to a second level standard string matching data structure. Depending on the expected traffic conditions and desired system design, it may be worthwhile to relax the precision of the MSM filter if it might lead to overall higher throughput. Hence, we explore the (l, u) version of the MSM problem defined as follows: The filter should output a “Yes” (called a “Maybe” from the point of view of the entire string matching subsystem) if the traffic stream has a matching substring of size u or greater, “No” if the traffic stream has no l or larger byte matching substring, and is allowed to output anything if there is a matching substring of size at least l but none of size u or larger. This is generally considered safe because the second level string data structure will anyway verify the match. For example, the $(S-1, S)$ problem allows the MSM filter to report either a “Yes” or a “No” when an $S-1$ byte (but not S byte) matching substring occurs in a traffic window. This relaxation in the responsibility of the MSM filter implies $N_j = 0$ for $1 \leq j \leq u-l$ which drastically reduces the false positive probability as lower values of j comprise a large portion of the false positive probability in equation 5 (higher values of j have a smaller false positive probability contribution because j occurs in the exponent in the expression for the false positive probability for a given k).

For example, for worst-case traffic conditions, we get a traffic window size of 27 bytes (implying 27 Gbps bandwidth) for the (9, 10) problem.

5.3 Comparison with related multi-byte skipping work

In this subsection, we compare our subset query approach to MSM with two alternative approaches that can also skip multiple bytes of the traffic stream.

5.3.1 Exact-match with replication

An alternative implementation of the set query problem in the context of MSM would be to do exact match of W shingles of the query window using simple hashing. To get the same speed as that of our ternary MSM architecture, this replicated exact-match solution will need a parallelism of W , which would be quite complex and costly for large W and N . In each of the W instances, this scheme would hash the query shingle and the shingles of each string into, say, h bits. This solution has false positive probability equal to $N2^{-h}W$ and it consumes Wh bits per string. For a false positive probability of 1% with $N = 95000$ and $W = 15$, we get $h \approx 27$ and $Wh = 405$ bits per element. In contrast, our architecture needs no parallelism and has no storage penalty

(infact it uses fewer – 288 bits – per element). Of course, it can be argued that we use the parallelism *inside* a TCAM, but that is an off-the-shelf commodity device.

5.3.2 Boyer-Moore family of algorithms

The classic Boyer-Moore algorithm [6] works on a single string and can advance the traffic stream by multiple bytes. This has been extended to multiple patterns by Wu and Manber [38]. A variation is also used in current ClamAV source code. The general idea is as follows: As a preprocessing step, hash each of the first t byte shingles of every database string, say up to u bytes of the string (assuming that every string is of size at least u) and thus construct a hash table consisting of the first $u - t + 1$ shingles of each string. At query time, consider a window containing u bytes of the traffic stream, and lookup the t -byte tail of this window in this hash table. If the hash table does not return a match, $u - t + 1$ bytes can be safely skipped. In other words, the Boyer-Moore filter returns a ‘yes’ if one of the shingles of a database string matches the tail shingle in the traffic window. This is clearly a much less selective filter than our ternary Bloom filter, as the latter says a ‘yes’ only when all shingles of the database string are present in the traffic window. Further, the ternary architecture decouples the window size W (number of bytes that can be skipped) from the length of the string u , while they are tightly coupled in the Boyer-Moore filter. For example, with $N = 95000$, one would use at least a 3-byte hash and therefore probably at least a 4-byte shingle, implying $t \geq 4$. Therefore, if the minimum size of the strings in the database is $u = 10$, the Boyer-Moore filter can advance no more than $u - t + 1 = 10 - 4 + 1 = 7$ bytes per hash table lookup, and no more than 3 bytes per hash table lookup if $u = 6$. In contrast, the ternary MSM architecture can advance by 20 and 23 bytes respectively for false positive probabilities of 1% and 0.1% for *worst-case* traffic in the case of $u = 6$, and by 45 and 53 bytes respectively for $u = 10$ (as the Boyer-Moore lookup is equivalent to the (t, u) problem defined in Section 5.2.3). For a full discussion of the Boyer-Moore family of multi-string algorithms, please refer to [30] where the maximum speed of any of these algorithms is reported to be no more than approximately 100 Mbits per second on random traffic.

6. CONCLUSIONS

Small set queries have many applications but no efficient software based methods are known for these queries. Ternary Content Addressable Memories (TCAMs) allow wildcard searches which makes them a very powerful algorithmic primitive. We showed how each entry of a TCAM can be used to implement a small Bloom filter, and hence, small subset/superset queries against a database of sets can be answered in one memory cycle, assuming the database fits into the TCAM. We analyzed the false positive probability of our scheme by establishing a connection between small Bloom filters and asymptotic Bloom filters. This is the first example of a complex randomized data structure implemented using a TCAM. It remains an interesting open problem to design (or prove the impossibility of) a general subset query algorithm where the only inefficient parts are small set queries, which can then be rendered efficient using our architecture.

We demonstrated an application of our architecture to the problem of Multiple String Matching (MSM). Previous work in MSM for intrusion detection [17, 39, 12, 2, 13, 37] has focused on the open-source Snort [34] database that consists of the order of 5000 signatures (these are primarily protocol anomaly signatures, and do not include anti-virus signatures). The solutions previously proposed typically compress the data structure so that it fits entirely in on-chip memory (if a hardware solution) or in the cache (if a software solution). Modern intrusion detection systems need to scan for the presence of viruses in network traffic similar to an anti-virus software running on a host machine. However, as we have seen with ClamAV, the number of virus signatures is more than an order of magnitude higher, and is increasing rapidly. Existing solutions can not fully fit a compressed data structure for such a large number of signatures on-chip or in processor cache. Recently proposed approaches to speeding up the ClamAV scanning process (on a host) achieve a throughput of no more than 400 Megabits per second (and could conceivably go up to 1Gbps in dedicated hardware). In contrast, aggregate network traffic could easily be 10Gbps or higher in modern enterprise networks. Therefore, network administrators have to make do today with using only a subset of the entire signature database, most often of only those viruses considered newly discovered. Hopefully, the proposed Ternary Bloom filter architecture for MSM using commonly available TCAM devices will provide one viable solution for this problem.

In the experimental evaluation, we focused on network intrusion detection primarily for illustrative reasons, since that is an application with widely accepted benchmarks (in the form of anti-virus rules) and significant previous work for comparison purposes. The subset query problem itself is applicable broadly within static and streaming databases and is not restricted to networking applications.

Acknowledgments

This research was supported by NSF grant 0915040. We would like to thank Srinivasan Venkatachary for helpful discussions.

7. REFERENCES

- [1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18:333–340, 1975.
- [2] K. G. Anagnostakis, S. Antonatos, E. P. Markatos, and M. Polychronakis. E2xB: A domain-specific string matching algorithm for intrusion detection. In *Proceedings of the 18th IFIP International Information Security Conference (SEC2003)*, 2003.
- [3] N. Bandi, D. Agrawal, A. Abbadi, and A. Metwally. Fast data stream algorithms using associative memories. In *Proc. SIGMOD*, 2007.
- [4] Blog entry about query sizes. <http://www.beussery.com/blog/index.php/2008/02/google-average-number-of-words-per-query-have-increased/>.
- [5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

- [6] R. Boyer and J. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762, October 1977.
- [7] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, pages 636–646, 2002.
- [8] J. Byers and M. Mitzenmacher. Fast approximate reconciliation of set differences. *Draft paper, available as BU Computer Science TR 2002-019*, 2002.
- [9] C. Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *L. M. Haas and A. Tiwary, editors, Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 355–366, 1998.
- [10] M. Charikar, P. Indyk, and R. Panigrahy. New algorithms for subset query, partial match, orthogonal range searching, and related problems. *Lecture Notes In Computer Science; Vol. 2380. Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 451–462, 2002.
- [11] ClamAV. <http://www.clamav.net/>.
- [12] S. Dharmapurikar, M. Attig, and J. Lockwood. Design and implementation of a string matching system for network intrusion detection using FPGA-based bloom filters. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, 2004.
- [13] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. In *Proc. Hot Interconnects, Stanford, CA*, pages 44–51, August 2003.
- [14] D. Dubhashi and D. Ranjan. Balls and bins: a study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998.
- [15] O. Erdogan and P. Cao. Hash-AV: fast virus signature scanning by cache-resident filters. *International Journal of Security and Networks*, 2:50–59, 2007.
- [16] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *Proceedings of the 2001 ACM SIGCOMM Internet Measurement Workshop*, pages 75–80, November 2001.
- [17] M. Fisk and G. Varghese. Fast content based packet handling for intrusion detection. *Tech. report CS2001-0670, Univ. of California, San Diego*, 2001.
- [18] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects VII*, pages 34–41, 1999.
- [19] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network (Special Issue)*, 15(2):24–32, 2001.
- [20] Integrated Device Technology Inc. <http://www.idt.com/>.
- [21] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with Ternary CAMs. In *Proc. Sigcomm*, pages 193–204. ACM, 2005.
- [22] H. Liu. Efficient mapping of range classifier into Ternary-CAM. In *Proc. of Hot Interconnects*, 2002.
- [23] Y. Lu and B. Prabhakar. Perfect hashing for network applications. In *IEEE Symposium on Information Theory*, pages 2774–2778, 2006.
- [24] C. Masson, C. Robardet, and J. Boulicaut. Optimizing subset queries: a step towards sql-based inductive databases for itemsets. In *Proceedings of the 2004 ACM symposium on Applied computing*, 2004.
- [25] T. Morzy and R. Nanopoulos. Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes. In *Proc. ADBIS*, pages 236–252. Springer-Verlag, 2003.
- [26] T. Morzy and M. Zakrewicz. Group bitmap index: a structure for association rules retrieval. In *Proc. ACM SIGKDD*, pages 284–288, 1998.
- [27] Netlogic microsystems. <http://www.netlogicmicro.com/>.
- [28] R. L. Rivest. Analysis of associative retrieval algorithms. *Ph.D. thesis, Stanford University*, 1974.
- [29] R. L. Rivest. Partial match retrieval algorithms. *Siam Journal on Computing*, 5:19–50, 1976.
- [30] L. Salmela, J. Tarhio, and J. Kytöjoki. Multi-pattern string matching with q-grams. *ACM Journal of Experimental Algorithmics*, 11, 2006.
- [31] D. Shah and P. Gupta. Fast updates on Ternary-CAMs for packet lookups and classification. In *Proc. Hot Interconnects VIII, Stanford*, 2000.
- [32] S. Sharma and R. Panigrahy. Reducing TCAM power consumption and increasing throughput. In *Proceedings of the 10th Symposium on High Performance Interconnects HOT Interconnects (HotI'02)*, page 107, 2002.
- [33] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proc. of ACM Sigcomm*, pages 213–224, 2003.
- [34] Snort. <http://www.snort.org/>.
- [35] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *Proc. of ACM Sigcomm*, pages 135–146, 1999.
- [36] D. E. Taylor. Survey and taxonomy of packet classification techniques. In *ACM Computing Surveys*, 2004.
- [37] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proc. IEEE Infocom, Hong Kong*, pages 333–340, 2004.
- [38] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. *Technical Report, Department of Computer Science, University of Arizona - TR-94-17*, 1994.
- [39] F. Yu and R. H. Katz. Efficient multi-match packet classification with TCAM. In *Proc. of Hot Interconnects*, 2004.
- [40] F. Yu, R. H. Katz, and T. Lakshman. Gigabit rate multiple-pattern matching with TCAM. Sahara Retreat Posters Winter 2004.
- [41] F. Zane, G. Narlikar, and A. Basu. CoolCAM: Power-Efficient TCAMs for Forwarding Engines. In *Proc. of IEEE Infocom*, 2003.
- [42] X. Zhou, B. Xu, Y. Qi, and J. Li. MRSI: A fast pattern matching algorithm for anti-virus applications. In *ICN '08: Proceedings of the Seventh International*

APPENDIX

A. PROOF OF THE ANTI-CORRELATION INEQUALITY

Recall that we have M bits in the finite Bloom filter, which corresponds to having M bins. Consider the experiment where we first throw Nk red balls into M bins uniformly at random (with replacement) and then throw q green balls, also uniformly at random (with replacement). Recall that we defined $X_j = 1$ if the j -th green ball fell into a previously occupied bin and $X_j = 0$ otherwise. Let S_{j-1} denote the set of bins occupied by the first $j - 1$ green balls and let b_j denote the bin that the j -th green ball falls into. Our goal is to prove equation 2:

$$\Pr[X_j = 1 \mid X_r = 1 \text{ for } r = 1, \dots, j - 1] \leq \Pr[X_j = 1].$$

The conditioning in the above equation is equivalent to saying that all bins in S_{j-1} must have a red ball. We consider two cases:

1. $b_j \in S_{j-1}$: In this case the bin where the j -th ball falls has a green ball already, and $X_j = 1$ regardless of the conditioning.
2. $b_j \notin S_{j-1}$: In this case, $X_j = 1$ iff there is a red ball in bin b_j . Bin occupancies are not independent, which makes it non-trivial to analyze the probability that $X_j = 1$ conditioned on the event that all bins in S_{j-1} have at least one red bin. Fortunately, bin occupancies are known to satisfy negative dependency [14] which implies that

$$\begin{aligned} \Pr[\text{Bin } b_j \text{ has a red ball} \mid \text{All bins in } S_{j-1} \text{ have} \\ \text{at least one red ball}] \\ \leq \Pr[\text{Bin } b_j \text{ has a red ball}]. \end{aligned}$$

In both cases, $\Pr[X_j = 1 \mid X_r = 1 \text{ for } r = 1, \dots, j - 1] \leq \Pr[X_j = 1]$. \square

B. PROOF OF COROLLARY

We will now prove corollary 4.2 from section 4.1:

The false positive probability $f_S(M, E, k, q)$ is at most

$$\left(1 - \left(1 - \frac{1}{M}\right)^{Ek + (qk-1)/2}\right)^{kq}.$$

PROOF. Consider the function $h(x) = 1 - (1 - 1/M)^{Ek+x}$. From lemma 4.1, we know that $f_S(M, E, k, q) \leq \prod_{j=1}^{kq} h(j-1)$. Taking logs on both sides, we get $\log f_S(M, E, k, q) \leq \sum_{j=1}^{kq} \log h(j-1)$.

The function h is easily seen to be concave, and hence also log-concave, which implies

$$\sum_{j=1}^{kq} \log h(j-1) \leq kq \log h\left(\frac{1}{kq} \cdot \sum_{j=1}^{kq} (j-1)\right).$$

The argument of h in the RHS above is just the arithmetic mean of the series $0, 1, \dots, kq - 1$. Hence, we have

$$\sum_{j=1}^{kq} \log h(j-1) \leq kq \log h\left(\frac{kq-1}{2}\right).$$

Exponentiating both sides, we get

$$\prod_{j=1}^{kq} h(j-1) \leq \left(h\left(\frac{kq-1}{2}\right)\right)^{kq}$$

which completes our proof. \square

C. THE DYNAMIC PROGRAMMING APPROACH FOR COMPUTING THE OPTIMUM VALUE OF K AND THE FPP

We will assume that d elements are being hashed into a Ternary Bloom Filter of size M , and that the query has q elements. We will assume that each element chooses k bit positions. We will further assume that the query set is disjoint from the set being hashed into the Bloom filter. Thus, we are interested in computing $f_S(M, d, k, q)$. Recall that if only j elements of the query are distinct from the set in the Bloom filter, then we can pretend that $q = j$. We compute $f_S(M, d, k, q)$ in three steps:

1. For every $0 \leq s \leq M$, and every $0 \leq b \leq dk$, define $\text{SPAN}(M, b, s)$ to be the probability that when b balls are thrown into M bins, exactly s bins are non-empty, i.e. the b balls “span” s bins. These can be computed using the dynamic program

$$\begin{aligned} \text{SPAN}(M, b, s) &= \text{SPAN}(M, b-1, s-1) \cdot \left(\frac{M+1-s}{M}\right) \\ &\quad + \text{SPAN}(M, b-1, s) \cdot \left(\frac{s}{M}\right). \end{aligned}$$

The first term in the RHS corresponds to the first $b-1$ balls spanning $s-1$ bins and the b -th element falling into an empty bin, and the second term corresponds to the first $b-1$ elements spanning s bins and the b -th element falling into a non-empty bin. The boundary conditions for this program are easy: $\text{SPAN}(M, 0, 0) = 1$; $\text{SPAN}(M, 0, s) = 0$ for $s > 0$; $\text{SPAN}(M, b, 0) = 0$ for $b > 0$.

2. For every $1 \leq s \leq M$, and every b' , define $\text{COVER}(M, s, b')$ to be the probability that if b' balls are thrown into M bins, all of them fall into the first s . This is merely $(s/M)^{b'}$.

3. Now, $f_S(M, d, k, q) = \sum_{s=1}^M \text{SPAN}(M, dk, s) * \text{COVER}(M, s, qk)$.

Given a method of computing $f_S(M, d, k, q)$, we can iterate over all values of k to find the best k . Observe that:

1. Given that $1 \leq k \leq M$ and M is small (say 288), this is not an onerous step.
2. This needs to be done only once at the beginning before the database is stored in the TCAM.
3. This idea extends to the case where we are given N_j corresponding to different values of j as in equation 5.