SigMatch: Fast and Scalable Multi-Pattern Matching

Ramakrishnan Kandhan Nikhil Teletia Jignesh M. Patel
Computer Sciences Department, University of Wisconsin–Madison
{ramak, teletia, jignesh}@cs.wisc.edu

ABSTRACT

Multi-pattern matching involves matching a data item against a large database of "signature" patterns. Existing algorithms for multi-pattern matching do not scale well as the size of the signature database increases. In this paper, we present sigMatch – a fast, versatile, and scalable technique for multi-pattern signature matching. At its heart, sigMatch organizes the signature database into a (processor) cache-efficient q-gram index structure, called the sigTree. The sigTree groups patterns based on common sub-patterns, such that signatures that don't match can be quickly eliminated from the matching process. The sigTree also uses parallel Bloom filters and a technique to reduce imbalances across groups, for improved performance. Using extensive empirical evaluation across three diverse domains, we show that sigMatch often outperforms existing methods by an order of magnitude or more.

1. INTRODUCTION

The problem of multi-pattern matching is defined as follows: Given a set of patterns, $P = \{p_1, p_2, ..., p_n\}$, where each p_i is a regular expression pattern over an alphabet $\sum$, find all occurrences of these patterns in a data item, D, over the same alphabet.

The term "pattern" in the definition above, is often also called "signature" in the literature, and in this paper we use these two terms interchangeably to refer to a regular expression. This problem of multi-pattern matching has a number of practical applications, including Information Extraction (IE), anti-virus scanners (AVSs) and Intrusion Detection Systems (IDSs).

First consider the use of multi-pattern matching in an IE system. An IE system often needs to match crawled web pages against a set of patterns. For example, in DBLife [13], mentions for the entity "University of Wisconsin" is coded as the following regular expression: $((University|Univ.|Univ)\backslash s + of|U|U.)\backslash s + Wisconsin(\backslash s + (at|in|, | - | - -))?\backslash s + Madison$. This regular expression allows matching this entity with different ways of referring to this university, including "Univ Of Wisconsin at Madison" and "U. Wisconsin, Madison".

We note that the problem considered in this paper is different from *approximate multi-string matching* [10, 17, 18], which is also used in IE systems. In approximate multi-string matching, each entity is matched to a string (e.g. "University of Wisconsin" in the example above), and some string similarity measure is used to capture the different ways in which this string could be referred to in the text documents. A detailed study of whether multi-pattern matching or approximate string matching (or some combination) is more effective for entity resolution in IE applications, is an interesting research topic and beyond the scope of this paper. Nevertheless, there are examples of IE systems, such as DBLife, that rely heavily on regular expression matching – i.e., they require multi-pattern matching techniques that we consider in this paper.

Other applications of multi-pattern matching include anti-virus scanners (AVSs) and Intrusion Detection Systems (IDSs), which require matching a set of signatures of known viruses/threats against some streaming data to check for presence of malware.

We note that in multi-pattern matching, the signatures are usually available beforehand while the data/text is streamed. This is in contrast to the problem of searching text [11] where the documents/text are available beforehand for indexing and the regular expression signatures are streamed. Prior techniques suggested in literature [15, 20, 28] build an index structure on the signatures and use it to scan through a data item in linear time (in terms of the length of the data item). However, these techniques do not scale as the size of the signature database increases. To make matters worse, in many applications, the signature databases are growing rapidly. For example, the number of signatures in AVSs and IDSs have nearly doubled over the last few years [2, 4]; IE systems are becoming more complex with larger signature databases [10]; the signature databases for applications like spam detection and content filtering are also growing rapidly [5]. This problem has been amplified by the rapid increase in disk space usage in the case of AVSs, network bandwidth in the case of IDSs, and web content in the case of IE systems, which increases the size of the underlying data leading to more frequent invocations of these multi-pattern matching systems.

Consequently, there is a compelling need for a multi-pattern matching method that is a) *fast*, b) *scalable* with increasing signature database sizes, and c) *generic* so that it can work in any domain that requires multi-pattern matching. In this paper we present a technique, called sigMatch, that addresses this need.

The sigMatch method is a generic filtering technique that can be plugged in as a pre-processing step for any existing multi-pattern matching system. Figure 1 provides a high-level overview of the sigMatch system. On the left of this figure is the data that needs to be matched, which is split into a number of *data items*. The description of the data item depends on the application. For example, in the case of an IE system a data item can be a web page, whereas for an IDS a data item can be a network packet, and for an AVS each

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. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

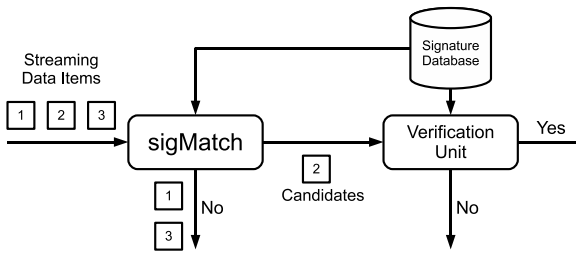


Figure 1: High-level overview of the system

data item can be a memory buffer or a file. Data items accepted by sigMatch, referred to as *candidates*, are then sent to a verification unit in the actual end application to perform a final (and definitive) check for the presence of patterns. Note that the sigMatch module does not send any other information to the verification unit regarding the patterns that may or may not be present in the data. This design ensures that sigMatch can be used as a portable "filter" across a wide variety of existing pattern matching systems. This generic approach is desirable, as multi-pattern matching applications have varying characteristics. For example IDSs often have few short signatures, while AVS databases often have a large number of long signatures. In addition, while IDSs use the full capabilities of regular expressions, AVSs and IE systems often restrict themselves primarily to the use of wildcard characters like * and ?. The sigMatch technique can be used as a pre-processing filter step with all these applications. Furthermore, sigMatch is designed to be a conservative filtering technique. So it can produce some false positive matches, but it never misses a match (i.e., has no false negatives). Thus, sigMatch is a "safe" filtering technique.

For speed and scalability, sigMatch uses an index, called the sigTree, to index a conveniently chosen substring (q-gram) from each signature. Consequently, if no substring indexed in the sigTree is present in a data item, then that data item can be quickly discarded as it is guaranteed to not have any content that matches the signature database (this situation is called the "no-match" case). Previous q-gram filter structures suggested in literature either use tries [20] or Bloom filters [15]. While trie-based structures are faster, Bloom filters (which are essentially bitmaps) scale better with increasing number of signatures. As far as we know, sigTree is the first structure that combines the benefits of both approaches.

The sigMatch technique is also designed to exploit processor caches effectively. The size of previous index structures (such as those used in [15, 20, 21]) increases rapidly as the size of the signature database increases. These structures quickly become larger than the processor (L2) caches, and accessing these structures often requires at least one random memory access *per byte* of the scanned data item. Given the increasing gap between processor speeds and memory access latencies [26], these algorithms are unable to leverage the full capabilities of current processors. Since in many multi-pattern matching applications the "no-match" cases are common, sigMatch uses a largely L2 cache-resident q-gram index structure to quickly discard these no-match cases. Thus, it dramatically reduces the number of main memory accesses, resulting in improved performance. In addition, the q-gram index structure used in sigTree is designed to eliminate a significant fraction of such no-match cases using only L1 cache references (which typically costs only one cycle to access, compared to a few tens of cycles for access to the L2 cache, and few hundreds of cycles for access to the main memory).

In order to test sigMatch, we integrated it with two of the largest (from the perspective of signature database sizes) public state-of-the-art systems: namely, ClamAV – a popular open-source AVS, and Bro – a network IDS. We also tested sigMatch on a real IE dataset used in DBLife [13], which uses the Perl regular expression engine for multi-pattern matching. We show that sigMatch improves the performance of ClamAV by 10-12X, Bro by 4X and DBLife by 15X. To test the scalability of our method for future (larger) signature databases we mimic the ClamAV signature database (which has about 90K regular expression signatures), and produce synthetic signature databases that have up to 300,000 signatures. For this large signature database, sigMatch improves the performance of ClamAV by a factor of nearly 28.

The remainder of this paper is organized as follows. In Section 2, we discuss related work. We provide an overview of the system in Section 3. Section 4 contains our experimental evaluation, and Section 5 contains our conclusions and directions for future work.

2. RELATED WORK

The multi-pattern matching problem has been well studied under both the approximate (e.g. [10, 16]) and exact match criterion (e.g [7, 9, 14, 20]) using both hardware and software methods.

The hardware approaches (e.g. [14, 23]) generate multiple specialized circuits for regular expression matching automaton to search the streaming data in parallel. However, such approaches are limited and are designed primarily for IDSs where the matching occurs only in dedicated systems like mail gateways. For general multi-pattern applications that are usually run on conventional machines, such hardware solutions are generally not feasible.

Software methods for exact multi-pattern matching are either shift table or automaton based. The shift table-based methods (e.g. [9, 12, 21, 25, 28]) use multiple pre-computed tables to determine the next viable location where a pattern can occur. Automaton-based methods (e.g. [7, 19, 22]) employ either a DFA or an NFA-based representation for regular expressions. While the NFA representation is compact in terms of storage, they are generally slower than the DFA representation. However, there is a state space explosion in the DFA representation as the size of the signature database increases. Recent approaches [19,22,27] have concentrated on reducing the memory requirements of DFAs using grouping and rewriting techniques. However, the performance of these systems degrades when the number of signatures increase beyond a few hundred. All of these approaches are complementary to our approach, as they concentrate on improving the efficiency of the verification unit (see Figure 1), and thus can be combined with our approach to further improve the overall performance of the system.

The filtering approaches for pattern matching can be broadly divided into two categories – prefix and q-gram based approaches.

Prefix-based filters build a set of strings $P = \{p_1, p_2, .., p_n\}$ such that each signature/pattern in the database has a prefix in P. The strings $p_1, p_2, ..., p_n$ could be of the same [15] or different [20] lengths. During scanning, if the pattern at a particular location in the data item is not in P (determined by either exact [20] or approximate [15] string matching), then no pattern can be found at this location and the scanner moves on to the next viable location, possibly with the help of a shift table [28]. If the pattern does exist in P, then some additional computation is performed to determine whether this particular location contains a pattern in the database. Most of the previous approaches suggested in literature for multi-pattern matching (e.g. [2,15,20]) have relied on prefix-based filters.

Filters that use q-grams follow an approach similar to the prefix-based filters but use substrings rather then prefixes. The advantage of q-gram filters is that the cardinality of the set P is significantly smaller when compared to prefix-based filters, so they provide a better filter rate, and hence potentially improved performance. The

disadvantage of using q-grams is that the entire data item has to be scanned for the presence of patterns in the case of a match.

Our work is different from these previous efforts as it focuses on building a generic cache-efficient q-gram based filter for multi-pattern matching that can handle a wide spectrum of patterns, and can easily be combined with any existing multi-pattern matching application. Also to the best of our knowledge, our effort is the first one to demonstrate a system that can scales to handle very large pattern/signature databases.

3. SIGMATCH

In this section we describe sigTree (the q-gram index that we use in sigMatch), and its construction method. But before we begin with the sigTree description, we briefly present background information about Bloom filters, which we use in the sigTree.

3.1 Bloom Filters: Background

A Bloom filter [8] is a bit array of m bits that is used to check if an element belongs to a set. Initially all bits in this array are set to zero. When an element is added to the Bloom filter, a set of k predefined hash functions are applied to the element to obtain k values. The bit positions corresponding to these values are then set to 1 in the array. To ascertain whether an element belongs to a set, the same set of k hash functions are applied to the element to obtain k values. If at least one of the bit positions corresponding to these values are not set in the bit array, then the element does not belong to the set. If they are all set to 1, then the element *may* belong to the set. Due to collisions, the Bloom filter can provide false positives, but it never results in a false negative.

Bloom filters have two main advantages. First, they require less space compared to other data structures used for encoding sets, such as tries [20] and hash tables [10]. Second, the time complexity for adding and searching an element is constant independent of the number of elements inserted into the Bloom filter. Of course, as we "index" more elements in a Bloom filter the probability of collision and the number of false positives increases. But this problem can be mitigated by increasing the number of bits in the Bloom filter. Starobinski et al. [24] present a detailed analysis for finding an appropriate Bloom filter size for a desired collision rate.

3.2 Overview of the sigTree Structure

In any q-gram based filtering technique, a substring from *each* signature in the pattern database is indexed in a trie [20], or a Bloom filter [15]. Tries are generally faster than Bloom filters as in the latter at least one hash computation has to be performed at each viable location in the data item (and there could be collisions in the Bloom filter, which can lead to more false positives, and poor performance). However, the memory requirement of tries is generally greater than that of a Bloom filter. The sigTree combines the benefits of both these approaches.

An additional issue with tries is choosing the length of the substring from each signature to index. If we pick short substrings, then the index will not be very effective as a filter (i.e. it will result in many false positives). On the other hand, if we pick substrings that are very long, then the index size will be large. When the index is large, it is unlikely to fit into the processor caches. Consequently, node accesses in the trie will be very expensive as they will require fetching data from main memory, which is often slower by an order of magnitude or more, over fetching data from the processor caches. The sigTree addresses these tradeoffs by intelligently picking discriminative short substrings, part of which is indexed in a trie and the remaining part is indexed using a Bloom filter.

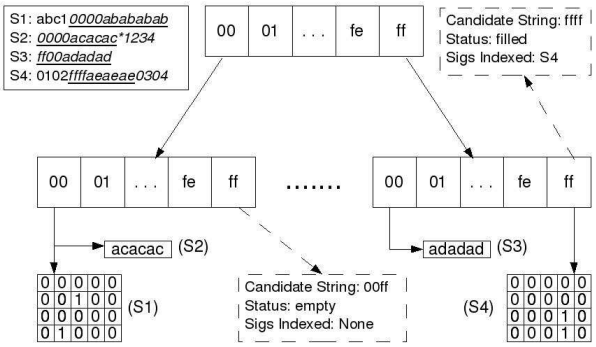


Figure 2: An example sigTree with b=2 and β=4 bytes.

At a high-level, the sigTree is essentially a truncated trie with a Bloom filter/bitmap at the leaf level of the trie.

Figure 2 shows an example sigTree with four indexed signatures. In this figure, each byte in the pattern is represented as a two digit hexadecimal number, and the substring of the signature that is indexed in the sigTree is underlined. Each sigTree node is a lookup array of size 256, one for each possible byte value. (This size can be reduced if the alphabet size is smaller, and vice versa.) A non-leaf node has 256 pointers to child nodes. A leaf node could point to a linked list or a Bloom filter, or both. Some or all of these leaf pointers can be nulls. Each leaf node corresponds to a unique string of length b (the height of the tree). This unique string is called the *candidate string* of that leaf node.

A sigTree is constructed from the signature database by indexing exactly one substring (without any wildcard characters) of length $b + \beta$ from each signature. Here b is the height of the index and β is the number of bytes indexed in the Bloom filter. The first b bytes of the substring map the signature to a node, and the next β bytes following the b bytes in the substring are used to hash into the Bloom filter at that node using a "set" of hash functions. The linked lists are used for *short signatures* that do not have $b+\beta$ consecutive bytes without any wildcard characters. The sigTree construction algorithm is described in detail below in Section 3.4. Next we discuss how the sigTree is used to detect hits/matches in the data items (see Figure 1), and factors that influence the performance of the sigTree.

3.3 Matching with sigTree

Each byte of each data item that is streamed through the sigMatch filter (see Figure 1) is run through the sigTree. The sigTree detects a potential match in two cases. A match is referred to as a *true positive* if a pattern in the signature database exists in the data item and sigTree detects it as a match. On the other hand, if a pattern in the signature database doesn't exist in the data item, the match is a *false positive*. There are two types of false positives. *Type A* false positives occur when the substring of a pattern indexed in the sigTree exists in the data, but the actual full pattern does not. *Type B* false positives occur when the substring indexed in the sigTree doesn't exist in the data, but a potential match is detected due to collisions in the Bloom filter.

Since calls to the verification unit (see Figure 1) are expensive, for improved performance we want to reduce the number of false positives. Type A false positives can be reduced by increasing the length of the q-gram (q= $b + \beta$) indexed in the sigTree. However, increasing b increases the cost of an index node traversal (as the index is larger and accesses to the index node has a higher chance of leading to a cache miss), while increasing β increases the cost

of computing the hash functions. A better approach to reduce the type A false positives is to choose substrings from the pattern that are less frequent in the data. We discuss techniques for reducing type A false positives in Section 3.5.

Type B false positives are dependent on the collision rate in the Bloom filter, which in turn is dependent on the hash functions used, the size of each Bloom filter, and the length of the q-grams indexed in the sigTree.

To better characterize design choices that can reduce the number of false positives when using a sigTree, consider the following performance model for the sigTree, which is similar in spirit to the performance modeling used in [15]: For simplicity, assume that there are no short signatures in the pattern database and that all leaf nodes are at the same height. Let p_t be the the fraction of non-empty leaf nodes and p_h be the probability that the hash functions return a false positive. Assume that the computational costs of the index tree traversal, hash computation and a call to the verification unit are c_t, c_h and C respectively. Then, the cost of the multi-pattern matching algorithm is:

$$c_t + p_t * [c_h + p_h * C] \quad (1)$$

To maximize performance, we want (a) the sigTree to be largely cache-resident and (b) the number of calls to the verification unit to be minimal. Note that the cost of the verification unit C is orders of magnitude larger than c_h and c_t, as it often involves at least one random memory access per scanned byte (and memory access is orders of magnitude more expensive than a cache access). The cost of tree traversal c_t is lower than c_h as the sigTree is designed so that the nodes can generally fit in the L1 cache, while the Bloom filters generally reside in the L2 cache. So, $c_t < c_h << C$. Hence, we need to keep p_t and p_h small for improved performance.

The value of p_t depends on b, the height of the tree and the substrings indexed in the sigTree. Increasing the height of the tree reduces p_t, but it also increases the cost of the tree traversal c_t. Also the height of the tree is restricted by the the shortest signature in the pattern database. A better approach to reduce p_t is to pick q-grams from signatures that minimize the total number of non-empty leaf nodes. We discuss an approach to do this in Section 3.4.

The value of p_h depends on three factors – the number and computational complexity of the hash functions used, the size of the Bloom filter, and the number of patterns allocated to each node. One way to reduce p_h is to increase the number and complexity of the hash functions used, but this technique also increases the value of c_h. In our experiments (Appendix C.2), we found that using a two-level hashing technique with cheap hash methods offers a good balance. Consequently, in the remainder of this paper we use a two-level hash function (*xor+shift* and *RShash*) as suggested by Erdogan et. al [15]

Increasing the size of the Bloom filter directly reduces p_h, but also increases the memory usage. We present a detailed evaluation of the effect of Bloom filter size in Section 4.2.

Overloading a leaf node with a large cluster of signatures can quickly increase the probability that a false negative is generated. Hence for better performance we want the signatures to be evenly distributed amongst the non-empty leaf nodes – i.e. the tree must be (nearly) balanced. We discuss techniques for uniform allocation of signatures in Section 3.4.2.

3.4 sigTree Construction

In this section, we describe the method for creating a sigTree given the set of signatures Sig, the height of the tree b, and the number of bytes to be hashed in the Bloom filter β. Before we present the method for constructing a sigTree, we define a few terms that are used in the discussion below.

Definition 1. A string str is a *Representative Substring* (RS) of a signature sig in a sigTree T with parameters b, β if it satisfies the following conditions.

1. The string str is a substring of the signature sig.
2. The string str has a length b and contains no wildcard characters.
3. The string str has at least β bytes following the b bytes without any wildcard characters in sig
4. If sig has no substring satisfying the third condition, then the first b bytes of the longest substring without wildcard characters in sig is its only representative substring. Such signatures are called *short signatures*.

Definition 2. The *frequency* of a string str, w.r.t. a signature set Sig, is defined as the number of signatures in Sig for which str is a RS.

Definition 3. A set of strings $S = \{s_1, s_2, ..., s_n\}$, each of length b, is a *b-gram cover* of signature set Sig, if every signature in Sig has at least one RS in S.

Definition 4. Let γ be some positive integer. A set of strings $S = \{s_1, s_2, ..., s_n\}$, each of length b, is defined to be a *γb-gram cover* of signature set Sig, if every signature in Sig has at least γ or all of its RSs (if it doesn't have γ RSs) in S.

The sigTree index construction involves a preprocessing step to prepare the signature database for indexing, and the actual index construction step. Each of these steps are described below.

Preprocessing: To construct a sigTree on a signature database, we require that each signature in the database have at least b consecutive bytes without any wildcard characters. However, this condition may not be satisfied by certain signatures. For such signatures, we use simple rewrite rules to convert them into equivalent signatures that have at least b consecutive character bytes. For example the signature $[A|a]b[C|c]$ is equivalent to the four signatures Abc, AbC, abC and abc. We use similar rewrite rules in our implementation to ensure that we can find at least one substring in each pattern to index in the sigTree.

Index Construction: The sigTree index is constructed using four steps.

The first step is to construct a b-gram cover S for the signature set. The cardinality of the b-gram cover must be small (to minimize p_t, the fraction of non-empty leaf nodes). A greedy algorithm for constructing an effective b-gram cover is discussed in Section 3.4.1.

Next, an empty sigTree structure is created using the b-gram cover S, where each leaf node with a candidate string in S is assigned an empty linked list and a Bloom filter, while all other leaf nodes point to null.

The third step is to allocate signatures amongst these nodes. These signatures must be equally distributed among the non-empty nodes for better performance. We discuss an approach for effective signature allocation in Section 3.4.2.

The final step is to fill the Bloom filter and the linked list at each non-empty leaf node with the strings from the signature database allocated to that leaf node. We describe our approach for this task in Section 3.4.3.

The pseudo code for the sigTree construction algorithm can be found in Appendix A. A detailed example of sigTree construction can be found in Appendix B.

3.4.1 b-gram Cover

The goal of this step is to find a b-gram cover for the signature set Sig, whose cardinality is minimal over all possible cover sets.

Since a brute force approach for finding a minimal b-gram cover involves searching an exponential number of possibilities, we suggest a greedy approach. The intuition behind the algorithm is as follows: Since most real datasets are neither random nor uniform, the distribution of the frequencies of b-grams is expected to be skewed. We exploit this feature to build an effective b-gram cover. The expectation here is that choosing b-grams with the highest frequencies will lead to a more effective (i.e. smaller) cover set.

The algorithm works as follows: First, the b-gram that has the highest frequency for the signature set Sig is added to S, the b-gram cover. All signatures that have this string as a RS are then removed from the signature set. The b-gram with the second highest frequency is picked next, and this process is repeated. If the addition of a b-gram to S doesn't result in any signature being removed from Sig, then it is removed from S as all the signatures for which this b-gram is a RS have already been accounted for by the other strings in S. This process continues until all signatures have at least one RS in S.

Next, we refine this set by removing any "redundant" strings. A b-gram str in a b-gram cover S of a signature set Sig is said to be *redundant* if $S - \{str\}$ is also a b-gram cover of Sig. The simplest method to remove redundant substrings is to remove one string at a time from S and test whether the remaining strings form a b-gram cover. However, the order in which the strings are tested determines the number of strings that can be removed. But, we know that the b-grams added towards the end of the last iteration to S are RSs of signatures that have few RSs, and are hence more likely to be present in the minimal b-gram cover. So now we loop through the set S in reverse, starting with the last added string. If we find that a b-gram in S is not a RS of at least one signature in Sig that has no other RS in S, then that string is removed from the set S. Empirically, we have found that this refinement step decreases the cardinality of the b-gram cover by as much as 20%. The pseudo code for this algorithm is shown in Appendix A.1.

3.4.2 Signature Allocation

In this step, we assign each signature in the signature set Sig to exactly one leaf node in the sigTree. A leaf node in the sigTree is referred to as a *candidate node* of a signature sig if the candidate string of the node str is a RS of sig, and str is in the b-gram cover S. The simplest approach for signature allocation is to pick a node from the set of candidate nodes at random for each signature in the database. However, for improved performance, the number of signatures at the non-empty leaf nodes must be (nearly) balanced in the sigTree, so that none of the Bloom filters are overloaded.

To achieve this goal, we follow two simple rules. First, a signature is assigned to a node only after all signatures with fewer candidate nodes have been allocated. Second, given a set of candidate nodes for a signature, we always pick the node with the least number of signatures allocated to it so far. Once a signature is allocated to a node N, the next β bytes following the candidate string in the signature is stored in an array V_N corresponding to the node N. Since short signatures do not have β bytes following the candidate string without any wildcard characters, the length of the string stored in V_N is less than β bytes long for these short signatures. The pseudo code for this method is shown in Appendix A.1.1.

3.4.3 FillNode

The goal of this step is to populate the Bloom filter and linked list at each sigTree leaf node N with the strings from the array V_N corresponding to that node. If the allocation of signatures is uniform, this task is simple as each string in the array V_N is inserted into the Bloom filter if it is β bytes long, or into the linked list if its length is less than β.

However, if the allocation is uneven, this task is not trivial. As stated earlier (Sections 3.2 and 3.3) overloading a node with a large cluster of signatures increases the number of false positives generated, which adversely impacts performance. One way to address this issue is to use a b-gram cover that facilitates uniform allocation. To do this, we can construct a b-gram cover S for the signature set Sig such that each signature in Sig has at least γ RSs in S, where γ is a positive integer greater than 1. This technique ensures that each signature in Sig has at least γ candidate nodes, which reduces the chance of uneven allocation. However, the drawback of this approach is that it increases the cardinality of the b-gram cover which affects the throughput of the system (as it increases p_t, the fraction of non-empty leaf nodes – c.f. Section 3.3).

The drawback of the approach above is that the cardinality of the b-gram cover that we find is likely to be larger than the optimal b-gram cover that we find using the method described in Section 3.4.1. Since we want to keep the cardinality of the b-gram cover to be as low as possible, we use the following alternate approach: Each non-empty leaf node N that has more than a certain number of patterns allocated to it (higher than a threshold MAX), is split, and the strings in V_N are indexed in its children. The first byte of each string in V_N determines the child node where it is indexed. The remaining bytes of each string are hashed in the Bloom filter or stored in the linked list at the appropriate child node. The pseudo code for this algorithm can be found in Appendix A.1.2.

3.5 Data-Conscious sigMatch

Previous filter-based structures [15, 20, 28] that have been used for multi-pattern matching, have not exploited the characteristics of the underlying data items that are being matched against the signatures. However, exploiting the underlying data characteristics offers many opportunities for improving the performance of multi-pattern matching systems.

We have also designed techniques that make the sigTree "data conscious", to reduce the number of false positives that are generated when using the sigTree. The basic idea here is to observe the underlying data characteristics and continuously adapt the sigTree structure so that the most "discriminative" strings (i.e. the ones that are most effective in filtering out data items) from the signature database are indexed.

Appendix B.1 presents the data-conscious adaptation of the sigMatch approach, and also presents results demonstrating the effectiveness of this data-conscious approach.

4. EVALUATION AND RESULTS

In this section, we describe the results from our experiments. We tested sigMatch by integrating it with three real-world systems; namely, Bro – a network IDS, ClamAV – an anti-virus system, and DBLife [13] – an IE system that focuses on content of interest to the database community.

DBLife uses a regular expression library of about 61K signatures to match newly crawled documents to find mentions of entities (e.g. people and universities) that it is tracking. DBLife currently uses Perl for matching the regular expressions. (Perl has an highly optimized regular expression evaluation engine.)

Both Bro and ClamAV have been the focus of many pattern matching techniques (e.g. [15, 19, 20, 27, 28]). They are also both already highly optimized, as they are actually deployed in practice. In addition, both have actual and large signature databases.

Using these three real-world systems we perform actual end-to-end evaluations of sigMatch.

In our evaluation, we focus on three metrics: Throughput, Speedup

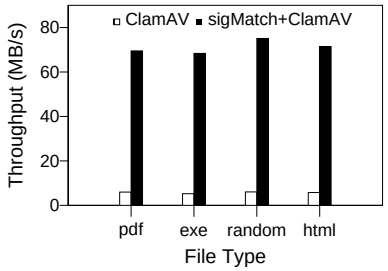


Figure 3: Evaluation with ClamAV using different file types: 89,903 signatures, b=2, β=6 bytes, Bloom filter size = 2^{14} bits.

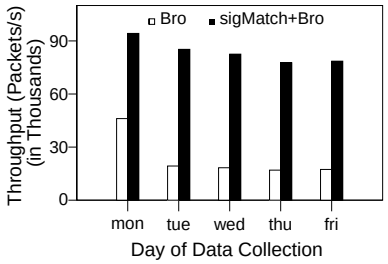


Figure 4: Evaluation with Bro using TCP payload data: 1,200 Snort signatures, b=2, β=3 bytes, Bloom filter size = 2^{14} bits.

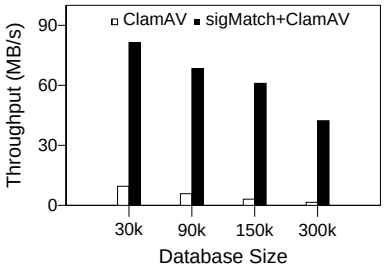


Figure 5: Evaluation with ClamAV using a 100 MB collection of .exe files and varying signature database sizes: b=2, β=6 bytes, Bloom filter size = 2^{14} bits.

System	sigTree size (KB)		L1 Cache Misses (in millions)			L2 Cache Misses (in millions)			Speedup
	Nodes only	Nodes+BF	w/o sigMatch	w/ sigMatch	% dec	w/o sigMatch	w/ sigMatch	% dec	
ClamAV (90K sigs)	23	2700	603.1	76.2	87.4	139.8	3.9	97.2	11.7X
Bro (1.2K sigs)	17	970	441.3	103.6	76.5	81.7	9.8	88.0	4.4X
DBLife (61K sigs)	21	2400	764.3	152.1	80.1	176.4	15.7	91.4	15X

Table 1: L1 and L2 cache misses for ClamAV, Bro and DBLife with and without sigMatch. Also shown in the memory usage of sigTree with and without Bloom filters (BF). ClamAV results are for the 100MB exe corpus using b=2, β=6 bytes, Bloom filter size = 2^{14} bits. Bro results are for a scan on the TCP payload for Tuesday 06/03/98 with b=2, β=3 bytes, Bloom filter size = 2^{14} bits. DBLife results are for a scan on 100 MB collection of web pages with b=2, β=4 bytes and Bloom filter size = 2^{14} bits.

and Filter Rate. *Throughput* is defined as the amount of data scanned per second. *Speedup* is defined as the throughput achieved by the system after integration with sigMatch divided by the throughput of the original system (i.e. without sigMatch). *Filter rate* is the fraction of the data items thrown out by sigMatch, i.e. the fraction of data items that are true negatives as detected by sigMatch.

All experiments were run on a 2.00 GHz Intel Core 2 Duo processor with 3 GB RAM with Ubuntu 8.04 OS, 32 KB L1 cache and 2 MB L2 cache. The cache misses quoted in the experiments were measured using the cachegrind tool in Valgrind [6].

4.1 Comparison with Current Systems

4.1.1 Comparison with ClamAV

ClamAV is a popular open-source AVS. Currently, ClamAV has 545,191 virus definitions in its database. However, only 89,903 of these signatures are *regular expression* and they account for 99.3% of the total scanning overhead in ClamAV [28]. (The other signatures are MD5 hashes, which can directly be passed to ClamAV.) In our evaluation, we use this set of ≈90K regular expressions as our signature database.

ClamAV uses a combination of banded row AC [7] and extended BM [9] algorithms to perform multi-pattern matching. In order to test the system we generated four corpuses, each containing a collection of files of a particular type. The first corpus, labeled as exe, was created from widely used Windows executables like MS Office, Firefox etc. Conference papers from SIGMOD 2009 was used to create the second corpus, which is labeled as pdf. The third corpus, html, contains html data from a crawl of the top 100 most popular websites in the United States (obtained from Alexa [1]). The last corpus, labeled as random, is made up of randomly generated files. We chose these file types as virus scanners primarily concentrate on executable files in personal computers, documents in mail servers, and html files in mail gateways and routers. To facilitate comparison, for each corpus we used only the first 100MB of data.

Figure 3 shows the comparison between the throughput of ClamAV with and without sigMatch. The speedup achieved by integrating sigMatch with ClamAV varies from 10-12X depending on the type of the corpus. The primary reason for the improvement is the difference in the cache misses between sigMatch and ClamAV.

The ClamAV data structures require nearly 21 MB of memory in this experiment. Since the L2 cache is only 2MB, most of the structure resides in the main memory. As a result, ClamAV tends to require at least one random memory access per scanned byte.

On the other hand sigMatch uses a compact indexing structure (sigTree) which occupies less than 3 MB (for 90K signatures and a Bloom filter size of 2^{14} bits). Consequently, a significant portion of the sigTree can fit in the processor caches. Also, since all the nodes of the sigTree need less than 32 KB memory (Table 1), they tend to fit within the L1 cache. Since most of the leaf nodes are empty (in the above experiments the fraction of filled nodes, p_t = 0.03), sigMatch requires only L1 cache accesses most of the time. The sigMatch method reduces the number of cache misses significantly as (a) most of its data structure is cache-resident and (b) calls to ClamAV are rare (the filter rate is between 0.93 and 0.96 for this experiment).

For example, in the above experiment with the exe corpus, integrating sigMatch with ClamAV reduces the L1 and L2 cache misses by 87.4% and 97.2% respectively. Table 1 (row 1 below the title row) provides detailed cache miss and memory usage statistics.

4.1.2 Comparison with Bro

In order to test with Bro, we used the Snort [4] signature set, which has 1,200 signatures. Bro builds a DFA "on the fly" to perform multi-pattern matching. The regular expression patterns that match the same kind of network traffic are grouped to cope with the exponential number of DFA states. The scans were performed on real world TCP packet traces obtained from the DARPA dataset provided by MIT Lincoln Laboratory [3]. We used a smaller β (3

DB Size	Memory Usage (in MB)		L1 Cache Misses (in Millions)		L2 Cache Misses (in Millions)		Filter Rate	Speedup
	ClamAV	sigMatch	ClamAV	sigMatch+ClamAV	ClamAV	sigMatch+ClamAV		
30k	8.2	1.4	352.2	66.3	42.5	1.5	0.958	8.5x
90K	21.1	2.7	603.1	76.2	139.8	3.9	0.947	11.7X
150K	40.5	3.7	2280.2	93.5	179.2	6.9	0.945	19.7X
300K	81.1	5.1	3051.9	121.2	272.7	19.0	0.939	27.8X

Table 2: L1 and L2 cache misses, and memory usage for signature databases of different sizes. Results are for the 100MB exe corpus with ClamAV, b=2, β=6 bytes, Bloom filter size = 2^{14} bits.

bytes) in our experiments with Bro as the signatures in the Snort dataset are shorter than those in the ClamAV database.

Figure 4 shows the comparison between Bro and sigMatch+Bro for the TCP payload captured for each day during a particular week (05/03/98 – 09/03/98). The sigMatch method improved the performance of Bro by a factor of 4, as it dramatically decreased the number of cache misses. For example, for the TCP payload captured on Tuesday 06/03/98, the L1 and L2 cache misses were reduced by 76.5% and 88% respectively.

Table 1 (row 2) shows the detailed cache miss and memory usage statistics for this experiment.

4.1.3 Comparison with DBLife

The DBLife signature database consists of 60931 regular expressions that provide the name variations of prominent universities and researchers in the database community. DBLife uses the Perl regular expression matching engine for multi-pattern matching. Perl builds a DFA to match the text against the patterns. We integrated sigMatch with this system and used a collection of 9914 web documents (≈100 MB) for testing.

SigMatch (with b=2, β=4 bytes and Bloom filter size = 2^{14} bits) has a filter rate of nearly 97% and improved the throughput of the multi-pattern matching system by a factor of nearly 15. This improvement is due to the decrease in the number of L1 and L2 cache misses when sigMatch is integrated into the system. In this particular experiment, sigMatch reduces the L1 and L2 cache misses by 80.2% and 91.4% respectively. Table 1 (row 3) gives the memory usage and cache miss statistics for this experiment.

4.1.4 Scalability Comparison

In order to test the scalability of the system, we developed a synthetic signature generator, similar to the one used in Erdogan et al. [15], to create larger signature databases that emulate the characteristics of the real signature database. Using this generator we generated virus signature databases that have up to 300K signatures, using the original ClamAV signature database as the model.

Figure 5 shows the difference in throughput between ClamAV and sigMatch+ClamAV for signature databases of various sizes. As can be seen in this figure, the speedup achieved by using sigMatch increases rapidly as the size of the signature database increases. While for a database of 30K signatures, the speedup achieved is only 8.5, sigMatch provides a speedup of nearly 28 for a database of 300K signatures. The primary reason for this behavior is that the memory usage of the sigTree structure increases at a modest pace as the size of the signature database increases. For example, the sigTree for 30K signatures has a memory requirement of 1.4 MB, while that for 300K signatures needs around 5 MB when each Bloom filter has 2^{14} bits.

This modest increase in the sigTree size is because the memory usage of the sigTree is dictated only by p_t (the fraction of non-empty nodes in Eq. 1) for a given Bloom filter size, and the increase in p_t with the the number of signatures is modest. Even for 300K signatures p_t is less than 0.08. Hence, a significant portion of the sigTree structure can fit in the L2 cache even for 300K signatures.

On the other hand, the memory consumption of the ClamAV data structure increases by nearly 10 times as the number of signatures increases from 30K to 300K. ClamAV has a memory usage of 8.2 MB for 30,000 signatures, and 81.1 MB for 300K signatures. As a result, the decrease in cache misses due to the addition of sigMatch also increases rapidly as the size of the signature database increases. Consequently, while the integration of sigMatch reduces the L2 cache misses by around 40 million for 30K signatures, this number increases to more than 250 million for 300K signatures.

The memory usage of sigMatch and ClamAV, and the cache miss statistics for the system with and without sigMatch, for different signature databases is shown in Table 2.

We performed similar experiments with the Bro dataset and obtained similar results which are omitted in the interest of space.

4.2 Effect of Bloom Filter Size

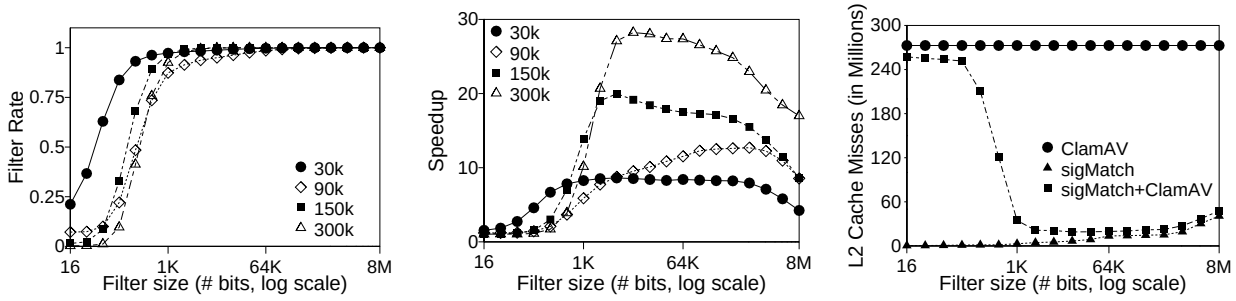
In this experiment we used the synthetic signature databases described in section 4.1.4 and compared the speedup achieved by integrating sigMatch with ClamAV for different Bloom filter sizes.

Figure 6(a) and (b) shows the effect of changing the Bloom filter size on the filter rate and the speedup respectively, for signature databases ranging from 30K to 300K signatures. From Figure 6(a) we see that initially the filter rate improves rapidly as the Bloom filter size increases. But, after a certain limit, around 16K bits in most cases, there is diminishing return in the improvements to the filter rate. In fact, as shown in Figure 6(b) after this limit, performance actually degrades.

This behavior can be explained by examining the cache miss numbers shown in Figure 6(c) for the 300K signature database. For small Bloom filter values, the filter rate is low and hence nearly all the data items are sent to ClamAV (compare the lines for ClamAV and sigMatch+ClamAV in Figure 6(c)). As a result, increasing the Bloom filter size only marginally reduces the overall number of L2 cache misses. However, as the Bloom filter size increases further, the filter rate increases and the L2 cache misses decrease rapidly as an increasing number of data items are discarded by the filter (since most cache misses are caused by random memory accesses of ClamAV). At this stage, increasing the Bloom filter size increases the L2 cache misses incurred by sigMatch, but decreases the number of data items sent to ClamAV due to the increase in the filter rate, and overall system performance improves. The performance of the system degrades when the increase in the sigMatch cache misses is not compensated by a decrease in the number of ClamAV cache misses due to the increase in filter rate. This degradation happens sooner for larger databases as their sigTree structure has a larger memory requirement.

4.3 Effect of Other sigTree Parameters

The other sigTree parameters that can impact the performance of sigMatch are: b – the height of the tree, β – the number of bytes indexed in the Bloom filter, and the Bloom filter hash functions. In our experiments, we keep b fixed at 2 as the longest substring (with-



(a) (b) (c)

Figure 6: (a) Filter Rate and (c) Speedup achieved after integrating sigMatch with ClamAV for different signature databases at various Bloom filter sizes. (b) L2 cache misses for sigMatch, ClamAV, sigMatch+ClamAV for a scan with the 300K signature database. The results are for a scan on a 100 MB collection of executable files. sigTree parameters b=2, β=6 bytes.

out any wildcard characters) of the shortest signature in all the three signature databases is 2. Increasing β doesn't have a major impact on the performance of the system as the increase in the filter rate is compensated by the increase in hash computation time and the number of signatures stored in the linked lists. Increasing the number of hash functions used improves the filter rate but increases the hash computation time in the case of a match. In our experiments we found that using a two level hash functions provides the best performance. These experiments are discussed in more detail in Appendix C.

5. CONCLUSION AND FUTURE WORK

In this paper, we have presented sigMatch, a generic, scalable and efficient method for evaluating multi-pattern matching. Our method leverages the insight that in many applications, most input data items do not match with any member in the signature database. We have developed a cache-friendly index structure that efficiently filters a large number of these "no-match" cases. Using real data sets we demonstrate that sigMatch significantly improves the performance of three state-of-the-art multi-pattern matching systems that are used in diverse domains. In addition, using synthetic signature databases, we show that sigMatch scales well to handle very large signature databases. As part of future work, we plan on exploring extension of sigMatch for multi-cores. We also plan on developing similar techniques to speed up approximate pattern matching. The idea here is to design a cache-efficient structure similar to the sigTree structure, to speed up appoximate similarity matching.

6. ACKNOWLEDGMENT

This research was supported in part by the National Science Foundation under grant DBI-0926269.

7. REFERENCES

[1] Alexa Web Rankings. `http://www.alexa.com`.
[2] ClamAV Anti-Virus System. `http://www.clamav.net/`.
[3] DARPA Intrusion Detection Evaluation Data Set. `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1998data.html`.
[4] Snort Intrusion Detection System. `http://www.snort.org/`.
[5] Spam Corp. `http://www.spamcop.net`.
[6] Valgrind. `http://www.valgrind.org/`.
[7] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *CACM*, pages 333–340, 1975.
[8] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13(7), 1970.
[9] R. S. Boyer and J. S. Moore. A Fast String Searching Algorithm. *CACM*, 20(10):762–772, 1977.
[10] K. Chakrabarti, S. Chaudhuri, V. Ganti, and D. Xin. An Efficient Filter for Approximate Membership Checking. In *SIGMOD*, pages 805–818, 2008.
[11] J. Cho and S. Rajagopalan. A Fast Regular Expression Indexing Engine. In *ICDE*, pages 418–429, 2002.
[12] B. Commentz-Walter. A String Matching Algorithm Fast on the Average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, pages 118–132, 1979.
[13] P. DeRose, W. Shen, F. C. 0002, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan. DBLife: A Community Information Management Platform for the Database Research Community (Demo). In *CIDR*, pages 169–172, 2007.
[14] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. *IEEE Micro*, 24(1):52–61, 2004.
[15] O. Erdogan and P. Cao. Hash-AV: Fast Virus Signature Scanning by Cache Resident Filters. *Int. J. Secur. Netw.*, 2(1/2):50–59, 2007.
[16] N. Koudas, A. Marathe, and D. Srivastava. Flexible String Matching Against Large Databases in Practice. In *VLDB*, pages 1078–1086, 2004.
[17] C. Li, J. Lu, and Y. Lu. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *ICDE*, pages 257–266, 2008.
[18] C. Li, B. Wang, and X. Yang. VGRAM: Improving Performance of Approximate Queries on String Collections Using Variable-Length Grams. In *VLDB*, pages 303–314, 2007.
[19] A. Majumder, R. Rastogi, and S. Vanama. Scalable Regular Expression Matching on Data Streams. In *SIGMOD*, pages 161–172, 2008.
[20] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An On-Access Anti-Virus File System. In *SSYM*, pages 73–88, 2004.
[21] L. Salmela, J. Tarhio, and J. Kytöjoki. Multi-Pattern String Matching with q-Grams. *J. Exp. Algorithmics*, 11:1–19, 2006.
[22] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *SIGCOMM*, pages 207–218, 2008.
[23] H. Song and J. W. Lockwood. Multi-Pattern Signature Matching for Hardware Network Intrusion Detection Systems. In *IEEE GLOBECOM*, pages 1686–1690, 2005.
[24] D. Starobinski, A. Trachtenberg, and S. Agarwal. Efficient PDA Synchronization. *IEEE Transactions on Mobile Computing*, 2(1):40–51, 2003.
[25] S. Wu and U. Manber. A Fast Algorithm for Multi-Pattern Searching. Technical report, University of Arizona, 1994.
[26] W. A. Wulf and S. A. Mckee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23:20–24, 1995.
[27] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *ANCS*, pages 93–102, 2006.
[28] X. Zhou, B. Xu, Y. Qi, and J. Li. MRSI: A Fast Pattern Matching Algorithm for Anti-Virus Applications. In *International Conference on Networking*, pages 256–261, 2008.

APPENDIX

A. SIGTREE CONSTRUCTION

In this section we present the pseudo code for the sigTree construction. The approach is described in Section 3.4.

Algorithm 1 outlines our approach for constructing the sigTree given the height of the tree b, the number of bytes to be hashed into the Bloom filter β, and the signature set Sig.

Algorithm 1 Construct sigTree

Input: Signature Set Sig; height of the tree b; β, the number of bytes hashed in the Bloom filter
Output: sigTree T

$Sig \leftarrow$ rewrite(Sig)
$S \leftarrow$ b-gram Cover(Sig, b, β)
$T \leftarrow$ createTree(b, S)
$V \leftarrow$ sigAlloc(Sig, S, b, β)
for each non-empty leaf node N in T **do**
  FillNode(N, V_N, β)
return T

A.1 b-gram Cover

Algorithm 2 presents the pseudo code for constructing an effective b-gram cover S given the signature set Sig, the height of the tree b, and β – the number of bytes to be hashed in the Bloom filter. The method is described in Section 3.4.1.

Algorithm 2 b-gram Cover

Input: The signature set Sig; the height of the tree b; β, the number of bytes hashed in the Bloom filter
Output: b-gram Cover S

$Str \leftarrow$ All possible strings of length b
Sort Str based on frequency.
$Set1, Set2 \leftarrow Sig$
$S \leftarrow \{\}$
$str \leftarrow$ String in Str with the highest frequency
while $Set1$ is not empty **do**
  $Found \leftarrow$ signatures in $Set1$ for which str is a RS
  if $Found$ is not empty **then**
    Remove all signatures in $Found$ from $Set1$
    Add str to S
  $str \leftarrow$ String in Str with next highest frequency
for $i \leftarrow length(S)$ to 1 **do**
  $Found \leftarrow$ signatures in $Set2$ for which $S[i]$ is a RS
  if $Found$ is not empty **then**
    Remove all signatures in $Found$ from $Set2$
  else
    Remove $S[i]$ from S
return S

A.1.1 Signature Allocation

Algorithm 3 presents the pseudo code for an effective allocation method that aims to keep the number of signatures allocated to each leaf node as uniform as possible. The inputs to this function are the signature set Sig, the height of the tree b, and β – the number of bytes to be hashed in the Bloom filter. The method is described in Section 3.4.2.

Algorithm 3 SigAlloc

Input: Signature Set Sig; S, the b-gram cover; height of the tree b; β, the number of bytes hashed in the Bloom filter
Output: V, the set of strings to be indexed in each node.

Sort Sig based on the number of candidate nodes.
for $i \leftarrow 1$ to $length(Sig)$ **do**
  $N \leftarrow$ Candidate Node of $Sig[i]$ with least number of signatures allocated to it
  $str \leftarrow$ Candidate String of N
  Add next β bytes following str in $Sig[i]$ to V_N
return V

A.1.2 FillNode

Algorithm 4 (described in Section 3.4.3) presents our method for populating each non-empty leaf node. The method works as follows: If a node is not overloaded (i.e. the number of signatures allocated to it is not greater than some threshold MAX), then the strings in the corresponding array V_N are indexed in the Bloom filter, or the linked list depending upon their length. If a node is overloaded, it is split, and its signatures are allocated to the appropriate child node.

Algorithm 4 Fill Node

Input: N, a leaf node; V_N, the set of strings to be indexed in N; β, the number of bytes indexed in the Bloom filter.

$N_l \leftarrow$ Linked List at Node N
$N_b \leftarrow$ Bloom Filter at Node N
if $length(V_N) < MAX$ **then**
  for $k \leftarrow 1$ to $length(V_N)$ **do**
    if $length[V_N[k]) < \beta$ **then**
      Add $V_N[k]$ to N_l
    else
      Add $hash(V_N[k])$ to N_b
if $length(V_N) > MAX$ **then**
  $newN \leftarrow splitNode(N)$
  for $k \leftarrow 1$ to $length(V_N)$ **do**
    $ind \leftarrow$ first byte of $V_N[k]$
    Add $rest(V_N[k])$ to $newV_{newN[ind]}$
  for $i \leftarrow 1$ to $length(newN)$ **do**
    $FillNode(newN[i], newV_{newN[i]}, \beta - 1)$
return

B. AN EXAMPLE

In this section, we present an example of the sigTree construction for the signatures given below.

Sig 1: 10cd21eff85a59b440**6606**e81902b440cd21
Sig 2: 41cd21efff0b788ffbd4**6606**245d97373
Sig 3: 305145d691*3430abacd214?**7676**92efffc278
Sig 4: **6606**cd213478710b126578*a789
Sig 5: 6174767 66e20636f**efff**670757465210ca204e
Sig 6: **88ff**ea4b41
Sig 7: **7666**abddb1*6606ab??ccca*c777888

For this example, let the sigTree parameters be b = 2 and β = 4 bytes. Each byte in the signature is represented as a two digit hexadecimal number for convenience. The wildcard character '*'

is used to indicate that any number of arbitrary bytes can be at that location, while the character '?' is used to indicate that exactly one arbitrary byte can be at that location.

The first step in the sigTree construction is to find an effective b-gram cover S for the signature set. The approach (Algorithm 2) that we use has three sub-tasks.

In the first sub-task we sort the set of all b-grams Str based on their frequency and create two copies of the signature set Sig namely $Set1$ and $Set2$.

In the second sub-task we use a greedy approach to build a b-gram cover S. Here we add the b-gram with the highest frequency str to S and remove all signatures that have str as a RS from $Set1$. The string with the next highest frequency in Str is then picked and this process is repeated. A string is only added to S if it is a RS of at least one signature left in $Set1$. This sub-task terminates when $Set1$ becomes empty.

In the final sub-task we refine S by removing the redundant strings in S. In order to do this, we loop through S in reverse, starting with the last added string. For each b-gram, we remove all signatures in $Set2$ which have this b-gram as a RS. If no signature in $Set2$ has this b-gram as a RS then it is removed from S as it is redundant.

In the example above, the RS responsible for each signature to be removed from $Set1$ (second subtask) is marked in bold, and underlined for $Set2$ (in the third subtask).

First we sort all the 2-grams based on their frequency. Initially the set $Set1$ has all the signatures in Sig. The 2-grams 6606 and cd21 have the highest frequency of 3.

Lets say we pick 6606 first. It is added to S, and signatures 1, 2 and 4 are marked as found and removed from Set1. Note that although 6606 is a substring of signature 7, it is not a RS as it doesn't have β characters following it without any wildcard characters in the signature.

Next, we pick cd21. However it is not added to S as all the signatures that have cd21 as a RS have already been accounted for by 6606.

Next, we have three 2-grams with the next highest frequency of 2, namely efff, 7676 and 88ff. Lets say we pick efff first. It is added to S and signature 5 is removed from Set1. Next 7676 is added to S and signature 4 is removed from Set1. Now we are left with only the short signatures 6 and 7 each of which have exactly one RS each – 88ff and 7666 respectively. So these two 2-grams are added to S and the second sub-task is completed as Set1 is now empty. Now, $S = \{$6606,efff,7676,88ff,7666$\}$.

Now we move to the third sub-task and refine S by removing the redundant strings. $Set2$ now contains all the signatures in Sig. First we check 7666 as it was the 2-gram last added to S. Signature 7 has 7666 as its RS, and is removed from Set2. Next we pick 88ff and remove signatures 2 and 6 from Set2 as they have 88ff as one of their RS. Next, we pick 7676 and remove signatures 5 and 3 from Set2. Then efff is picked but since all signatures that have efff as a RS have already been removed from Set2, it is redundant and hence removed from S. Finally, 6606 is picked from S and signatures 1 and 4 are removed from Set2, and the second iteration is terminated as Set2 is now empty. The b-gram cover $S = \{$88ff, 7666, 7676, 6606$\}$

The next step is signature allocation (Algorithm 3). In this step, we first sort the signature set Sig based on the number of candidate nodes that they have, and then allocate them in that order. Each signature is allocated to its candidate node N which has the least number of signatures allocated to it. For convenience, we refer to each leaf node by its candidate string. Signatures 7, 6, 5, 4, 3 and 1 have exactly one candidate node each and are allocated to the nodes

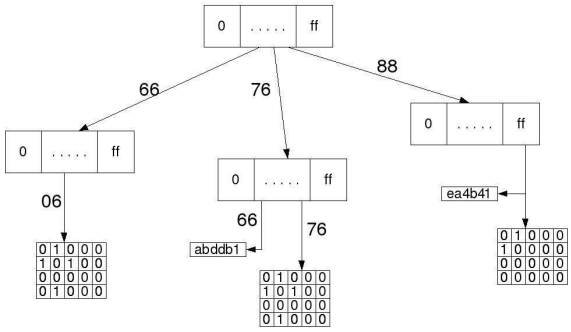


Figure 7: An example sigTree.

7666, 88ff, 7676, 6606, 7676 and 6606 respectively. Signature 2 has two candidate nodes – 88ff and 6606. The node 88ff is picked as it has fewer signatures allocated to it.

The last step is to populate the Bloom filter and linked list at the leaf nodes. Other than the four nodes corresponding to the strings in S, all other leaf nodes are empty. As the node corresponding to 7666 has just one short signature, it only has a linked list. As 88ff has a short as well as a regular signature assigned to it, it has a linked list and a Bloom filter. Since the other nodes (7676 and 6606) have only regular signatures assigned to them, they only have a Bloom filter each.

Figure 7 shows the sigTree built for the signatures.

B.1 Data-Conscious SigMatch

The intuition behind data-conscious multi-pattern matching is as follows: If the data being scanned is uniformly random, then the choice of strings (from each pattern in the signature database) that we index in the sigTree does not affect the performance of the system.

However, the distribution of patterns in the data encountered in real world application is often skewed. Ideally, we would like to pick the strings (for indexing) that have the least frequency in the data that is going to be scanned (i.e. are highly selective). If a sample of the data that the application will encounter is available beforehand, it can be used to construct a sigTree which is data-conscious.

Unfortunately, for many applications such as AVSs and IDSs, a representative dataset is not readily available as the data they encounter varies widely from one system to another. Also in many multi-pattern matching applications the data encountered changes over time. Hence, in this section, we propose an online method for modifying the sigTree using the data obtained after observing the behavior of the system for a particular period of time. This period of time is referred to as the *monitoring phase*.

Instead of indexing one substring from each signature in sigTree, the data-conscious sigMatch indexes γ substrings from each signature. Then during the monitoring phase, it gathers statistics on these γ substrings. At the end of the monitoring phase, it then prunes the index and only assigns each substring to one node. These steps are described in more detail below.

For the data-conscious sigMatch, the initial index construction step has to find a γb-gram cover for the signature set (instead of a simpler b-gram cover). The original index construction step (see

Section 3.4) is modified to assign γ nodes to each signature if possible. The rest of the index building steps remain the same.

Then, during the monitoring phase, we collect statistics about the number of false positives that are detected at each non-empty leaf node.

After the monitoring phase, we pick all leaf nodes that have more than a certain number of false positives decisions for deletion. Each leaf node marked for deletion is then removed, if possible. A leaf node can be deleted if every signature allocated to it has been allocated to at least one other leaf node. The order in which we try to delete the nodes determines the number of nodes that are actually removed. The goal here is to reduce the number of false positives by the largest possible value. For this part, we employ a greedy approach that sorts the leaf nodes marked for deletion based on the number of false positive decisions that they made. We then remove the nodes in this order starting with the node that made the largest number of false positive decisions.

Once all removable nodes are deleted, each signature in the database can still be allocated to as many as γ nodes. Next, we prune the tree further such that the fewest possible leaf nodes are left non-empty (to reduce p_t, the fraction of non-empty leaf nodes). For this step, we use an approach similar to the one described in Section 3.4.2. We first sort the signatures based on the number of nodes that they have been allocated to. For each signature, we then pick the node with the least number of signatures assigned to it, as long as it is not zero. If all candidate nodes of a signature have no signatures allocated to them, then we pick a random node amongst these choices.

Periodically, as the data changes, we use the original sigTree constructed using the γb-gram (a copy of this sigTree can be stored after it is first constructed above, so that it does not have to be rebuilt), and re-tune the tree.

Section B.1.1 describes our approach to build a DC sigTree (during the monitoring phase), and Section B.1.2 presents our approach to tune this structure according to the statistics collected during the monitoring phase to produce the "production" sigTree.

Algorithm 5 Construct DC sigTree

Input: Signature Set Sig; height of the tree b; β, the number of bytes hashed in the Bloom filter; γ, the number of nodes each signature is allocated to.
Output: sigTree T

$Sig \leftarrow$ rewrite(Sig)
$S \leftarrow \gamma$b-gram Cover(Sig, b, β, γ)
$T \leftarrow$ createTree(b, S)
$V \leftarrow \gamma$sigAlloc$(Sig, S, b, \beta, \gamma)$
for each non-empty leaf node N in T **do**
    FillNode(N, V_N, β)
return T

B.1.1 DC SigTree Construction

Algorithm 5 shows the pseudo code for constructing the DC sigTree that is used in the monitoring phase. It is similar to the sigTree construction algorithm discussed in A, but with one modification. Instead of allocating each signature to one node, we allocate it to as many as γ nodes. So instead of finding a b-gram cover, we find a γb-gram cover for the signature set Sig.

An effective γb-gram cover can be found by using a technique similar to the one described in Algorithm 2. The signature allocation algorithm also varies from the technique described in Algorithm 3. Instead of allocating a signature to the candidate node with the least number of signatures, a signature is allocated to the candidate nodes which have the γ fewest signatures allocated to them.

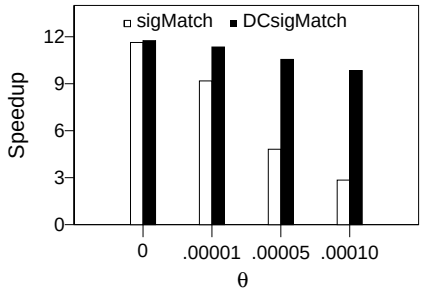


Figure 8: Comparison between sigMatch and DC sigMatch with ClamAV for different θ. These results are for a scan on a 50 MB random corpus using the the ClamAV database (89,903 signatures). b=2, β=6 bytes, Bloom filter size = 2^{14} bits, γ=5.

B.1.2 DC SigTree Tuning

Algorithm 6 gives our approach for tuning the sigTree according to the data characteristics.

During the monitoring phase, we collect the number of false positive matches detected by the Bloom filter at each non-empty leaf node.

After the monitoring phase all nodes that produced more than δ false positives are marked for potential deletion. Each of these nodes are then deleted if possible. A node can be deleted if every signature allocated to it is assigned to at least one other node. Once all removable nodes have been deleted, we then remove all the additional assignments for each signature in an effective way which ensures that the resulting tree has the least number of non-empty leaf nodes.

Algorithm 6 Tune DC sigTree

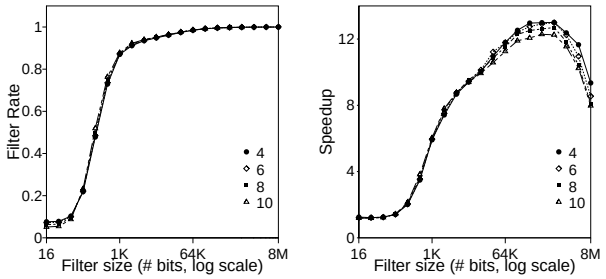
Input: Signature Set Sig; γ, the number of nodes each signature is allocated to; δ, parameter for node removal; sigTree T.

$N \leftarrow$ set of non-empty leaf nodes in T
Collect no. of false positives detected by each node in N during the monitoring phase.
sort N based on no. of false positives detected
$Remove \leftarrow \{\}$
for $i \leftarrow length(N)$ **do**
    if $N[i] < \delta$ **then**
        break;
    Add $N[i]$ to $Remove$
for $i \leftarrow length(Remove)$ **do**
    Delete $Remove[i]$ from T if possible
Remove additional assignments for each signature in Sig

B.1.3 Evaluating Data-Conscious SigMatch

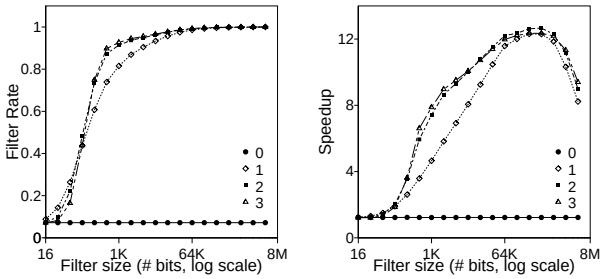
To test the data-conscious sigMatch (DC sigMatch), we generated corpuses that produce more type A false positives (see Section 3.3) using the following method: First, we collected a subset QG of the q-grams indexed in the sigTree from the 89,903 signatures in the ClamAV database. Then, we interleaved a randomly chosen member from QG into certain locations (chosen with a probability θ) in the random corpus described in Section 4.1.1.

Figure 8 compares the speedup achieved using sigMatch and DC sigMatch for different values of θ. The parameter θ provides a measure of the number of type A false positives detected in the corpus.



(a) (b)

Figure 9: (a) Filter rate and (b) Speedup achieved after integrating sigMatch with ClamAV for different β (number of bytes hashed in the Bloom filter) at various Bloom filter sizes. The results are for a scan on a 100 MB collection of executable files. The sigTree parameter b=2.



(a) (b)

Figure 10: (a) Filter Percentage and (b) Speedup achieved after integrating sigMatch with ClamAV while using different number of hash functions at various Bloom filter sizes. The results are for a scan on a 100 MB collection of executable files. The sigTree parameters are b=2 and β=6 bytes

When θ=0, the filter rate offered by sigMatch is very close to 1, so very little improvement is achieved by tuning sigMatch to be data-conscious. However, as we increase θ, the speedup achieved by sigMatch reduces rapidly, as the filter rate achieved drops due to an increase in the type A false positives. In contrast, the filter rate of DC sigMatch is not affected, and its speedup remains fairly constant. DC sigMatch improves the speedup achieved by sigMatch by a factor of nearly 3.5 for a θ value of 0.0001.

C. EFFECT OF OTHER SIGTREE PARAMETERS

The sigTree parameters are the tree height b, the length of the string indexed in the Bloom filter β, the set of hash functions used, and the Bloom filter size. The effect of Bloom filter size has already been discussed in detail in Section 4.2. In this section we discuss the impact of the other parameters namely: b, β, and the hash functions.

In our experiments, we keep b fixed at 2 as the longest substring (without any wildcard characters) of the shortest signature in both the ClamAV and Bro signature databases is 2. Also increasing the tree height increases the memory consumption of the tree dramatically as each node consumes about 1024 bytes. Even for those signature databases that have a longer minimum length, we recommend building the tree to height 2 and then splitting a node only if it has more than a certain number of patterns assigned to it. This technique can quickly reduce memory usage as very few nodes are expected to be "crowded" because of the way signatures are allocated (section 3.4.2).

C.1 Effect of the Number of Bytes Hashed in the Bloom Filter (β)

Figure 9 shows the effect of the parameter β on the filter rate and the speedup achieved when integrated with ClamAV. This experiment used a 100 MB corpus of executable files, and the 89,903 regular expression signatures in the ClamAV signature dataset.

As can be seen in Figure 9, increasing β doesn't have a major impact on the performance of the system. Normally we would expect increasing β to increase the filter rate by reducing the number of type A false positives (see Section 3.3) detected. However, in our experiments we found that increasing β does not improve the filter rate (see Figure 9(a)). Consequently, the speedup is also not significantly affected by β (see Figure 9(a) (b)). This indicates that signatures in this real world applications are distinct enough that they can be distinguished even by short substrings (around 4 bytes).

C.2 Effect of the Hash Functions

In this section we discuss the performance impact of the choice of hash functions used by sigMatch. Hash functions play a crucial role in the performance of sigMatch as they influence the number of type B false positives (section 3.2) detected by sigMatch. The criteria for hash functions is that they should be cheap to compute, and they should produce relatively random distributions in order to ensure that the collisions in the Bloom filter remain small.

If we decide to use only one hash function, we have two choices: a) we can either choose hash functions that are used for general purpose string matching, which tend to have a low collision rate but are computationally expensive, or b) we can choose a computationally cheaper hash function that is fast but can have a high collision rate.

Using multiple hash functions lets us combine the benefits of both these options. The idea here is to use multiple computationally cheap hash functions such as *xor+shift* [15] for the first few hash functions. For the last few hash functions, we choose expensive methods to reduce the number of false positives. The advantage with this technique is that the expensive hash methods are computed only when the first few hash methods return a false positive. The disadvantage with using multiple hash functions is the additional computation involved when there is a match. Also when more hash functions are used, more bits are set to 1 in the Bloom filter, which reduces the filtering efficiency.

Figure 10(b) shows the effect of using different number of hash functions in the sigTree index. We used three hash functions *mask*, *xor+shift* and *RShash* (as suggested by Erdogan et al. [15]), and tested the system by using different combinations of these methods. In this experiment, the one hash function case uses the RShash method, while the two hash function case uses the xor+shift and RShash methods in that order. The three hash function case shown in Figure 10(b) uses mask, xor+shift and RShash in that order.

As can be seen in Figure 10(b), when no hash functions are used, the Bloom filters are non-existent and the filter rate is very low. Using multiple hash functions has a clear performance advantage over using just a single hash function. Although using three hash functions provides a marginally better filter rate (Figure 10(a)), it also take a longer time to compute the true positive cases. Consequently, for large Bloom filter sizes, the performance is best when sigTree uses just two hash functions.