International Journal of Foundations of Computer Science
Vol. 20, No. 4 (2009) 597–612
© World Scientific Publishing Company

ANALYSIS OF BIT-SPLIT LANGUAGES FOR PACKET SCANNING AND EXPERIMENTS WITH WILDCARD MATCHING

RYAN DIXON, ÖMER EĞECIOĞLU and TIMOTHY SHERWOOD
Department of Computer Science,
University of California, Santa Barbara
{rsd,omer,sherwood}@cs.ucsb.edu

Received 18 May 2009
Accepted 25 June 2009
Communicated by Oscar H. Ibarra and Balasubramanian Ravikumar

Monitoring traffic payloads to detect the occurrence of suspicious patterns has proven to be a useful and necessary tool for network security. Bit-splitting breaks the problem of monitoring payloads to detect such patterns into several parallel components, each of which searches for a particular bit pattern. We analyze bit-splitting as applied to Aho-Corasick style string matching and present a formal treatment of bit-slicing to prove correctness and to provide bounds on the NFA to DFA conversion of the Aho-Corasick type machine used for bit-splitting. The problem can be viewed as the recovery of a special class of regular languages over product alphabets from a collection of homomorphic images. Furthermore, in an attempt to extend the flexibility and applicability of the original bit-splitting algorithm, we explore the expressiveness and limitations of bit-slicing with respect to wildcard matching applications.

Keywords: Packet scanning; bit-split languages; NFA-DFA conversion.

1. Introduction

Increasingly, routers are asked to play a role in scanning for, logging, and even preventing network based attacks. Signature based schemes rely on a set of signatures to describe malicious or suspicious data. While a wide variety of signature types are possible, depending on the exact nature of the intrusion detection or prevention method, a signature usually consists of at least a type of packet to search, a sequence of bytes to match, and a location where that sequence is to be searched for.

In an ideal case a signature includes a sequence of bytes which are always transmitted during a specific attack. The SQLSlammer worm, for example, sends 376 bytes to UDP port 1434 and can be detected in part by searching for the invariant framing byte 0x04 [14]. It is not uncommon to have thousands of signatures, each 4 to 40 bytes long. Searching through every byte of the payload of every packet for one of a large number of signatures quickly becomes a significant computational challenge.

One implementation concern is storage. A single state of a DFA must have 256 next-pointers each of which can address one of 10,000 states. At 448 bytes per state, the entire rule set of the intrusion detection system Snort [15] would require of 6 MB of on-chip storage.

To address these problems, bit-split Aho-Corasick machines have been proposed to reduce the storage requirements by a factor of 10, and enable scanning throughput on the order of 10 Gb/s (see [17]). While this work has demonstrated that bit-splitting works in the specific case of Aho-Corasick machines built over the Snort rule set, correctness or efficiency in the general case has not been shown. In this paper we analyze bit-splitting as applied to Aho-Corasick based string matching and prove that it works correctly in general. In addition, we prove that this approach avoids a potential combinatorial explosion observed in the simulation of NFA by DFA.

Finally in an attempt to extend the flexibility and applicability of the original bit-splitting algorithm, we report on a new process capable of matching not only character sequences, but also sequences that contain wildcards.

String matching in the context of bit-splitting can be viewed as the problem of efficiently recognizing languages of the form

$$\Sigma^*(p_1 + p_2 + \cdots + p_m) \tag{1}$$

where $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of patterns (keywords). This corresponds to locating the first index in the given packet (text) where a signature (pattern) starts.

Intrusion detection is inherently a stream problem: given the flow of bytes over a network, identify all sequences of bytes that appear suspicious. This critical operation should ideally happen at full network line rate (billions of bytes per second) and over very large sets of rules (thousands or tens of thousands) for any legal input (even in the worst case). While some intrusion detection techniques make use of context free grammars to define a language of signatures, such as the $LL(k)$ parser at the heart of STATL [9], to maintain streaming performance the majority of intrusion detection systems are far more restrictive. For example, the set P of patterns that Snort searches is a finite language. However, because Snort needs to find any member of P at any offset it is essentially a recognizer for languages of finite suffixes as in (1). While necessary for performance, restricting the language recognized will introduce greater possibility for false positives because the rules defined may include a larger subset of legal activities [a], however the vast majority of rules can be specified as a simple string that cleanly captures an attempted exploit.

[a]A false positive is a where a packet is identified as suspicious when in fact it is not part of a malicious activity. Typically intrusion detection systems include a couple of additional filtering rules that help reduce the number of false positives before reports are generated, but these rules do not need to operated at the full scan rate.

2. Packet Scanning

Due to a combination of increasing bandwidths and worm activity, efficient string matching and packet scanning techniques have become a highly active area of research. Monitoring the network for any one of a huge list of possible attacks is a significant computational challenge as bandwidths increase. It is also desirable to have strict performance guarantees so that we may be confident that important data is not missed. The idea of efficient pattern matching is obviously not new, and over the years a variety of methods have been devised for both string matching and general pattern matching. Both hardware and software-based solutions have been proposed, and to address the growing threat of polymorphic viruses we expect to see a move from strict keyword matching to the more general forms of pattern matching. While a full description of all past pattern matching techniques is not possible in the confines of this paper, we attempt to outline some of most recent approaches and highlight the past work we have most directly built upon. For an introduction to basic finite automata theory, the reader is referred to [13].

String matching is one of the most constrained forms of pattern matching, and there are currently many algorithms available with excellent asymptotic time and space complexities. Some well-known algorithms for exact string matching include: Knuth-Morris-Pratt, Boyer-Moore, and Horspool [5, 12]. Many of today's current solutions start with ideas presented in these algorithms and attempt to heavily optimize the constants involved. Unfortunately, each of these techniques only match one string at a time. Our problem is related but slightly different. We are concerned with searching a stream of data for any one of a large list of possible rules.

The Aho-Corasick algorithm [1] addresses this problem by constructing a finite automaton based on the keywords that must be matched. In this manner, a body of text can be searched for any number of keywords simultaneously by merging the keywords into a single large state machine. The key to this approach is that the state machine is a trie with back/cross edges which can be constructed and stored in linear time and space with respect to the total complexity of all the keywords. Fisk and Varghese [11] extend this idea, and present a hybrid solution combining features of both the Aho-Corasick and Boyer-Moore algorithms (good for common case performance). Tuck, et al. [19] optimize the Aho-Corasick algorithm instead by focusing on memory saving techniques using different forms of compression. If probabilistic methods are acceptable, hash-based techniques may be useful. Dharmapurikar, et al. [8] describes one approach using an FPGA. The implementation uses a set of bloom filters to search for predefined keywords in parallel. One potential downside of this approach, as noted in [10], is that a bloom filter must exist for each unique pattern length. This prohibits the use of regular expressions and has the potential to break in situations where keyword pattern lengths are highly variable. Baker and Prasanna describe multiple approaches to string matching using FPGA devices [4, 2, 3]. In [3], a modified Knuth-Morris-Pratt algorithm is developed and shown to perform well with relatively small keyword patterns. Our work extends upon the ideas presented in [17], where Tan and Sherwood demonstrate how a set of keyword search strings can be mapped to a tile-based memory architecture using bit-slicing over the output generated by the Aho-Corasick algorithm. The key advantage of bit-slicing is that the resulting state machines are far more compressed and easy to implement because a full byte can be processed each cycle with only 2 possible next states for each machine, as opposed to a monolithic machine with 256 possible next states (one for each possible input character). A more detailed comparison of the implementation issues can be found in [18].

The state of the art in network security and intrusion detection is now demanding faster algorithms that are at the same time applicable to a broader class of patterns than accepted by Aho-Corasick machines, for example full regular expressions. Some hardware-based schemes have been proposed to attack this problem, usually through some form of state machine based matching. In [16], the total amount of memory required for regular expression matching is reduced by partitioning the rules into a set of parallel running state machines. Rather than deal with the full complexity introduced by general state machines, Baker et. al. present a scheme that combines the idea of string matching with counters to help in the matching of wildcards [4]. Others have even made use of the associated search capabilities of TCAM to aid in matching [10]. Concurrent with our work, Brodie et. al have introduced a new method for compressing and efficiently operating on compressed state machines for general regular expressions [6], but there is no reason that this new technique cannot be used in conjunction with the idea of bit-splitting. While a scheme which combined these two techniques would be interesting, it is not the focus of this work.

3. Bit-Slicing Formalized

Our starting point is *bit-splitting* as described in [17] where a set of binary machines that run in parallel from a given Aho-Corasick machine M are constructed. Each machine searches for one bit of the input at a time, and a match occurs only when all of the machines agree. Since the split machines have exactly two possible next states they are far easier to compact into a small amount of memory. Also they are loosely coupled, and they can be run independently of one another.

3.1. *The general case of two alphabets*

The alphabet of M can be thought of as being $\Sigma = \{0,1\}^8$. The correctness and performance of bit-splitting has to do with languages defined over alphabets which are Cartesian products of other alphabets, binary or otherwise.

Consider a DFA where the input alphabet is a Cartesian product of two alphabets. Such an automaton is a finite state machine $M = (Q, \Sigma, \delta, q_1, F)$ where $Q = \{q_1, q_2, \ldots, q_m\}$ is a set of states, q_1 is the start state, and $\delta : Q \times \Sigma \to Q$ is the transition function and $F \subseteq Q$ is the set of final states.

Suppose $\Sigma = A \times B$ for $A = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ and $B = \{\beta_1, \beta_2, \ldots, \beta_s\}$. We further assume that $r, s \geq 2$.

Let $\mathcal{L} = \mathcal{L}(M)$ denote the language accepted by M. Each $w \in \mathcal{L}$ is of the form $w = a_1b_1\ a_2b_2 \cdots a_nb_n$ for some $n \geq 0$ and $a_i \in A, b_i \in B$ for $i = 1, 2, \ldots, n$.

M can be "bit-split" to construct two nondeterministic finite state machines M_A and M_B. This is done by changing the alphabet and the transition function of M, but not the set of states, the initial state, or the set of final states, in the following manner.

Definition 1. *Given a DFA* $M = (Q, \Sigma, \delta, q_1, F)$ *where* $\Sigma = A \times B$ *with* $A = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ *and* $B = \{\beta_1, \beta_2, \ldots, \beta_s\}$, *define*

$$M_A = (Q, A, \delta_A, q_1, F) \textit{ where } \forall a \in A, q \in Q, \quad \delta_A(q, a) = \bigcup_{j=1}^{s} \delta(q, a\beta_j)\ ,$$

$$M_B = (Q, B, \delta_B, q_1, F) \textit{ where } \forall b \in A, q \in Q, \quad \delta_B(q, b) = \bigcup_{i=1}^{r} \delta(q, \alpha_i b)\ .$$

M_A *and* M_B *are called* bit-split automata *or* projection automata *obtained from* M. $\mathcal{L}_A = \mathcal{L}(M_A)$ *and* $\mathcal{L}_B = \mathcal{L}(M_B)$ *denote the languages accepted by* M_A *and* M_B, *respectively.*

M_A and M_B can be described in a number of ways. Probably the easiest visualization is as follows: To construct the transition diagram of M_A, make a copy of M and erase the second letter in every transition in the transition diagram of M. M_B is constructed similarly. Since $r, s \geq 2$, M_A and M_B are both nondeterministic. The final step in bit-splitting is to take M_A and M_B and construct an equivalent DFA DM_A to M_A and an equivalent DFA DM_A to M_B. This final step is very important from an implementation standpoint, both because DFA can be more efficiently implemented on real machines [b] and at the same time, the construction from NFA to DFA in general has the potential to increase the number of states exponentially.

The languages $\mathcal{L}_A$ and $\mathcal{L}_B$ are easily seen to be homomorphic images of $\mathcal{L}$. For example, if we define the homomorphism $h_A : \Sigma \to A$ by setting $h_A(\alpha_i\beta_j) = \alpha_i$ for every letter $\alpha_i\beta_j \in \Sigma$ for $i = 1, 2, \ldots, r$, $j = 1, 2, \ldots, s$, then $\mathcal{L}_A = h_A(\mathcal{L})$. In particular, given a regular expression R denoting $\mathcal{L}$, a regular expression for $\mathcal{L}_A$ is obtained from R by replacing each occurrence of the letter $\alpha_i\beta_j$ by α_i, and a regular expression for $\mathcal{L}_B$ is obtained from R by replacing each occurrence of $\alpha_i\beta_j$ by β_j.

Example: When $\Sigma = A \times B$ with $A = \{0, 1\}$ and $B = \{a, b\}$, the language $\mathcal{L}$ over Σ denoted by the regular expression $(0a + 0b + 1a + 1b)^*0b$ results in the languages $\mathcal{L}_A$ over A and $\mathcal{L}_B$ over B denoted by the regular expressions $(0 + 1)^*0$ and $(a + b)^*b$, respectively. The transition diagrams of M, M_A and M_B are given in Figure 1.

[b]To simulate an NFA of size N processing a single input character requires N steps in the worst case as each state needs to be updated. A DFA in comparison requires only a single step because it can be in one and only one state at a time. Given the very large number of states the application demands, a DFA is required for efficient traversal.

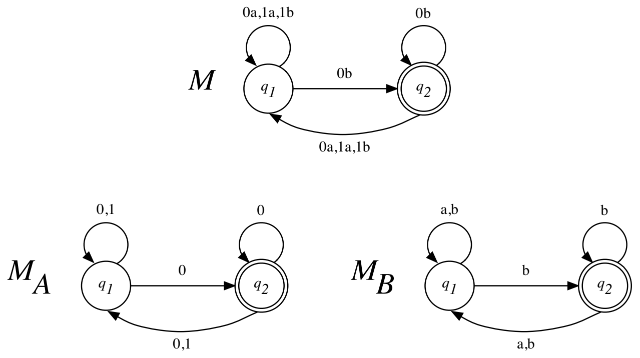


Fig. 1. M_A and M_B from M: $\Sigma = A \times B$ with $A = \{0, 1\}$, $B = \{a, b\}$.

Definition 2. *Given $\mathcal{L}_A$ over A and $\mathcal{L}_B$ over B, the language $Alt(\mathcal{L}_A, \mathcal{L}_B)$ over $A \times B$ is defined by*

$$Alt(\mathcal{L}_A, \mathcal{L}_B) = \{a_1b_1\ a_2b_2 \cdots a_nb_n \mid n \geq 0, a_1a_2 \cdots a_n \in \mathcal{L}_A, b_1b_2 \cdots b_n \in \mathcal{L}_B\}.$$

The problems that we formalize in this paper come down to the recovery of $\mathcal{L}$ from $\mathcal{L}_A$ and $\mathcal{L}_B$, and the state complexity of the conversion of M_A to DM_A and M_B to DM_B for Aho-Corasick machines.

Lemma 3. *Suppose $w = a_1b_1\ a_2b_2\ \cdots\ a_nb_n \in \mathcal{L}$, where $\mathcal{L} = \mathcal{L}(M)$ for a DFA M over the alphabet $A \times B$, as in Definition 1. Then $a_1a_2 \cdots a_n \in \mathcal{L}_A$ and $b_1b_2 \cdots b_n \in \mathcal{L}_B$. In other words*

$$\mathcal{L} \subseteq Alt(\mathcal{L}_A, \mathcal{L}_B). \tag{2}$$

Proof. Suppose $a_1b_1\ a_2b_2 \cdots a_nb_n \in \mathcal{L}$. Then there are states $q_{i_1}, q_{i_2}, \ldots, q_{i_{n+1}}$ in Q with $q_1 = q_{i_1}$ and $q_{i_{n+1}} \in F$ with $\delta(q_{i_j}, a_jb_j) = q_{i_{j+1}}$ for $j = 1, 2, \ldots, n$. By the definition of δ_A, $q_{i_{j+1}} \in \delta_A(q_{i_j}, a_j)$ for $j = 1, 2, \ldots, n$. Furthermore $q_{i_{n+1}}$ is also a final state of M_A. Thus $a_1a_2 \cdots a_n \in \mathcal{L}_A$. Similarly $b_1b_2 \cdots b_n \in \mathcal{L}_B$. Therefore every $a_1b_1\ a_2b_2 \cdots a_nb_n \in \mathcal{L}$ belongs to $Alt(\mathcal{L}_A, \mathcal{L}_B)$ and (2) follows. □

Remark: Equality in (2) does not necessarily hold. For example when $\Sigma = \{0, 1\} \times \{a, b\}$ and $\mathcal{L}$ over Σ is the language denoted by the $(0a + 0b + 1a + 1b)^*(0b + 1a)$, $\mathcal{L}_A$ and $\mathcal{L}_B$ are the languages denoted by the regular expressions $(0 + 1)^*(0 + 1)$ and $(a + b)^*(a + b)$, respectively. Thus $a_1 = 0$ and $b_1 = a$ are in $\mathcal{L}_A$ and $\mathcal{L}_B$, respectively. Therefore $a_1b_1 = 0a \in Alt(\mathcal{L}_A, \mathcal{L}_B)$, but $0a \notin \mathcal{L}$.

Definition 4. *Suppose $\mathcal{L} = \mathcal{L}(M)$ where M is a DFA over $\Sigma = A \times B$. $\mathcal{L}$ satisfies the* alternation property *if for every $n \geq 0$, $a_i, x_i \in A$, $b_i, y_i \in B$ for $i = 1, 2, \ldots, n$,*

$$a_1y_1\ a_2y_2 \cdots a_ny_n,\ x_1b_1\ x_2b_2 \cdots x_nb_n \in \mathcal{L} \ \text{ implies } \ a_1b_1\ a_2b_2 \cdots a_nb_n \in \mathcal{L}\ . \tag{3}$$

This property suffices to prove equality in (2).

Proposition 5. *Suppose $\mathcal{L} = \mathcal{L}(M)$ over the alphabet $\Sigma = A \times B$, $\mathcal{L}_A$ and $\mathcal{L}_B$ defined as in Definition 1. If $\mathcal{L}$ has the alternation property, then $\mathcal{L} = Alt(\mathcal{L}_A, \mathcal{L}_B)$.*

Proof. By Lemma 3, we have $\mathcal{L} \subseteq Alt(\mathcal{L}_A, \mathcal{L}_B)$. To show $Alt(\mathcal{L}_A, \mathcal{L}_B) \subseteq \mathcal{L}$, assume $a_1b_1\ a_2b_2 \cdots a_nb_n \in Alt(\mathcal{L}_A, \mathcal{L}_B)$ for some $n \geq 0$. By definition of $Alt(\mathcal{L}_A, \mathcal{L}_B)$, $a_1a_2 \cdots a_n \in \mathcal{L}_A$ and $b_1b_2 \cdots b_n \in \mathcal{L}_B$. First we show that $a_1a_2 \cdots a_n \in \mathcal{L}_A$ implies that there exist $y_1, y_2, \ldots, y_n \in B$ with $a_1y_1\, a_2y_2 \cdots a_ny_n \in \mathcal{L}$. Consider a sequence of states $q_{i_1}, q_{i_2}, \ldots, q_{i_{n+1}}$ in Q with $q_1 = q_{i_1}$ and $q_{i_{n+1}} \in F$ with $q_{i_{j+1}} \in \delta_A(q_{i_j}, a_j)$ for $j = 1, 2, \ldots, n$. By definition of δ_A, $q_{i_{j+1}} = \delta(q_{i_j}, a_j\beta_{k_j})$ for some $\beta_{k_j} \in B$ and we can take $y_j = \beta_{k_j}$ for $j = 1, 2, \ldots, n$. Similarly, $b_1b_2 \cdots b_n \in \mathcal{L}_B$ implies that there exist $x_1, x_2, \ldots, x_n \in A$ with $x_1b_1\, x_2b_2 \cdots x_nb_n \in \mathcal{L}$. Since $\mathcal{L}$ satisfies the alternation property, we have $a_1b_1\ a_2b_2 \cdots a_nb_n \in \mathcal{L}$. Thus $Alt(\mathcal{L}_A, \mathcal{L}_B) \subseteq \mathcal{L}$. □

Lemma 6. *The language $\mathcal{L} = \mathcal{L}(M)$ accepted by a Aho-Corasick machine M with a single keyword satisfies the alternation property.*

Proof. $\mathcal{L}$ is of the form $\Sigma^* p$ where $\Sigma = A \times B$ and p is the keyword. With the notation of Definition 4, $a_1y_1\, a_2y_2 \cdots a_ny_n,\ x_1b_1\, x_2b_2 \cdots x_nb_n \in \mathcal{L}$ implies that for some k,

$$a_1y_1\ a_2y_2 \cdots a_ny_n = a_1y_1\ a_2y_2 \cdots a_ky_k\ p\ ,$$
$$x_1b_1\ x_2b_2 \cdots x_nb_n = x_1b_1\ x_2b_2 \cdots x_kb_k\ p\ .$$

Therefore $a_1b_1\ a_2b_2 \cdots a_nb_n = a_1b_1\ a_2b_2 \cdots a_kb_k\ p \in \mathcal{L}$. □

The language $\mathcal{L}$ we are interested in is a finite union of languages of the form $\Sigma^* p$, where the union is over the keywords p. However $\mathcal{L}$ in this generality need not satisfy the alternation property of Proposition 5.

It is possible to have an exponential blow-up in the number of states of a DFA for a language $\mathcal{L}$ and the minimum state DFA for its homomorphic image $h(\mathcal{L})$, even if the homomorphism just identifies a pair of letters of the alphabet, e.g. a homomorphism such as

$$h : \{a, b, c\} \to \{a, b\}^*, \text{ where } h(a) = a, \quad h(b) = b, \quad h(c) = b. \tag{4}$$

Example: Let $\Sigma = \{a, b, c\}$. Given an integer $k > 0$, consider the DFA M on $k+2$ states shown in Figure 2. M accepts the language $\mathcal{L}$ denoted by $(a+b)^*c(a+b)^{k-1}$. The homomorphic image of $\mathcal{L}$ under the homomorphism (4) is given by $(a+b)^*b(a+b)^{k-1}$. It is well-known that the minimum state DFA for this latter language requires $\Omega(2^k)$ states.

3.2. *NFA to DFA conversion in bit-splitting*

We can show that for any Aho-Corasick pattern matching machine, the projection automata M_A and M_B in our construction do not blow up in size when converted to the equivalent DFA DM_A and DM_B.

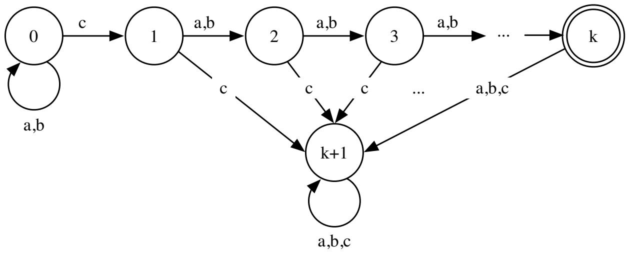


Fig. 2. DFA for a language whose homomorphic image requires $\Omega(2^k)$ DFA states.

Recall that the Aho-Corasick algorithm [1] constructs a special state machine which is essentially a trie with back/cross edges, that can be constructed and stored in linear time and space with respect to the total complexity of all the keywords. The preprocessing for the construction is in two stages. The first stage builds up a tree of all keyword strings. The tree has a branching factor equal to the number of symbols in the language, and is thus a *trie.* The root represents the state where no strings have been even partially matched. To match a string, we start at the root node and traverse down the edges according to the input characters observed. The second half of the preprocessing is inserting failure edges. When a string match is not found, it is possible for the suffix of one keyword to match a prefix of another. To handle this case, transitions are inserted which shortcut from a partial match of one string to a partial match of another. In the Aho-Corasick automaton, there is a one-to-one correspondence between accepting states and strings, where each accepting state indicates the match to a unique keyword.

Proposition 7. *Suppose M is a Aho-Corasick automaton on n states over $\Sigma = A \times B$ and M_A, M_B are the two NFA obtained from M using bit-splitting. Then the equivalent DFA DM_A and DM_B each have at most n states.*

Proof. M is built on a trie for a set of keywords $P = \{p_1, p_2, \ldots, p_m\}$ with a number of back and cross edges defined by the longest proper suffix that is also a prefix of some keyword, as described above and in detail in [1].

It suffices to show that the trie part of M_A (and M_B) has no more than n states, as the back and cross edges for DM_A and DM_B are constructed by the longest proper suffix condition for the patterns obtained from P after collapsing the alphabets to A and B, and this process does not change the number of states.

Note that we can obtain M_A from M in stages, where in each stage a pair of letters of the current alphabet are identified and the alphabet is reduced in size by one. For example starting with $A \times B = \{0,1\} \times \{a,b,c\} = \{0a, 0b, 0c, 1a, 1b, 1c\}$, we can identify $1c$ and $1b$, and then $1b$ with $1a$ obtaining the intermediate alphabet $\{0a, 0b, 0c, 1a\}$. Then we can identify $0c$ and $0b$, and then $0b$ with $0a$ obtaining $\{0a, 1a\}$, which is a copy of A. Thus it suffices to show that when only two letters

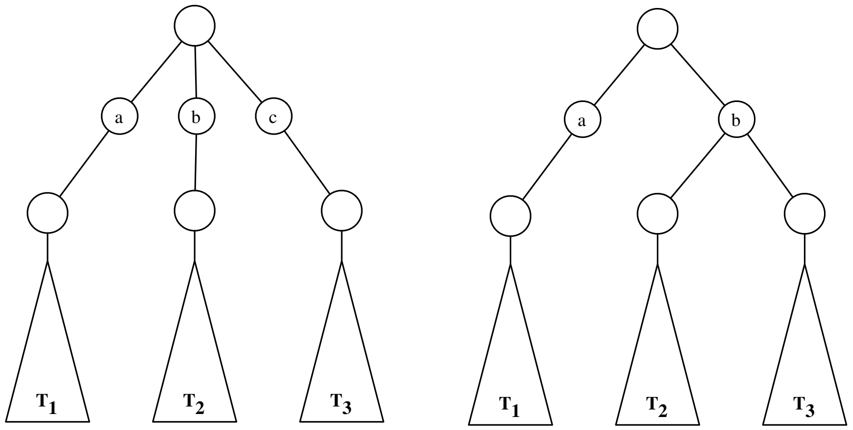


Fig. 3. Identification of b and c: at the root of the trie.

are identified, the resulting machine has a deterministic counterpart with no more than n states.

Suppose we are given a trie T of an Aho-Corasick machine M on some alphabet Σ (which need not be a product of two alphabets), and we identify two letters $b, c \in \Sigma$. In T, we first replace each occurrence of c by b. The resulting structure is a nondeterministic trie, in the sense that a node can have more than one child labeled by the letter b. As the second step, we identify nodes of the trie top down, level by level, and at each level, from left to right. At the root of the trie, we identify the children of the root indexed by the letter b. At other nodes, we may also need to identify children of a node labeled by the same letter for letters other than b, because identifications at the previous level may produce more than one child in an identified node that is labeled by a letter other than b. In addition, if any one of the identified nodes is a final state of the original machine, then the node obtained by the identification is made into a final state of the resulting machine. Since a sequence of identifications can only decrease the number of nodes of the trie, the result follows. □

Note that the identifications can produce multiple back edges or cross edges if we keep these edges in addition to the trie structure when we execute the two steps in the proof above. The final step in creating the Aho-Corasick machine requires the elimination of multiple edges of this type which may have been created by the identification nodes. In other words, we need not recompute the back and cross edges anew for each the new set of keywords obtained by identifying a pair of letters. Figure 3 and Figure 4 show the operation of identification on root and non-root nodes of the trie.

Example: The trie in Figure 5 is built on the patterns $P = \{abbc, abcc, bab, bba, ca, cba, cc\}$ over $\Sigma = \{a, b, c\}$. Identification of c and b results in the trie in Figure 6

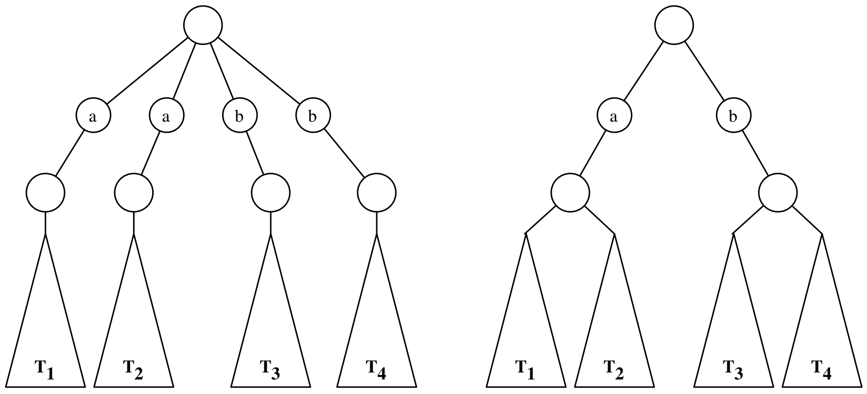


Fig. 4. Identification of b and c: at an arbitrary node of the trie.

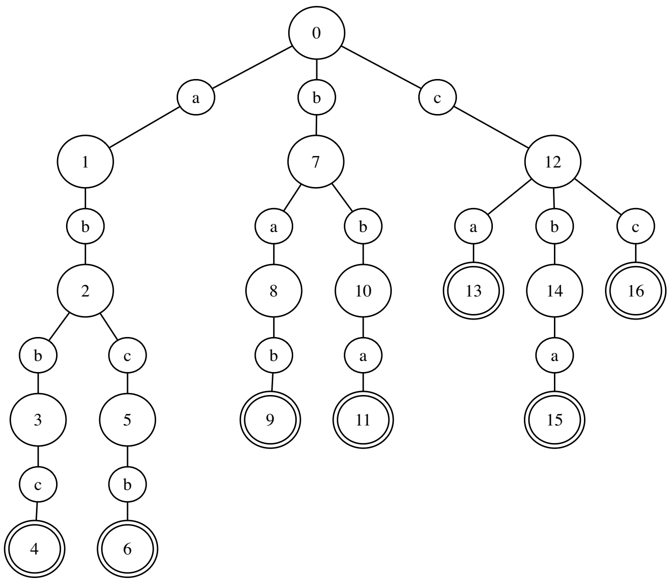


Fig. 5. Trie portion of the Aho-Corasick machine for the keywords $\{abbc$, $abcc$, bab, bba, ca, cba, $cc\}$ over $\Sigma = \{a, b, c\}$.

built on the set of patterns $\{abbb, bab, bba, ba, bb\}$ over $\Sigma = \{a, b\}$.

3.3. *Recovering $\mathcal{L}$*

If there is a single pattern p, then the language $\mathcal{L}_M$ is the form $\Sigma^* p$. Since this language has the alternation property of Definition 4, $\mathcal{L}$ can be recovered completely from the knowledge of $\mathcal{L}_A$ and $\mathcal{L}_B$. Thus by Proposition 5 the input

$a_1b_1\ a_2b_2 \cdots a_nb_n \in \mathcal{L}$ iff $a_1a_2 \cdots a_n \in \mathcal{L}_A$ and $b_1b_2 \cdots b_n \in \mathcal{L}_B$. But this works because both M_A and M_B have a single final state, i.e. the unique final state of M that corresponds to the keyword p. When there is more than one keyword, $\mathcal{L}$ no longer satisfies the alternation property, and therefore equality of the languages in Proposition 5 does not hold. However we can recover $\mathcal{L}$ from M_A and M_B by considering a type of diagonal acceptance as follows

Proposition 8. *Suppose $\mathcal{L} = \mathcal{L}(M)$ over the alphabet $\Sigma = A \times B$ for some Aho-Corasick machine M. Define $M_A(f)$ and $M_B(f)$ as in Definition 1, except a fixed $f \in F$ is made the final state. For $a_1a_2 \cdots a_n \in \mathcal{L}_A$ and $b_1b_2 \cdots b_n \in \mathcal{L}_B$, $a_1b_1\ a_2b_2 \cdots a_nb_n \in \mathcal{L}$ iff $a_1a_2 \cdots a_n \in \mathcal{L}(M_A(f))$ and $b_1b_2 \cdots b_n \in \mathcal{L}(M_B(f))$ for some $f \in F$.*

Proof. An Aho-Corasick machine M accepts languages of the form (1). The condition of the proposition forces M_A and M_B to accept by the same final state. Thus for each final state, the language accepted is of the form $\Sigma^* p_i$, and therefore satisfies the alternation property and Lemma 6 is applicable. □

Remarks: Note that we are not able to recover $\mathcal{L}$ from an arbitrary description of the languages $\mathcal{L}_A$ and $\mathcal{L}_B$ for more than one pattern. However for the packet scanning application, this presents no problems. We make sure that the M_A and M_B accept on the same final state. Otherwise the input is rejected.

If we use the deterministic versions of M_A and M_B obtained by the algorithm described in the proof of Proposition 7 and keep the names of the identified final as an equivalence class, then we can still recover $\mathcal{L}$ by acceptance by the "same" final state, meaning that there is a common final state in the two equivalence classes of names after identifications in the resulting DFA.

The results given above for the Cartesian product of two alphabets readily generalize to $\Sigma = A_1 \times A_2 \times \cdots \times A_m$. We omit the details of the general case. In particular, $\Sigma = \{0,1\}^8$, results in the 8 binary machines $M_0, M_1, \ldots, M_7$ of the bit-split Aho-Corasick.

4. Bit-Sliced Wildcards

In an attempt to extend the flexibility and applicability of the original bit-splitting algorithm, we investigated a new process that is capable of matching not only character sequences, but also sequences that contain wildcards. Specifically, wildcards given by single letters of the alphabet Σ.

4.1. *Wildcard bit-split construction and analysis*

The ability to reduce strings containing wildcards into a functionally correct bit-split machine hinges on one key observation: the Aho-Corasick DFA is actually an NFA representation of the final bit-split machine. This means that in order to

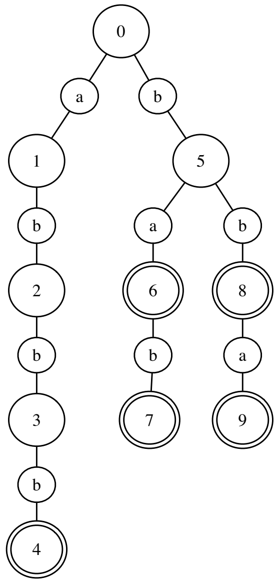
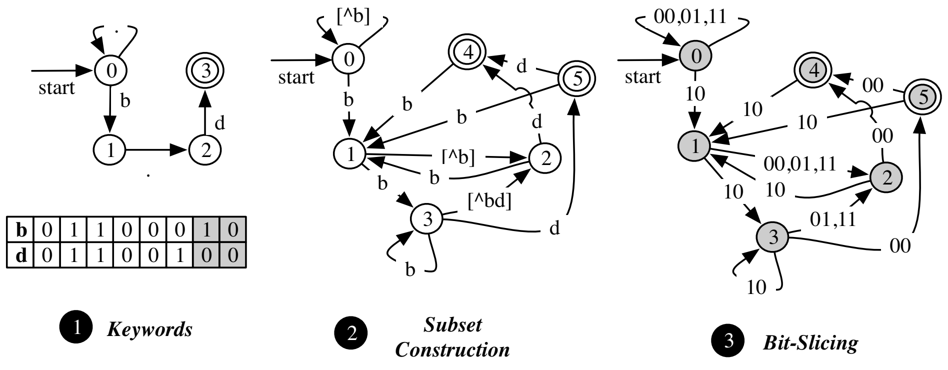


Fig. 6. After identifying c and b, the resulting trie of the Aho-Corasick machine for the keywords $\{abbb, bab, bba, ba, bb\}$ over $\Sigma = \{a, b\}$.

produce a functionally correct bit-split machine, we simply need to follow the rules of subset construction. The advantage of having greater expressive power, however, comes with a cost. As we will demonstrate, one potential downside of extending the accepted language is that we are no longer protected by the space bound guarantees of the original design.

The process of constructing bit-split machines capable of detecting strings containing wildcards requires multiple applications of subset construction. Figure 4.1 provides an outline of the conversion process. We modify the first step of the Aho-Corasick algorithm so that our initial graph is an NFA. The start state contains an edge back to itself that essentially acts as a wildcard. As we construct our NFA, we add edges in an identical fashion to the Aho-Corasick algorithm. If a wildcard is encountered, 256 edges, representing each element in the ASCII character set Σ, are added as transitions to the next state.

Once the NFA is fully constructed, we begin the first application of subset construction. The graph produced is a DFA capable of detecting the full set of keywords. To produce the desired bit-split state machines, we must perform subset construction over the keyword DFA. The number of times we must perform subset construction over the DFA depends exclusively on how many bits we want the bit-split machines to evaluate per cycle. In the example we have provided, we perform subset construction four times over the keyword DFA in order to create four bit-split state machines that each interpret two bits per cycle.



b	0	1	1	0	0	0	1	0
d	0	1	1	0	0	1	0	0

Fig. 7. A simple example of bit-slicing a language with wildcards to recognize $\Sigma\Sigma^*(b\Sigma d)$. The first stage is identical to the construction used in the Aho-Corasick algorithm [1] with the addition of an explicit wildcard back edge to and from the start state. The second stage graph is generated by performing subset construction over the initial graph and then bit-sliced according to the number of bits per cycle read by the architecture. [∧b] and [∧bd] denote the complements of $\{b\}$ and $\{b, d\}$ in Σ. In this example, the architecture reads two bits per cycle, so four bit-sliced graphs are created. The final state machines, capable of recognizing the original set of keywords, requires one final application of subset construction over each of the bit-sliced graphs. The example highlighted in gray follows the required transformations using the two rightmost bits. A detailed explanation of the final output can be found in [17].

While the transformation from keyword strings to binary state machines is relatively simple, there is still potential for even basic keyword patterns to cause significant storage and/or processing penalties. Without wildcards, the bit-splitting algorithm guarantees space bounds on the BSMs such that they will not exceed the size of the DFA from which they are composed. With the addition of wildcards, however, this guarantee no longer holds. Although the random insertion of wildcards throughout a set of keywords may not appear to have an adverse effect on the space requirements, a worst-case scenario can easily be produced. By increasing the number of consecutive wildcards the final storage requirements grow exponentially. If it is known that a large number of wildcards will occur consecutively, this approach will not suffice. It is possible that this technique could be augmented with a counting mechanism to overcome space limitations and we hope to address this issue in future work.

The increased amount of processing time and space needed to produce the BSM graphs leads to a number of concerns. Namely, how practical is this approach when dealing with content from Internet-based attacks?

4.2. *Practical concerns about wildcard bit-splitting*

To place our results within the context of network intrusion detection, we extracted signatures from real-world intrusion detection systems and evaluated the text patterns used to search for and identify malicious content. Our results indicate that

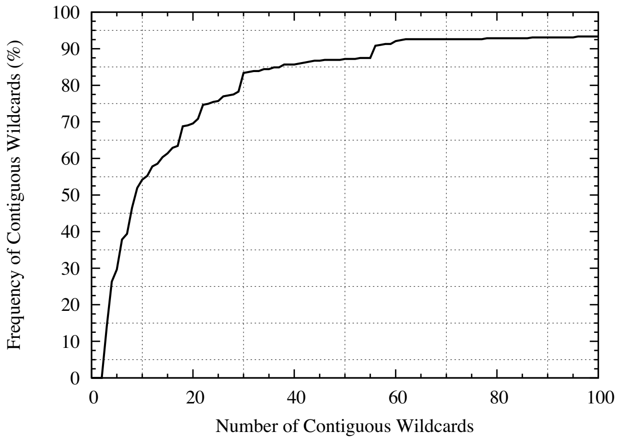


Fig. 8. Using signatures generated to detect polymorphic and metamorphic worms [7], we evaluated the frequency of contiguous wildcards needed to identify current Internet-based attacks. The original signatures provided byte ranges for invariant sequences of known malicious content. Our results indicate that a large number of consecutive wildcards would be required to capture a majority of the attack patterns.

while there may be instances where single wildcard character matching would be beneficial, in many cases, it is simply unable to meet the demands of current attack patterns.

Figure 4.2 is an evaluation of the signatures used to detect polymorphic and metamorphic attacks as presented in [7]. Our goal was to identify how well our wildcard pattern matching system would scale in order to incorporate these signatures. Noting the difficulty of handling too many contiguous wildcards, our key concern was the frequency in which contiguous polymorphic bytes occur. In other words, is it possible to accurately recreate the signatures using single-character wildcards even with their known space limitations?

In Figure 4.2 we examine how many contiguous wildcards would be required to reproduce the signatures in [7]. The x-axis identifies the number of contiguous wildcards discovered between invariant byte sequences. The y-axis describes the cumulative percentage of contiguous wildcards. Therefore, if we wanted to successfully reproduce 80% of the signatures present, we would need to use between 20 to 40 contiguous wildcards in our keyword strings. The results of this graph imply that it may be impractical to use single character wildcards to approximate the signatures. When used in conjunction with our current architectural constraints, we discovered that seven contiguous wildcards would nearly fill the available memory space. According to the numbers in the figure, limiting ourselves to only seven contiguous wildcards would cover significantly less than 10% of the entire set of signatures. As such, we believe that this method, by itself, is not suitable for the purpose of matching intrusion detection signatures.

5. Conclusions

While the need for efficient pattern matching techniques has existed for many years now, applications in networking and security environment necessitate strong guarantees on worst-case performance and highly efficient storage. In this paper we formally describe and verify a technique known as bit-splitting. We prove for the first time that bit-splitting Aho-Corasick machines is functionally correct, and provide strict space and time bounds for this approach. One open formal problem remaining is that the language property described in Proposition 3 is sufficient, but perhaps not necessary to preserve correctness. However, even without a necessary condition, the formal description of how and why bit-splitting works opens the door to new potential applications, including the use of bit-splitting for other classes of languages or other problem domains.

Using this formal framework we have explored the possibility of using bit-splitting to search for patterns embedded with single character wildcards. We have shown that bit-splitting will still be functionally correct (the language recognized by the bit-split machines is identical to that of the original machine) and that the time required for search will not be affected. However, before such a scheme could be useful in practice the space overhead might have to be reduced. If there is a significant number of wildcard rules, the size of the bit-split machines can grow exponentially do to the required subset construction step. As we have shown, to capture polymorphic viruses in the most general way possible, a significant number of wildcards will be required in the future. Attacks like the SQLSlammer worm, that transfer only a specific number of bytes, imply the need for more general pattern matching techniques. Future work could perhaps address this problem with a clever combination of bit-splitting (as we present) and counters (as presented in [4]).

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Z. K. Baker and V. K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on FPGAs. In *Proceedings of the Field-Programmable Custom Computing Machines*, pages 135–144, 2004.

[3] Z. K. Baker and V. K. Prasanna. Time and area efficient pattern matching on FPGAs. In *Proceeding of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 223–232, 2004.

[4] Z. K. Baker and V. K. Prasanna. High-throughput Linked-Pattern Matching for Intrusion Detection Systems. In *Proceedings of the First Annual ACM Symposium on Architectures for Networking and Communications Systems*, 2005.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):761–772, 1977.

[6] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *International Symposium on Computer Architecture*, 2006.

[7] J. R. Crandall, Z. Su, and S. F. Wu. On deriving unknown vulnerabilities from zero-

day polymorphic and metamorphic worm exploits. In *ACM Conference on Computer and Communications Security*, pages 235–248, 2005.

[8] S. Dharmapurikar, M. Attig, and J. Lockwood. Deep packet inspection using parallel bloom filters. *Micro, IEEE*, 24(1):52–61, 2004.

[9] S. T. Eckmann, G. Vigna, and R. A. Kemmerer. STATL: An attack language for state-based intrusion detection. *Journal of Computer Security*, 10(1/2):71–104, 2002.

[10] Y. Fang, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *ICNP*, pages 174–183, 2004.

[11] M. Fisk and G. Varghese. Applying fast string matching to intrusion detection. Technical Report In preparation, successor to UCSD TR CS2001-0670, University of California, San Diego.

[12] R. N. Horspool. Practical fast searching in strings. *Software—Practice and Experience*, 10(6):501–506, June 1980.

[13] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[14] J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[15] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of LISA'99: 13th Systems Administration Conference*, pages 229–238, November 1999.

[16] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *Proceedings of the Field-Programmable Custom Computing Machines*, pages 258–267, 2004.

[17] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 112–122, 2005.

[18] L. Tan, B. Brotherton and T. Sherwood. Bit-Split String-Matching Engines for Intrusion Detection and Prevention. In *ACM Transactions on Architecture and Code Optimization (TACO),* Vol 3 No 1, June 2006.

[19] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *the 23rd Conference of the IEEE Communications Society (Infocomm)*, March 2004.