Automata-Theoretic Analysis of Bit-Split Languages for Packet Scanning

Ryan Dixon, Ömer Eğecioğlu, and Timothy Sherwood

Department of Computer Science
University of California, Santa Barbara
{rsd,omer,sherwood}@cs.ucsb.edu

Abstract. Bit-splitting breaks the problem of monitoring traffic payloads to detect the occurrence of suspicious 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. The problem can be viewed as the recovery of a special class of regular languages over product alphabets from a collection of homomorphic images. We use this characterization to prove correctness and to give space bounds. In particular we show that the NFA to DFA conversion of the Aho-Corasick type machine used for bit-splitting incurs only linear overhead.

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 [4]. 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 [5] 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 [6]). While this work has demonstrated

O.H. Ibarra and B. Ravikumar (Eds.): CIAA 2008, LNCS 5148, pp. 141–150, 2008.

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.

String matching in this context 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.

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 [3]. The majority of intrusion detection systems are far more restrictive. 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).

For some of the recent approaches to packet scanning techniques we refer the reader to [2, 6, 7] and the references therein.

2 The General Case of Two Alphabets

Our starting point is *bit-splitting* as described in [6] 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.

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 $w = a_1 b_1\ a_2 b_2 \cdots a_n b_n$ 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*

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

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

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 are the only models that can be implemented on real machines 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 as shown in Figure 1.

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.

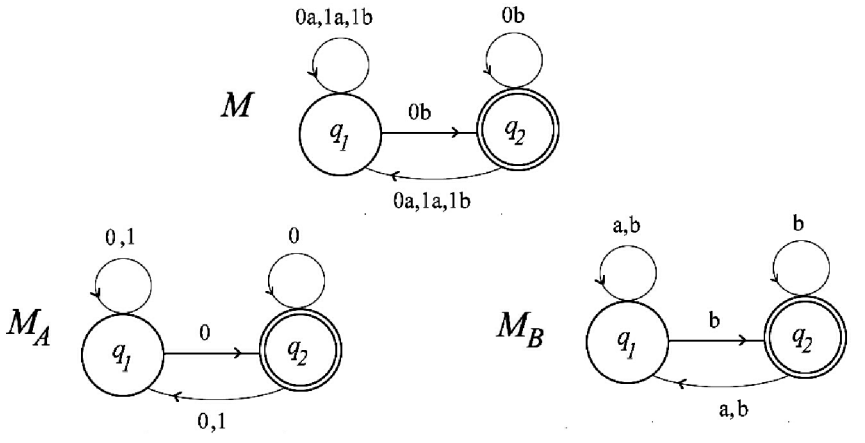

Fig. 1. M_A and M_B from M: $\Sigma = A \times B$ with $A = \{0, 1\}$, $B = \{a, b\}$

Lemma 1. *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 3. *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 1. *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 1, 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 $x_1b_1x_2b_2 \cdots x_nb_n \in \mathcal{L}$. 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 2. *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 Σ^*p where $\Sigma = A \times B$ and p is the keyword. With the notation of Definition 3,

$$a_1y_1\ a_2y_2 \cdots a_ny_n,\ x_1b_1\ x_2b_2 \cdots x_nb_n \in \mathcal{L}$$

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 Σ^*p, where the union is over the keywords p. However $\mathcal{L}$ in this generality need not satisfy the alternation property of Proposition 1.

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

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

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.

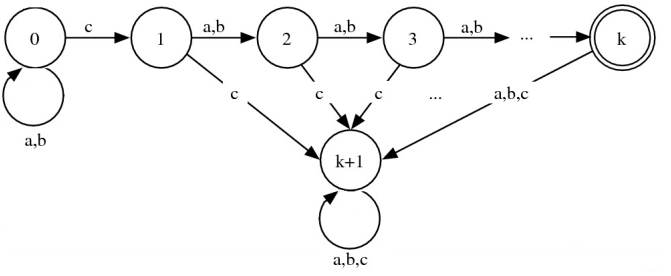

Fig. 2. DFA for a language whose homomorphic image requires $\Omega(2^k)$ DFA states

2.1 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.

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 2. *Suppose M is a Aho-Corasick automaton on n states over the alphabet $\Sigma = A \times B$ 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 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. □

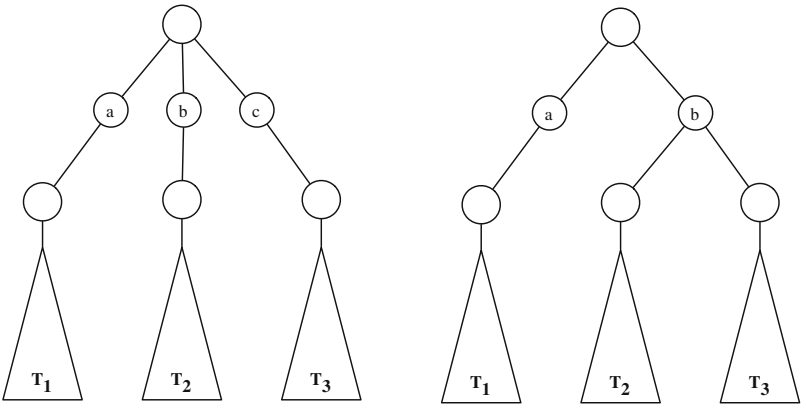

Fig. 3. Identification of b and c: at the root of the trie

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.

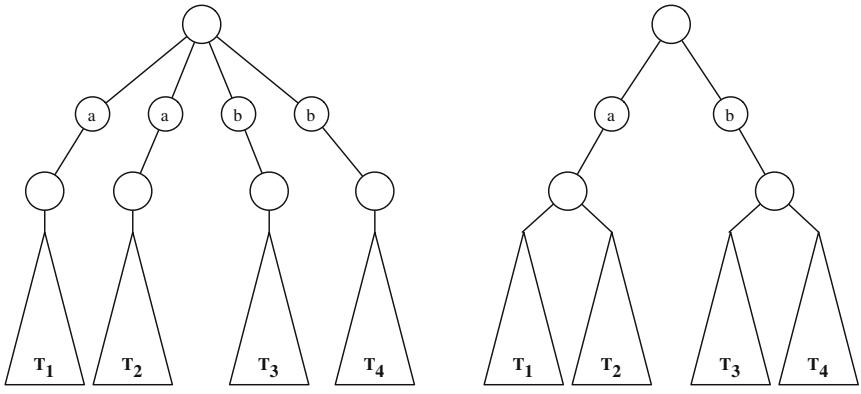

Fig. 4. Identification of b and c: at an arbitrary node of the trie

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

2.2 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 3, $\mathcal{L}$ can be recovered completely from the knowledge of $\mathcal{L}_A$ and $\mathcal{L}_B$. Thus by Proposition 1 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 1 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 3. *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 2 is applicable. □

Remark: 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

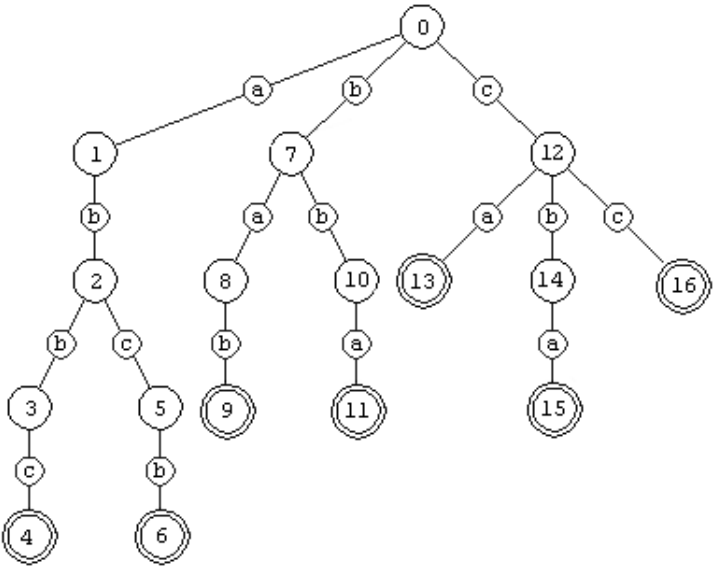

Fig. 5. Trie portion of the Aho-Corasick machine for the keywords $\{abbc, abcc, bab, bba, ca, cba, cc\}$ over $\Sigma = \{a, b, c\}$

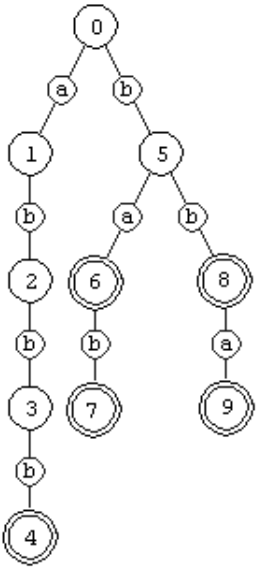

Fig. 6. After identifying c and b, the resulting trie of the Aho-Corasick machine for the keywords $\{abbb, bab, bba, ba, bb\}$ over $\Sigma = \{a, b\}$

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.

Remark: If we use the deterministic versions of M_A and M_B obtained by the algorithm described in the proof of Proposition 2 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.

Remark: 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.

3 Conclusions

We proved that bit-splitting Aho-Corasick machines is functionally correct, and provided strict space bounds for this approach. The formal description of how and why bit-splitting works opens the door to new potential applications for other classes of languages in similar problem domains. Future work could address a formal framework for bit-splitting to search for patterns embedded with single character wildcards.

References

[1] Aho, A.V., Corasick, M.J.: Efficient String Matching: An Aid to Bibliographic Search. Comm. of the ACM 18(6), 333–340 (1975)
[2] Baker, Z.K., Prasanna, V.K.: High-throughput Linked-Pattern Matching for Intrusion Detection Systems. In: Proc. of the First Annual ACM Sym. on Arch. for Networking and Comm. Systems (2005)
[3] Eckmann, S.T., Vigna, G., Kemmerer, R.A.: STATL: An Attack Language for State-Based Intrusion Detection. J. of Computer Security 10(1/2), 71–104 (2002)
[4] Newsome, J., Karp, B., Song, D.X.: Polygraph: Automatically Generating Signatures for Polymorphic Worms. In: IEEE Sym. on Security and Privacy, pp. 226–241 (2005)
[5] Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proc. of LISA 1999: 13th Systems Adm. Conf., November 1999, pp. 229–238 (1999)
[6] Tan, L., Sherwood, T.: A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In: ISCA 2005: Proc. of the 32nd Annual Int. Sym. on Computer Architecture, pp. 112–122 (2005)
[7] Tuck, N., Sherwood, T., Calder, B., Varghese, G.: Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. In: The 23rd Conf. of the IEEE Comm. Society (Infocomm) (2004)