Technical University of Crete

Department of Electronic and Computer Engineering

Microprocessor and Hardware Laboratory

Master Thesis

A Memory Efficient Intrusion Detection Architecture Using the Split-AC Pattern Matching Algorithm

Author
Vasilis Dimopoulos

Supervisor
Associate Professor Dionisios Pnevmatikatos

September 2006

Table of Contents

CHAPTER 1 – INTRODUCTION

In these past few years, an information revolution is taking place as the entire world has become connected into one global network, the Internet. People are able to communicate and cooperate with other people or access resources that may be located on the other side of the globe. Transactions over the Internet have become a steadily increasing part of global economy, while more and more Internet-only companies are emerging.

Of course, the Internet is not just about business. Academic work and research has benefited significantly, since the Internet can also serve as an enormous library where any information one is looking for can be accessed with the press of a button. Last but not least, the Internet also serves as a radical new method of entertainment: movies, music and literature are now available worldwide, while computer games such as Massively Multiplayer Online Role Playing Games, or MMORPGs, have created online gaming communities with tens of thousands or more members.

The only thing needed to access this huge world of information is a computer and an internet connection, both of which are becoming more widely available as time goes by.

However, this global availability of information raises considerable security issues. Malicious programs, or "viruses", can be easily transmitted from one computer to the next and can cause irregular operation of executing programs or deterioration of the operating system. Some types of viruses, such as "spyware" or "worms", may transmit personal information of a user or can leave a back-door open through which another user can access someone's computer. There are also many occasions where people with advanced programming skills were able to infiltrate secure networks and cause significant damage or steal corporate secrets.

In order to provide the necessary security, several methods have been developed. One of these methods is the "firewall", which can be either a software or a hardware platform. The firewall examines the headers of the packets being transmitted and received and allows only packets satisfying certain criteria, for example packets originating from a known secure network, to pass. Another widespread method of protection is the "antivirus" programs, which continuously check a computer and remove any known viruses that may have infected it.

A more recent manner of protection, which is not yet as widespread as the above two, is the "Intrusion Detection Systems", or IDS. Essentially a more evolved version of the firewall idea, an IDS examines not only the header of a packet, it also scans the payload in order to find specific patterns that constitute part of an attack or otherwise irregular behavior. Currently, Snort is the most widespread IDS and, due to the fact that it is open-source and freely distributed, is also the point of reference for all IDS-related work.

Although powerful in their capabilities, IDS can be rather computationally expensive because they rely on pattern matching algorithms to scan the entire payload of a packet and discover suspicious patterns hidden inside. A common type of attack used against intrusion detection systems, called Denial of Service (DoS), is to flood them with packets containing many suspicious patterns. In this case, the IDS must process a large number of packets and the processing of each packet takes time since many patterns are matched; if the packets arrive faster than the IDS can process them,

the IDS buffers eventually fill up and subsequent packets are either dropped, which means possible loss useful data, or allowed to pass through unchecked, which means that attacks can enter the system. This means that the pattern matching algorithms must be as fast as possible and perform in such a way that their throughput doesn't deteriorate when under attack traffic conditions.

The need for fast intrusion detection systems is made even more pressing by the hardware improvements that are rapidly increasing transmission speeds, which have currently reached multiple Gigabits per second. Since IDS are usually software based, the rate at which their processing speed increases is much smaller than that of the transmitting hardware.

Effective IDS implementations are software-based, because the hardware required to support the entire functionality of an IDS would be extremely large and complicated. However, a software-based IDS could utilize small amounts of hardware to perform certain tasks, such as pattern matching, which can be particularly expensive when performed in software.

The contribution of our work towards improving Intrusion Detection Systems is two-fold. First, we present a variation of the Aho-Corasick pattern matching algorithm, which we call "Split-AC", that is particularly suitable for hardware based IDS pattern matching and requires very low amounts of memory. There has been significant research on Aho-Corasick based algorithms, and our motivation was to develop a variation that would require the smallest possible amount of memory. Our second and last contribution is a simple IDS architecture that can fit on a single FPGA chip, performs packet header classification and pattern matching and can cooperate with a software-based IDS in order to significantly reduce the amount of work performed by the software.

The rest of this thesis is organized in the following manner. Chapter 2 provides a background in pattern matching by presenting some of the most widespread and effective algorithms. In chapter 3 we present the memory-efficient Split-AC algorithm as well as the revisions we made in order to improve some of the algorithm's aspects. Chapter 4 describes the rules being used, i.e. the manner in which suspicious network traffic is described, as well as the manner in which these rules are grouped into sets and subsets. Chapter 5 presents a detailed architecture for a hardware IDS implementation which can cooperate with software-based IDS and uses the Split-AC pattern matching algorithm, as well as the challenge that we faced trying to implement the IDS architecture in FPGAs and the manner in which the necessary validation was carried out. Finally, chapter 6 provides the detailed results for both the Split-AC algorithm and the FPGA implementation of the IDS architecture.

CHAPTER 2 – PATTERN MATCHING ALGORITHMS

The topic of pattern matching has been the focus of significant research efforts in the past 30 or so years, with scientific areas such as bioinformatics, database searching and, of course, Intrusion Detection Systems relying heavily on the performance of the underlying pattern matching algorithms. Specifically, Intrusion Detection Systems use pattern matching algorithms to uncover certain suspicious strings inside the payload of a packet that may constitute part of an attack against the system or otherwise unorthodox behavior. The continued proliferation of Intrusion Detection Systems has helped rekindle interest in this scientific topic.

In this chapter we will first present some of the most effective pattern matching algorithms their important variations. At the second part of this chapter we present recent work on memory-efficient variations of the Aho-Corasick algorithm, since our work is also based on the Aho-Corasick algorithm.

2.1 The Boyer-Moore Algorithm.

First published in 1977 [1], the Boyer-Moore (BM) algorithm still remains the most efficient method for locating a single pattern with a best- and average-case search cost of O(*N*/*m*), where *N* is the length of the text being examined and *m* the length of the pattern. The worst case search cost of the algorithm is O(*N***m*). Practically, this algorithm performs best for large patterns and alphabet size.

The first step in this algorithm is the construction of a lookup table, the *bad-character shift* table, which has a size equal to the size of the used alphabet (e.g. 256 entries for byte matching). For every possible input character, the table holds the distance of the *rightmost* instance of that character from the end of the pattern or *m* if the character doesn't appear in the pattern. For example, the shift table for the pattern "*banana*" is shown in table 2.1.

TABLE 2.1
*The bad character shift table for the pattern "banana". The symbol * means any other character.*

Character	a	b	n	*
Value	2	5	1	6

The algorithm also uses another table, the *good suffix shift* table. This table holds *m+1* entries, with each entry *j* indicating where we can find a reoccurrence of the *m-j* suffix characters.

The Boyer-Moore algorithm aligns the pattern with the start of the text, but always tries to match it from right to left. As long as text and pattern character matches, we keep proceeding towards the start of the pattern. Whenever a text character *c* is found that doesn't match the corresponding (*m-j*)-th pattern character, the entire pattern is shifted right by an amount of max(GS[*j*], BC[*c*] – *j*), where GS[*j*]

is the *good suffix shift* table value for the position of the mismatch and BC[*c*] the *bad character shift* value corresponding to the specific text character.

2.1.1 Horspool's Variation.

A simplification of the original Boyer-Moore algorithm [2], this algorithm goes to show that a simpler but "dumber" algorithm can be faster than a complex and "smarter" algorithm. Horspool showed that using only the *bad character shift* table is usually faster for large alphabets (such as the ASCII table) and non-periodic patterns.

Horspool's variation has a best- and average-case cost of O(*N/m*) and a worst-case cost of O(*N*m*), much like the original Boyer-Moore algorithm, The essential difference between the two is that Horspool performs more character comparisons but each comparison is faster than in Boyer-Moore.

2.1.2 Adapting Boyer-Moore to Multiple Pattern Matching.

The Boyer-Moore algorithm can be adapted to simultaneously search for a set of strings, as shown in [3], by inverting some of the original ideas. The patterns are combined in a suffix-tree structure, which initially aligned with the *end* of the text, the patterns are matched from *left to right* and in case of a mismatch the tree is *shifted to the left*.

In this algorithm, the *bad character shift* table recommends a shift based on the distance from the start of the pattern of the *leftmost* instance of a character *c* in any of the patterns (instead of the *rightmost* appearance, as was the case with the traditional algorithm).

On the other hand, instead of the traditional *good suffix shift* table, this algorithm uses a *good prefix shift* table, which recommends a shift to the next occurrence of a complete prefix that has already been seen as a substring of another pattern, or shift to the next occurrence of some prefix of the correctly matched text as the suffix of another pattern in the set.

This algorithm has a best- and average-case search cost of O(*N/m'*), where *m'* is the length of the *smallest* pattern in the set. The fact that no shift can be larger than the length of the smallest pattern is one of the main drawbacks of this approach. Another drawback is that for a large group of patterns with reasonably varied contents, the bad character shift table (which is the main performance-giving mechanism) becomes saturated with very small values, causing the algorithm to have super-linear search cost.

2.2 The Aho-Corasick Algorithm

When concerned with the conditions under which a pattern matching algorithm is preferable, the Aho-Corasick (AC) algorithm lies on the opposite end of the spectrum from both Boyer-Moore and Wu-Manber. The AC algorithm checks

each text character only once regardless of how many or how small patterns we are trying to match, making it ideal for sets with very small patterns.

Since it was first proposed [4] in 1975, this algorithm has remained one of the standard methods of pattern matching, especially for very large groups of patterns (tens of thousands). During the initial phase of the algorithm, all patterns are combined in a syntax tree structure, in which every edge represents a character, every node has a unique number id, and the route from the initial node to another represents a certain pattern (or sub-pattern). Patterns that share a common prefix also share a number of nodes equal to the length of the prefix. The nodes coinciding with the end of a pattern are called "final" nodes.

After all the patterns have been added the syntax tree, specific algorithms are used to convert that tree first into a non-deterministic automaton and, finally, into a deterministic automaton or finite state machine. The syntax tree and derived deterministic automaton corresponding to the pattern set {"bat", "batch", "cat"} is shown in figure 2.1. Note that during the conversion from syntax tree to finite state machine, the number of "final" states (not the number of total nodes) may increase. Such an effect does not occur in figure 2.1 due to the simplicity of the example.

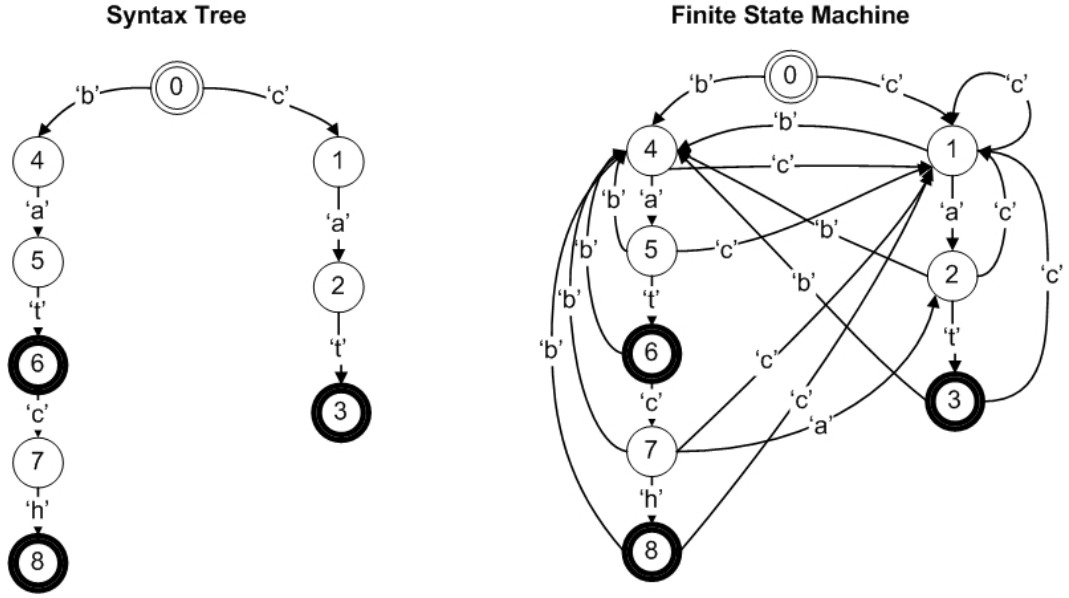

Fig. 2.1: The Aho-Corasick syntax tree and the derived FSM for matching the patterns "bat", "batch" and "cat". Each node represents a state. The arrows represent transitions for specific characters. Nodes with a bold outline represent "final" states.

The finite state machine shown in figure 2.1 is slightly simplified. In reality, there should be arrows from each node leading back to the initial state, state 0, for all characters not already present in transitions from that node. For example, node 8 has defined transitions for characters 'b' and 'c'; for all other characters the transition is back to the initial state.

After the finite state machine has been generated, the pattern matching process is extremely simple: the fsm starts at the initial state and begins processing the characters of the text one after the other. For each input character the fsm performs exactly one transition. Whenever the fsm arrives at a "final" state then a pattern has been successfully matched.

The non-deterministic automaton can also be used to match the contained patterns. However, in that case the automaton may execute up to 2*N* transitions. The other difference between the non-deterministic and the deterministic automaton is that the non-deterministic usually has significantly less transitions per node, a fact that has been exploited in several memory-efficient variations of the Aho-Corasick algorithm.

The strongest point of the Aho-Corasick algorithm can also be its major weakness: the fact that the number of character comparisons is precisely *N*. When searching for a set of long patterns using another algorithm such as multi-pattern Boyer Moore or Wu-Manber (description follows), the average case lookup cost is roughly O(*N/m*) : for a large value of *m*, these algorithms perform only a fraction of the character comparisons that AC does. However, the tables turn if there is even one small pattern (one or two characters long) in the set. In that case, the other algorithms will have super-linear search cost most of the time. For a minimum pattern length of 2 or 3 characters, the Wu-Manber algorithm will *always* have super-linear cost.

Another drawback of the Aho-Corasick algorithm is its extremely large, by comparison to the other methods, memory needs. Fast implementations of the algorithm hold at each node an array of 256 pointers to next states, one for each possible character input. For software implementations of the algorithm using 4 bytes per pointer this translates to 1Kbyte per node. The large memory requirements of the algorithm make hardware implementations rather challenging.

2.3 The Wu-Manber Algorithm

A more recent entry to the field of pattern matching, the Wu-Manber [5] pattern matching algorithm was first published in 1994 and is currently considered one of the most efficient pattern set matching algorithm, with a best-case search cost of O(*N*/*m*).

The Wu-Manber algorithm (WM), which also provides the pattern matching mechanism for the Snort IDS, could be considered a radical evolution of the Boyer-Moore algorithm since it also relies on bad character shift mechanism to avoid certain checking parts of the text. Specifically, a *shift* table holding 2^k entries, instead of the BM's 256 entries, is used. The parameter *k* is the size in bits of the generated hashing index that will be explained shortly.

The contents of the shift table are initially set to *m-B+1*, where *m* is the length of the shortest pattern in the set and *B* is the number of characters that are hashed together. Afterwards, starting from the first character and proceeding to position *m-B* one character at a time, the *B* sequential characters are hashed together (using the same hash function every time), producing a *k*-bit wide value *h*. The shift table is then updated so that the value for index *h* is equal to the minimum between the previous shift table contents and the distance of the last one of the *B* hashed characters from the *m*-th position. In essence, the shift table holds the minimum distance of the rightmost appearance of every possible hashing of *B* sequential pattern characters value from the *m*-th position. For example, the shift table for the pattern "banana" and for B=2 is shown it table 2.2.

TABLE 2.2
*The Shift table of the Wu-Manber algorithm for the pattern "banana". The * character means any other.*

Hash Index	ba	an	na	*
Shift amount	4	1	0	5

Apart from the *shift* table, the algorithm uses two more tables: the *hash* table and the *prefix* table. If *shift*[*h*]=0, then *hash*[*h*] holds a list of patterns whose *B* "last" characters (i.e. characters in positions *m-B* to *m*) hash into value *h*, while *prefix*[*h*] holds a list of hash values, one for each pattern, of the *B'* prefix characters of the pattern.

The search procedure of the Wu-Manber algorithm is the following: starting from position *m* in the text, the *B* previous characters are hashed together and a value *h* is generated. If *shift*[*h*] is non-zero, then we proceed to the right by *shift*[*h*] characters and repeat the process. If *shift*[*h*] is zero, then the *B'* text characters in the positions where the prefixes of the patterns would be are hashed together and form *h'*. *h'* is then compared with the contents of the *prefix*[*h*] list. If any of the prefix[h] values match *h'*, then the corresponding pattern from *hash*[*h*] is selected and compared against the text one character at a time.

In general, the Wu-Manber algorithm has proven to be very effective for large pattern sets when the minimum pattern length is not very short (i.e at least 5 characters). The fact that the algorithm is parameterized so that a user can select the *B*, *B'* and *k* values that best fit his resources and the problem at hand is another significant strength of the algorithm. The algorithm major weakness is that the performance relies heavily on the length of the minimum pattern.

2.4 Related Work: Memory Efficient Aho-Corasick Variations

In this section we present related work on memory-efficient variations of the Aho-Corasick algorithm. We focus on Aho-Corasick variations and not memory-efficient algorithms in general because the algorithm we developed and which is presented in Chapter 3 is also a variation of the Aho-Corasick algorithm.

2.4.1 Snort's Banded-Row Aho-Corasick

The first memory efficient variation we examine [6] comes directly from one of the people behind Snort. The proposed variation supports two different state types: states with a full, 256-entry lookup table and states with banded-row format lookup tables. Typically, the full lookup table is used in states located near the initial state of the fsm, since they tend to have a large number of transitions, while the banded-row format is used in states located further away, since they tend to have a sparse lookup table.

Typically, the lookup table for a state of the Aho-Corasick automaton is an array of 256 values. In this array, a non-zero value signifies a valid transition to a specific state, while zero signifies a transition back to the initial state.

A lookup table using banded-row format holds all the elements located between the first and the last non-zero contents of the lookup table, the distance between the first and last non-zero elements and the position of the first non-zero element.

A simple example of a state lookup table and the corresponding banded-row format is the following:

Original Lookup table	: 0 0 0 2 4 0 0 0 7 0 6 0 0 0 0 0 0 0 0
Number of elements	: 8
First non-zero position	: 4
Banded-row table	: 8 4 2 4 0 0 0 7 0 6

The lookup process for banded-row format is the following. When an input character *c* arrives and the fsm is in a state with a banded-row lookup table, the character is first checked if it is within the banded area. If *c* lies before the first non-zero transition or greater is than the sum of the first position and the number of elements (thus failing after the last element), then the transition is 0 and the fsm returns to the initial state. Otherwise, the first non-zero position is subtracted from c and the resulting value is used to directly access the banded-row lookup table.

The banded-row format provides substantial memory compression when the lookup table is sparse (i.e. has many 0 values) and the non-zero values are located close together. The disadvantage of this method is that the performance, memory-wise, depends heavily on the position of the non-zero values. For example, a 256 entry lookup table in which only the first and last entries were non-zero would receive no benefits from this approach.

2.4.2 Bitmap Compression and Path Compression

Both techniques are presented in [7] and were based mainly upon the Eatherton tree-bitmap algorithm [8] which is widely used for IP routing. Also, both techniques have been designed for the non-deterministic Aho-Corasick automaton because it has significantly less transitions per state than the deterministic automaton does.

The *Bitmap Compression* method is applied in all fsm states and uses a 256 entry bitmap and an array of non-zero transition pointers instead of the typical, 256 pointer state lookup table. A bit *j* of the bitmap is set if the fsm has a valid transition for input character *j*. The array of pointers holds sequentially all the non-zero transitions. A small example of the resulting bitmap and transition array is the following:

Original Lookup table	: 0 0 0 2 4 0 0 0 7 0 6 0 0 0 0 0 0 0 0
Bitmap	: 0 0 0 1 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0
Transition Array	: 2 4 6 7

When looking up the next state transition for a specific character *c*, we first check if bit *j* is set. If the bit is cleared then the non-deterministic automaton performs a failure transition. However, when the bit is set a popcount operation is performed on the bitmap up to position *c* (i.e. we count all the '1' bits appearing in the bitmap before bit *c*). The result of the popcount is then used to address the transition array, which returns the appropriate next state

On the other hand, the *Path Compression* technique can not be applied to all fsm states. It applies only to the states from which a chain of states with only one transition per state begins (as occurs for unique pattern suffixes inside the Aho-Corasick non-deterministic finite automaton). For example, there is a state A inside the automaton from which only the following chain of transitions begins:

$$A \rightarrow B \rightarrow C \rightarrow D.$$

Instead of allocating space for four nodes, only node A is created containing the additional information of the suffix that needs to be matched in order to perform the → B → C → D transitions. In this way, whenever the automaton reaches state A we simply check if the following text characters match the specific suffix.

These two methods can reduce the required memory significantly, but have other major disadvantages. First of all, both have been designed for use with the non-deterministic AC automaton which can execute up to 2*N* state transition for an *N*-sized text. The *Bitmap Compression* method can be adapted to deterministic automatons, in which case it will offer diminished memory gains, but the *Path Compression* method cannot be utilized in deterministic fsms.

The second major disadvantage concerns the popcount operation necessary for the *Bitmap Compression* technique. The hardware required for a 256-wide popcount would be a significant cost due to the operation's complexity. An easier way of implementing the popcount is to maintain running sums every *k* bits of the bitmap and perform only a *k*-bit wide popcount (which can easily be performed by a 2^k sized lookup table holding all possible popcount results). However this would come with an increase in memory requirements.

2.4.3 Bit-Split Finite State Machines

A rather novel and particularly promising approach to the Aho-Corasick algorithm was proposed in [9].

In a nutshell, the idea is that instead of having one large fsm processing an 8-bit character with 256 possible transitions, you have 8 small bit-split fsms, each of which has 2 possible transitions per state and processes a different bit of the character. Each bit-split fsm state also contains a *g*-bit wide partial-match vector, where *g* is the number of patterns in the original fsm, which indicates the patterns that are partially matched in this state of the bit-split fsm.

Figure 2.2[9] shows an example of an Aho-Corasick fsm for matching the pattern set { "he", "she", "his", "hers"} and the two derived bit-split fsms B_3 and B_4, which process the input character's bits 3 and 4 respectively.

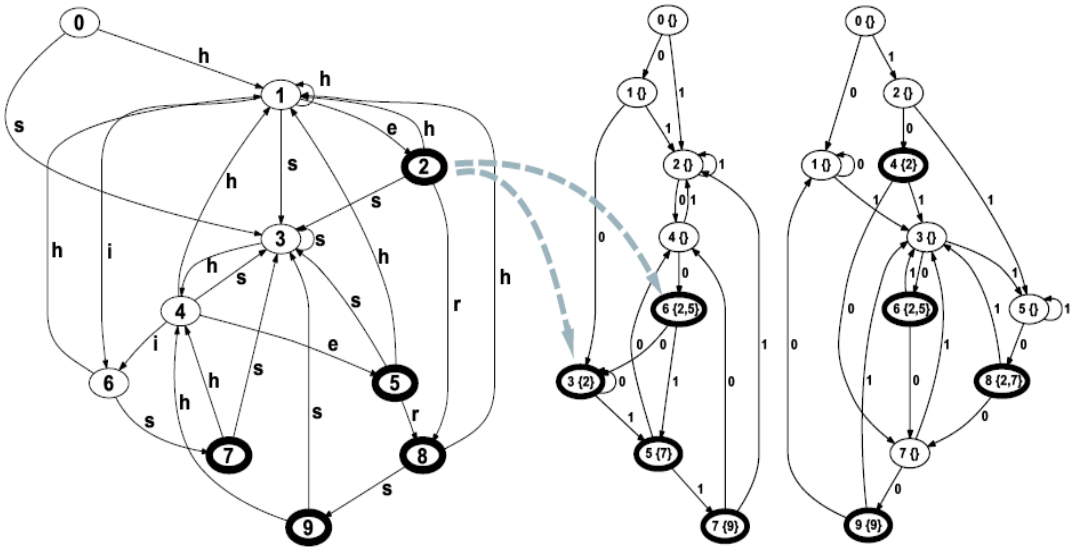

fig. 2.2 [9]. *The initial Aho-Corasick fsm (left) and the resulting bit-split fsms handling input bits 3(center) and 4 (right). States with a bold outline represent "final" states*

A bit-split fsm B_i is constructed from the initial Aho-Corasick fsm, called *D*, based on the following procedure. Starting from the initial state 0 of *D*, state 0 of B_i is constructed having a state set of {0}. The numbers in a bit-split state set correspond to *D* states. For all the states contained in the state set of B_i's State 0, we find which *D* states can be reached when bit *i* of the input character is '0': these *D* states form the state set of a new B_i state. The same process is repeated for the *D* states that can be reached when bit *i* of the character is '1', resulting in another new B_i state. In the example of figure 2.2, when starting from *D* state 0 then *D* states 0 and 1 are reached for a '0' value of bit 3, while D states 0 and 3 are reached for a '1' value of bit 3. This process is repeated until the next states for all B_i states have been constructed.

When processing an input text, all 8 bit-split fsms perform independent transitions based on their corresponding input bit values. After the transition, each small fsm outputs the partial match vector of the state that it is currently located. The partial match vectors are AND-ed together and the resulting final match vector will have 1 at a certain location only if all 8 bit-split fsms have located the corresponding pattern.

The standard Aho-Corasick fsms aren't necessarily split into 8 bit-split fsms. The authors describe how, depending on the situation, it can be more effective to split it into 4 bit-split fsms with each fsm handling 2 character bits and each state having 4 possible transitions, or into 2 bit-split fsms with each one processing 4 character bits and each state having 8 possible transitions

The authors show that their algorithm is very effective in terms of both performance and memory requirements. They also present an architecture for hardware implementation of the algorithm that can be updated at runtime whenever the rule set changes. All in all, this algorithm seems to have no discernible major disadvantages.

2.5 Summary

In this chapter we have presented several of the most important pattern matching algorithms currently used. On one hand we have algorithms such as Boyer-Moore or Wu-Manber which take advantage of specific information from the pattern characters to skip over portions of the examined text. These algorithms usually have sublinear lookup cost, but their performance is restricted by the length of the shortest pattern in the set: a pattern of one or two characters can cause these algorithms to have superlinear cost. Since we frequently have patterns that are 1 to 3 characters long in Intrusion Detection Systems, these algorithms are not particularly suitable.

On the other hand, we have the Aho-Corasick algorithm. The idea behind this algorithm is to combine all the patterns into one large finite state machine and then examine the characters of the text one after the other. This algorithm is very powerful due to the fact that the cost to examine a text is always linear, regardless of the number of patterns or of their lengths. However, the major disadvantage of the Aho-Corasick algorithm is that it requires very large amounts of memory.

The suitability of Aho-Corasick to Intrusion Detection Systems was the reason that inspired us to develop the Split-AC algorithm, presented in the next chapter, which is based on Aho-Corasick while requiring substantially less memory.

CHAPTER 3 – THE SPLIT-AC ALGORITHM

3.1 Introduction

This chapter is dedicated to the algorithm we developed for reducing the memory requirements of the Aho-Corasick pattern matching algorithm. We call this algorithm "Split-AC", which is short for Split-Aho-Corasick.

In the first part of this chapter we present the reasons for selecting the Aho-Corasick as the basis for our work as well. In the second part we present the ideas that led to the development of the Split-AC algorithm as well as a general description. The third and final part presents the original Split-AC algorithm in detail, as well as the two subsequent revisions that were made.

3.2 Why Use the Aho-Corasick Algorithm

There are several reasons behind our choice of the deterministic Aho-Corasick algorithm over others.

The first reason behind our choice has to do with the scientific area we are targeting: Intrusion Detection Systems. Snort rules, which are used as a reference point among all IDS-related research, specify patterns of various lengths. The problem is that several patterns are only one or two bytes long. The performance of "fast" pattern matching algorithms, i.e. algorithms that skip over sections of the text, such as Wu-Manber, is directly proportional to the length of the minimum signature. For a minimum pattern length of 3, a modest implementation of the Wu-Manber algorithm that hashes 2 characters per iteration would compare every text character *at least once*. The Aho-Corasick algorithm checks every text character only once.

Another reason is the relative simplicity of implementing the algorithm in hardware. A state lookup memory and a register holding the current state are enough: during each cycle the next character arrives, which we use along with the current state to address the lookup memory and find the next state. Algorithms that can move back and forth in the text present a larger complexity to implement.

Also, we want to combine accurate pattern matching with fine-grained packet header classification. The header classification and rule grouping processes described in section 4.2.2 result in the creation of nearly 400 different rule sets, each of which may require a pattern matching engine. This raises synchronization issues, i.e. knowing what position of the text each engine is examining. Since every Aho-Corasick fsm processes one character at a time, at any time all the fsms will be examining the same character of the text. This also solves the problem of maintaining and updating parts of the payload on-chip: the entire payload can be kept off-chip and fed to the fsms one character at a time. If an algorithm that text-skipping was used, there would be an additional difficulty of maintaining a payload segment for each instance of the algorithm.

3.2.1 Aho-Corasick Simplification Heuristic

During our experimentations with the Aho-Corasick algorithm, we found a simple way to slightly reduce its complexity. Traditionally, each state of the fsm has a list of patterns or some other information associated with itself in order to understand during the search phase if it is a "final" state or not. In hardware implementations, an extra bit would be needed for the state lookup contents that would indicate which states are final and which not, or otherwise a logic block that would check if the current state number belongs to a final state.

That need for additional information can be eliminated by a simple state renaming, in which all "final" states are assigned numbers from 1 to *F*, with *F* being the total number of "final" states. In this way, we know we are at a final state whenever the current state number is between 1 and *F* (inclusive). Note that this makes easier only to know *if* we are at a "final" state; if we want to know the patterns that have been matched we still need to check a pattern list.

We can utilize this information by having an abstraction layer responsible for monitoring and managing the fsms, as well as storing the information concerning which pattern corresponds to every final state. Whenever an fsm reaches a final state, it informs the abstraction layer of the final state that was reached and from there on the abstraction layer knows precisely which pattern was matched.

This heuristic has been utilized fully in every version of the split-AC algorithm.

3.3 Split-AC: General Description.

The Split-AC algorithm was based on several observations concerning the rule groups and the corresponding Aho-Corasick finite state machines that are generated from a fine-grained header classification.

Our first observation was that the fsms corresponding to most rule groups had valid transitions, i.e. transitions not leading back to the initial state, for only a fraction of all the possible input characters: many Snort rule patterns use only alphabet characters, arithmetic digits or some standard symbols(e.g. '\'). Also, a significant number of rule groups require Aho-Corasick fsms of only a few dozen states or less, which has a similar effect since the number of different characters appearing in transitions is always less than the number of states. This observation led us to search for a method to reduce the number of input bits by grouping together the characters that cause valid transitions. Reducing the number of character bits by *n* will reduce the size of the state lookup table by 2^n, since we will require, for example, only 32 or 64 transitions per state instead of the typical 256.

The second observation was that not all characters appear with the same frequency in transitions of an Aho-Corasick fsm. Some characters may appear in transitions from nearly every state, as is the case for the first character of every pattern, while others may appear transitions from only one or two states. In the example of figure 2.1, it is apparent that characters 'b' and 'c', which are initial pattern characters, appear in every state while character 'h' appears in only one transition. This observation led us to consider the possibility that it may be more efficient in terms of memory to store only the transitions of frequently appearing

characters in the state lookup table and use some other, possibly more resource expensive approach, like Content Addressable Memories, to discover the next state for specific (character, state) pairs.

Finally, there can be several characters that don't appear in even one transition in an Aho-Corasick fsm. Whenever such a character is observed, the fsm always returns to the initial state regardless of its previous state.

A simple diagram visualizing all the above is shown in figure 3.1.

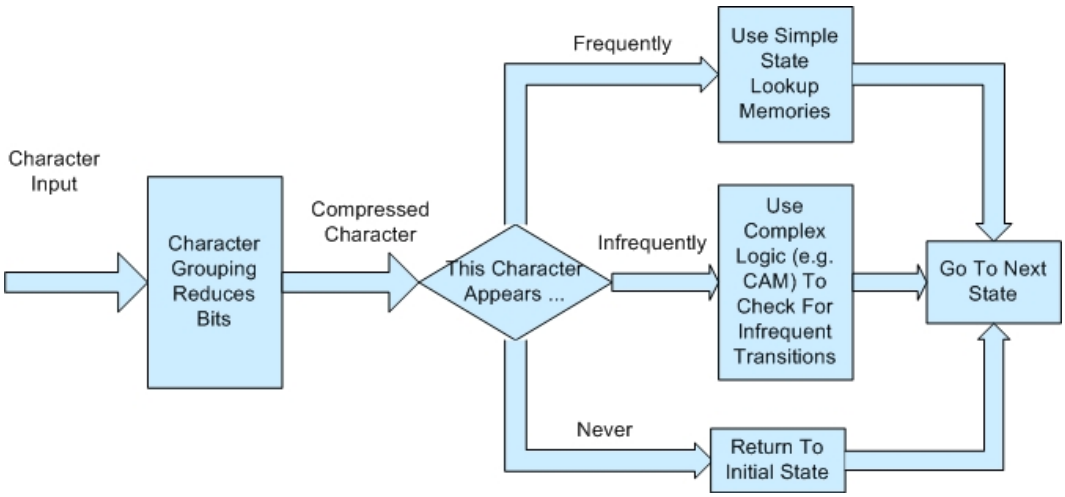

Fig. 3.1: An abstract diagram showing how the Split-AC algorithm operates.

3.3.1 Original Split-AC Algorithm

We start off with a typical Aho-Corasick deterministic fsm with S states. First of all, we define as $f(c)$ the occurrence frequency of character c in transitions of this fsm. The value for $f(c)$ is calculated by summing the number of states that have transitions for input character c and dividing the sum with S. Since the number of occurrences for any character c is between 0 and S, the value for $f(c)$ is $0 \leq f(c) \leq 1$.

We now define the frequency threshold as T_f. Any character c with an occurrence frequency $f(c)$ greater or equal than the threshold is considered a *frequent character*. Accordingly, valid (i.e. not leading back to the initial state) transitions the fsm makes for frequent characters are called *frequent transitions*. Characters with an occurrence frequency smaller than the frequency threshold but greater than zero are considered *infrequent characters* and valid transitions made for these characters are called *infrequent transitions*. Finally, characters with a zero occurrence frequency are considered *non-existent characters*. This splitting of the characters into three groups is the reason we named the algorithm "Split-AC".

We define the total number of frequent characters as K. In order to perform the character grouping and "compress" the character we use a simple lookup table of 256 entries, which is addressed by the input character. This table, which we call "character translation table", holds a value of '-1' for all infrequent and non-existent characters and a different value from 0 to K-1 for each one of the K frequent characters. The width of the character translation table is defined as $k = \log_2(K+1)$ bits, rounded up. The "+1" in this formula is due to the '-1' translation value for infrequent and non-existent characters, which must also be taken into account. We name the result of this

lookup table "translated character". A simple example of a character translation table is shown in figure 3.2.

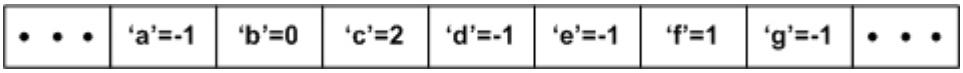

Fig. 3.2: A simple character translation table. The only frequent characters are 'b', 'c' and 'f'.

At this point we define the number of bits needed to represent the state of the fsm with $s=\log_2(S)$, rounded up. Whenever the input character is a frequent character we use a second lookup table, the "state lookup memory" to find the next state of the fsm. This table has $2^{(s+k)}$ entries and is s bits wide. This table is addressed by the current fsm state and the translated character: the state provides the s most significant address bits while the translated character provides the k least significant bits. In this manner, we can imagine the state lookup memory as being divided into 2^n blocks, one for each state, with each block holding 2^k transitions, one for each possible value of the translated character. A simple example of a state lookup memory is shown in figure 3.3. For each state the last transition will be empty because of the '-1' value (binary: "11…11") that is reserved for infrequent and non-existent characters.

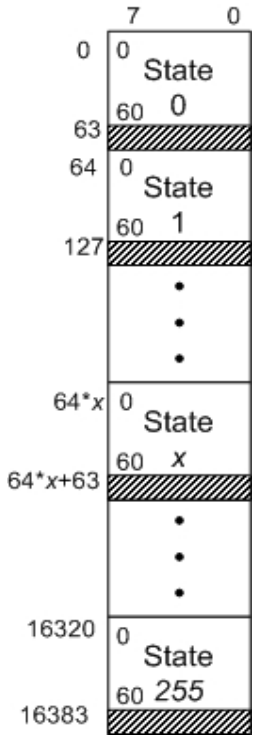

Fig. 3.3: An example of how the state lookup table is divided into different blocks. The fsm has 256 states and 61 frequent characters. The grayed-out segment in each state represents unused space.

The manner in which infrequent characters are processed depends mostly on the target platform for the algorithm. If the split-AC algorithm is to be implemented in software, the "infrequent" character transitions for each state would probably be maintained in a linked list that would be examined sequentially whenever an

"infrequent" character was observed. On the other hand, if we wanted to implement the algorithm in hardware a reasonable solution that offers high performance is to use a series of comparators or a Content Addressable Memory: in either case the current fsm state and the input character would be fed as inputs and the result would be the appropriate next state.

Taking all of the above into account, the lookup procedure for the Split-AC algorithm can be summarized in three simple steps.

Algorithm 3.1: *Text scanning procedure of the Split-AC algorithm*

Step 1. The next input character *c* is used to address the character translation table, which responds with the corresponding translated character *c'*.

Step 2. If the translated character *c'* is a "frequent" character, use the state lookup table to find the next state. Otherwise, the next state is found by checking the "infrequent" character transitions for input character *c*.

Step 3. Check if the next state is also a "final" state. If it is "final", a pattern was located. Return to Step 1.

3.3.2 The Frequency Threshold Parameter.

The only user defined parameter of the algorithm is the frequency threshold T_f. In this manner, a user can adapt the algorithm according to resources he has available. A low threshold value, close to 0, means that most characters are considered frequent. This in turn means that the character translation table has many non-zero entries, the translated character will possibly be more bits wide and the state lookup table will increase in size accordingly. The benefit of having a small threshold is that the effort or logic required for infrequent transitions is significantly smaller.

On the other hand, a large threshold value (close to 1), means that there are fewer frequent characters and the character translation and state lookup tables are much smaller. However, in this case a lot more logic or effort will be required for the infrequent characters

3.3.3 Maximum Utilization of Available Space

As we mentioned section 3.3.1, only the characters with an occurrence frequency equal or greater than the frequency threshold T_f are considered frequent. However, this first allocation may result in some space remaining unused. For example, consider the case where 10 frequent characters have been selected based on the threshold: the translated character is 4 bits wide and the state lookup table has allocated 16 frequent transitions per state, out of which only 10 are used and 1 is reserved. In this case, we could accommodate another 5 frequent character essentially for free.

We take this fact into account by performing a second frequent character selection. After the initial frequent character selection (based on the frequency threshold T_f) is completed and the k number of bits has been set, we calculate the number of unused transitions per state. Let that be X. We then select the X infrequent

characters with the largest occurrence frequencies and convert them into frequent characters.

3.3.4 Matching Patterns without Case Sensitivity

An issue with every pattern matching is if and how they support case insensitive patterns matches. By case insensitive we mean that no distinction is made between capital and small letters. For example, if are searching for pattern “foo” in a case insensitive manner, then the algorithm must also be able to match “Foo”, “FOO”, “fOo” etc.

When searching only for case insensitive patterns, the answer is simple: use a lookup table to convert each lowercase letter into the equivalent uppercase. The situation gets pretty when a pattern set contains both case sensitive and case insensitive patterns. The usual solution is to search for all patterns in a case *insensitive* manner and to reexamine the input text in a case *sensitive* manner whenever a case *sensitive* signature is discovered.

For our hardware implementation of the algorithm, we decided against adding support for concurrent sensitive/insensitive pattern searches. This decision was based on the significant increase of implementation complexity due to the fact that text portions would need to be reexamined on occasions. Instead, whenever case sensitive and case insensitive patterns are contained in the same pattern group we will separate them in two smaller groups, one containing only the sensitive patterns while the other contains the insensitive ones, and perform two concurrent searches using the corresponding fsms.

The Split-AC algorithm can easily support case insensitive patterns matches due to the character translation table it uses. First, all patterns in a case insensitive pattern set are converted to either uppercase or lowercase (we chose uppercase). Afterwards, whenever a character *C* is placed in the character translation table and receives a value *x*, the corresponding lowercase character *c* receives the same value *x*.

In the revisions of the Split-AC algorithm, this mechanism is adequate since both frequent and infrequent characters are stored in the character translation table. However, in the original Split-AC algorithm this by itself is not enough because the infrequent transition mechanism checks the *input* character, not the translated one; since all pattern characters are converted to uppercase, a lowercase character would be ignored. In answer to this problem, a case translation table must also be used to convert lowercase characters to uppercase.

3.4 Algorithm Revisions

We developed the Split-AC algorithm in order to implement it in hardware. For the hardware implementation we decided on using a Content Addressable Memory-like structure for storing and checking the infrequent character transitions. This CAM is addressed with the input character and the current fsm state and responds with the corresponding next state or with the initial state 0 if no such transition exists.

CAM structures are very powerful and convenient, but they are also very demanding in resources: if a CAM contains *X* data entries and the CAM input is *Y* bits wide, then during each cycle *every* data entry is compared against the input in order to find a match. This translates to *X*Y* bit comparisons. With this in mind, we can estimate the cost of a CAM as roughly proportionate to the sum of input times the number of data entries. The output bits of the CAM are not of equal importance, since they perform some work only when a CAM entry has been matched.

The number of data entries in the CAM obviously can't be reduced, since that would mean omitting some transitions. Instead, we focused on reducing the number of input bits. In revision 1 we show how we can reduce the number of character input bits for the CAM at the cost of increasing the character translation table size. Finally, revision 2 shows how the state input bits of the CAM can be reduced.

The two revisions are independent and complementary: either or both can be applied and the received benefits are additive, since they target different aspects of the problem. However, during the algorithm's development we had the idea for revision 2 after revision 1 had already been implement. As a result, whenever we refer to revision 2 we assume that revision 1 has already been applied.

3.4.1 Split-AC Revision 1: Translating infrequent characters.

In the original algorithm, the CAM receives the untranslated 8-bit character as input. However, it is quite possible that there will be only a handful of infrequent characters in the Split-AC fsm, which could be represented by a lot less than 8 bits. To that end, we tried translating the "infrequent" characters much like we translate the "frequent" ones.

In the original algorithm Split-AC described in section 3.3.1, a total of *K1* frequent characters are selected, for which the character translation table stores values from 0 to *K1*-1. Now, we define as *K2* the number of different infrequent characters. These characters are also stored in the character translation table and each one of the infrequent characters is assigned a value from 2^{c1} to $2^{c1} + K2$-1, where $c1=\log_2(K1)$ rounded up is the number of bits needed to represent the different frequent characters.

The width of the character translation table's result is now $k = \log_2(2^{c1} + K2)$ bits, rounded up, in order to represent both frequent and infrequent characters. Additionally, the value '-1' (i.e "11…11"), which was previously reserved for both infrequent and non-existent characters, is now reserved exclusively for non-existent characters. This value has an increased responsibility in this revision, because it informs us that there exists no transition, frequent or infrequent, for a character, which means that both the frequent character state lookup table and the infrequent character CAM must be ignored and the next fsm state will be 0.

For all frequent characters, the translated character value will be from 0 to *K1*-1 and, as such, will have the *k*-*c1* most significant bits equal to 0. Conversely, if any of the *k*'-*c1* most significant bits is 1 then the character is definitely not a frequent one. We take advantage of this fact by using only the *c1* least significant bits to address the state lookup table, as happened in the original algorithm, and ignoring the result of the table if any of the *k*-*c1* most significant bits is 1. Note: the number of character bits for the state lookup table is the same as in the original algorithm, the difference is that the translated character is now more bits wide

With regards to the CAM, there are two possibilities. The first possibility is if the number of infrequent characters is less or equal to the number of frequent characters. In this is true then only $c2=\log_2(K2)$ character input bits are required to distinguish among all the infrequent characters. Since the k-$c2$ most significant bits tell us that we have an frequent character when they are zero, we use the $c2$ least significant bits as character input in the CAM and ignore the result when all k-$c2$ most significant bits are 0.

The second possibility is if the number of infrequent characters is greater than the number of frequent characters. In this case, all $c2=k$ bits of the translated character are used for the character input of the CAM.

All of the above can be comprehended more easily with a couple of examples. First, let's consider a case with 63 frequent characters and 7 infrequent ones. The frequent characters are assigned numbers from "0000000" to "0111110" (=62), while the infrequent ones are assigned numbers from "1000000" (=64) to "1000110" (=70). Obviously, the state lookup table needs only the 6 least significant character bits if we make certain to ignore its result when the most significant bit is 0. On the other hand, the CAM requires only the 3 least significant character bits if we ignore its result when the most significant bit is 0, which would indicate a frequent character. The remaining bits, between bit 5 and bit 3, are not required since they are always 0 for infrequent characters.

Now let's consider a different example, where there are 15 frequent characters numbered "000000" to "001110"(=14) and 20 infrequent characters numbered "010000" (=16) to "100011"(=35). We can see that in this case k-$c1$=2, so if either of the 2 MSBs is 1 the translated character is infrequent. The CAM, on the other hand, requires all 7 bits for the character input since the number of infrequent characters is larger than the frequent ones.

We have shown how translating the infrequent characters can reduce the number of character bits needed by the CAM, especially in cases where only a small number of characters are infrequent. However, the tradeoff is that a larger character translation table, at least one bit wider, is required. Another minor tradeoff is that additional logic (i.e. a (k-$c1$)-bit OR gate) is necessary check the translated character's MSBs and decide where the next state will come from, the CAM or the state lookup table. In order to take this added complexity into account, we have amended step 2 of the lookup algorithm:

Algorithm 3.2: *Text scanning procedure for Revision 1 of the Split-AC algorithm*

Step 1. The next input character c is used to address the character translation table, which responds with the corresponding translated character c'.

Step 2. If c' is a "frequent" character, use the state lookup table to find the next state. If c' is an "infrequent" character, the next state is found by checking the "infrequent" character transitions. Otherwise, the next state is 0.

Step 3. Check if the next state is also a "final" state. If it is "final", a pattern was located. Return to Step 1.

3.4.2 Split-AC Revision 2: State reordering.

In our efforts to reduce the state input bits for the CAM, we frequently observed that only a small number of fsm states had infrequent transitions and would

use the CAM at some point or other. This led us to the idea reordering the states so that all states with infrequent transitions are grouped together and eliminating the common state prefix.

This state reordering is done in two stages. First, we reorder the final states. As we mentioned in section 3.2.1, all F final states are assigned numbers from 1 to F. These states are reordered internally, i.e. in the same F positions, but in such a manner that final states *without* infrequent transitions occupy position 1 to F-x, while final states with infrequent transitions are grouped to numbers F-x+1 to F, where x is the number of final states with infrequent transitions.

The second stage is the reordering of non-final states with infrequent transitions, which are now assigned numbers from F+1 to F+y, where y is the number of non-final states with infrequent transitions.

The result of the state reordering process is that all states with infrequent transitions are now located between positions F-x+1 and F+y, which share a common prefix of s-s' bits and a varying suffix of s' bits. By ignoring the s-s' common prefix bits between F-x+1 and F+y, we can use the remaining s' bits as a state input for the CAM, where s is the number of bits representing the fsm state.

However, the s-s' prefix bits are not completely ignored. They may be unnecessary for the CAM input, but they are necessary to control when we want to access the CAM. To assure correctness, the CAM must be accessed only when the s-s' prefix bits are equal to the common prefix of the states with infrequent transitions. For example, if an input character c is infrequent but the s-s' prefix bits have a different value from the common state prefix then we are certain that the CAM cannot give us a valid result since we aren't at a state that contains infrequent transitions.

A simple example of how this revision affects the algorithm is the following. We have an fsm with 60 states (s=6 bits) that has a total of 10 states, final and non-final, containing infrequent transitions. After reordering, these states are placed in sequential positions from "001010" (=10) to "010011" (=19). Obviously, the states with infrequent transitions have s-s'=1 bit common prefix and the CAM now requires only s'=5 state bits. In case of an infrequent input character, the CAM will be accessed only if the MSB of the current state is 0.

This algorithm modification reduces the size of the CAM input for "free", without increasing memory size like Revision 1. The only tradeoff is that additional logic (i.e. a (s-s')-bit AND gate with some inputs possibly inverted) is required to check the state prefix and control when the CAM is accessed.

The final version of the lookup algorithm, which takes both revisions into account, is the following:

Algorithm 3.3: *Text scanning procedure for Revision 2 of the Split-AC algorithm*

Step 1. The next input character c is used to address the character translation table, which responds with the corresponding translated character c'.

Step 2. If c' is a "frequent" character, use the state lookup table to find the next state. If c' is an "infrequent" character and the current state contains infrequent transitions, the next state is found by checking the "infrequent" character transitions. Otherwise, the next state is 0.

Step 3. Check if the next state is also a "final" state. If it is "final", a pattern was located. Return to Step 1.

CHAPTER 4 – RULES, RULE SETS AND RULE SUBSETS

In the first part of this chapter, we describe the rule format that Snort uses, which is also the supported rule format for our application. The second part of the chapter describes the manner in which rules are classified into groups in Snort and in the IDS we implement. Finally, the third part of the chapter describes our motivation for dividing the initial rule sets into smaller subsets, as well as the manner in which that division is performed.

4.1 Snort Rules

Rules are used to specify what is considered as suspicious or dangerous traffic. We use the same language as the Snort NIDS [11],[12],[13],[14] for writing rules. A typical rule would something like:

```
alert tcp 192.168.1.0/24 any -> $EXTERNAL_NET 100:200 (content: “foo”; sid:100;)
```

Each rule is composed of two distinct parts. The first seven parameters, up to the opening parenthesis, form the *rule header*. The purpose of the header is to specify what types of packets require more detailed inspection as well as the action to be taken if the specific rule is activated. The rule header parameters are (from left to right).

1) **Action**: specifies what manner of action will be taken if the rule is activated. The action parameter is almost always “alert”, which means an alert message is generated and the packet is logged, but it can also be “log” (logs the packet for processing at a later time), “pass” (ignores the packet), “activate” (generates alert for this packet and turns on another “dynamic”) or “dynamic” (remains idle until activated by an “activate” rule and then acts as a “log” rule).
2) **Protocol**: This parameter can be either “tcp”, “udp”, “icmp” or “ip”. If the value is “ip” then we are interested in protocols other than TCP, UDP and ICMP.
3) **Source IP**: source IP address for the packet. The value for this parameter can be “any”, which means that any source IP address is acceptable, a specific IP address (e.g. 192.168.1.20), a subnet address (e.g. 192.168.1.128/25), a list of addresses which is indicated by brackets and uses ‘;’ between the addresses to separate them (e.g. [10.20.30.40;50.60.70.0/24]) or a symbolic address which is specified by a $ (e.g. $HOME_NET). Aside from “any”, all the above values can be negated with a ‘!’ at the beginning, which signifies that we want anything except this specific address or subnet (for example !192.168.1.128/25 means all addresses below 192.168.1.128 or above 192.168.1.255).
4) **Source Port**: useful only for TCP and UDP rules (IP and ICMP rules always have “any” in the specific field. Aside from “any”, the value can be a specific port (e.g. 80), a port range (e.g. 100:200) or a symbolic port (e.g. $HTTP_PORTS). All values except “any” can be negated by use of the ‘!’ symbol (e.g. !100:200 means

ports below 100 or above 200). When indicating port ranges one end of the range can be left blank, in which case the port range translates to "equal or greater" or "equal or lesser" (e.g. :80 means ports less than or equal to 80).

5) **Rule Directionality**: a '->' value indicates a straightforward rule in which the parameters to the left of the arrow are the source parameters while those to the right are the destination parameters. The other possible value is '<>' which indicates a bidirectional rule in which the source and destination values are interchangeable.
6) **Destination IP**: similar use as the source IP.
7) **Destination Port:** similar use as the source port**.**

The second half of each rule is the *rule options*, which are located inside the parentheses and specify what attributes of the packet are considered suspicious.

Whenever an incoming packet satisfies the criteria of a rule header, the corresponding rule options are checked. If *all* specified options are verified, then the entire rule has been matched and appropriate action must be taken. Rule options offer a large variety of checks that can be performed and are fully described in [14]. However, for the purpose of this work we are interested only in the following options:

- sid: unique rule id number.
- content: specifies a string to be located inside the payload. If the rule contains more than one "content" option then only the first interests us.
- nocase: the string to be matched is case insensitive
- ip_proto: the protocol number of the packet (for IP rules)
- itype: ICMP type (for ICMP rules only)

4.2 Rule Grouping

Checking every single rule against an incoming packet independently is a very expensive process when you have thousands of rules; thousands of header parameter comparisons are required to find which rules can apply to the packet and each rule could trigger a payload scan for a certain pattern. On the other hand, scanning the payload of a packet for *every* pattern in *every* rule may require only one pass of the payload, it also requires significant time to sort through the results and discard the inconsequential (i.e. not applicable to this packet) pattern matches.

In order to strike a balance between the two extremes described above, rules that have the same values for certain header parameters are grouped into *rule sets*. Every incoming packet must now go through a header classifications process in order to discover which rule groups can be applied. Instances of all patterns contained in a rule set are later discovered by performing a single pass (for each activated rule group) of the payload by using a multiple pattern matching algorithm.

We define the following four criteria that a *perfect* rule grouping method and the corresponding packet header classification process must satisfy:

- **Distinct**: The derived rule sets are formed in such a way that at most one rule set will match every incoming packet
- **Complete**: The rule group or groups that are selected for an incoming packet contain all the rules that could be matched by this packet.

- **Minimal**: Rule groups should not contain rules that will be content matched and later discarded due to some other header parameter.
- **Fast**: Only a very short, fixed period of time is required to select the matching rule set.

Of course, like most things perfect, such a header-based classification of rules into groups has yet to be implemented. Instead, we will show how Snort groups its rules into sets as well as the rule grouping and subsequent packet header classification method used in our IDS implementation.

4.2.1 Snort Rule Grouping

The Snort IDS implements a very coarse-grain rule grouping method [12], in which only the protocol and one or two additional parameters are used. This method is implemented with two 64K arrays of sets, called "Source" and "Destination" as well as an additional, "Generic" rule set. Initially all sets are empty.

For TCP and UDP rules, Snort checks the specified source and destination port rule header parameters. If the source port has a unique value *x* (i.e not "any", negated port or a port range) then the rule is included in rule set "Source[*x*]". Likewise, if the destination port has a unique value *y*, the rule is inserted into set "Destination[*y*]". If neither the source or destination ports are unique, the rule is inserted into the "Generic" set. For ICMP and IP rules the process is simpler, since only one value is of interest: the ICMP type for ICMP rules and the protocol for IP rules. If the rule parameter is a specific number *z*, then the rule is inserted in "Source[*z*]", otherwise it is placed in the "Generic" set.

This rule grouping method has the benefit that the rule set matching a packet header can be discovered very fast: if an incoming TCP or UDP packet has source and destination port values 'i' and 'j' accordingly, only "Source[i]" and "Destination[j]" need to be examined. If both are empty, then the payload is scanned using the "Generic" set. If either is non-empty, then it is used to examine the payload. In case that both sets are non-empty then a conflict has appeared and possibly both sets will need to be examined. ICMP and IP packets are processed even faster, since only the "Source" array and the "Generic" set need to be checked. In this way, both the "fast" and the "distinct" criteria are satisfied rather well.

However, this rule grouping method also has significant disadvantages. First of all is the fact that rules with negated values or values inside a range are placed in the "Generic" set, which under conditions leads to inaccuracy. The "Generic" rules may be ignored in cases where they should actually be verified. For example, lets consider rule A with a port value of *X* and rule B with a port value of !*Y*, where *X* and *Y* are different. Each rule also specifies several other parameters. If a packet arrives with a port value of *X*, then it will be compared only against A even though it should also activate B. In short, the "complete" criterion is not satisfied very well.

One final disadvantage is that the resulting rule sets may contain a relatively large number of rules, of which plenty may later be discarded. For example, several rules of a set may actually be invalid because they have specified a different IP address than that of the packet. The patterns for these rules are matched during the payload scan, but must then be discarded because they don't satisfy the necessary conditions, causing the software to waste additional effort. This indicates a small degree of compliance with the "minimal" criterion.

4.2.2 Fine-Grained Header Classification

Since we are aiming for a hardware implementation of our system, we plan to take advantage of the available parallel processing capabilities to use a more fine-grained rule grouping method. In a nutshell, we use the same parameters as Snort with the addition of the Source and Destination IP addresses.

Specifically, when grouping rules into sets, the following parameters are always used.

- Protocol
- Source IP
- Destination IP

Aside from these three standard parameters, some protocol specific parameters are also used when grouping. For TCP and UDP rules, the source and destination port numbers are also used. ICMP rules use the ICMP type, which is specified by the "itype" *rule option*, while IP rules use the protocol number, which can be specified by the "ip_proto" *rule option*. When such a rule option is not present, the parameter is considered to have an "any" value.

The rule grouping procedure is very simple: all rules that have exactly the same parameters belong to the same rule set. Since rules usually specify a string to be located inside the payload, every rule set is usually associated with a specific pattern set, and the packet payload must be scanned for any of those patterns using traditional multiple string matching methods. Of course, it is possible to have rule sets in which no rules specify strings to be matched.

At this point we must clarify some exceptions to the rule grouping procedure, as they appear in our implementation. First, when a rule header specifies a list of IP addresses (e.g. [x; y; z]) as source or destination IP then that rule belongs to one set for each address in the list. If a rule header had address lists on both the source and destination IP address, then we take all possible combinations of one source IP and one destination IP and insert the rule to the corresponding rule set. For example, if the header source IP is a list of 3 addresses and the destination IP is a list of 4 addresses, we consider each of the 3*4=12 IP address combinations as a different rule set. This has been done to simplify the header matching process so that each time we will check for only one IP value.

A second exception is the bi-directional rules. Because we want each rule set to be unidirectional (for implementation simplicity), we consider each bidirectional rule as two unidirectional rules: one with the normal source and destination header parameters and one with the transposed parameters.

One final detail, which is also implemented in Snort, concerns the IP rules. When an IP rule specifies a range of protocols which includes one or more of the other three (i.e. TCP, UDP or ICMP) rule protocols, then this rule is inserted not only to the IP rule sets, but also to rule sets for any other matching protocols. For example, a rule in the form of

```
alert ip $HOME_NET any -> $EXTERNAL_NET any (ip_proto: <8; …)
```

will also be inserted as a TCP (protocol number=6) and ICMP (protocol number=1) rule with "$HOME_NET any -> $EXTERNAL_NET any" header parameters.

In summation, the performance of this header classification method according to the four specified criteria is the following:

- **Distinct**: Poorly satisfied, since all rule sets matching the header of a packet are activated. Our tests have shown that as many as 29 different rule sets can be activated for a single packet.
- **Complete**: Excellent satisfaction. Because all matching rule sets are activated and checked, no possibly matching rules will be ignored
- **Minimal**: Satisfied to a large degree. The header classification process doesn't take into account every possible header parameter specified by a rule (e.g. flags, Time To Live). As such, some content matching rules in a set may later be discarded.
- **Fast**: Excellent satisfaction. Due to the architecture of the header classification component described in section 5.3.1, all groups matching a packet are found in only two cycles.

4.3 Dividing Rule Groups into Subsets

We have focused on the Aho-Corasick algorithm and its derivatives, which would essentially mean that every rule set (and, by extension, every pattern set) has a corresponding finite state machine. However, several pattern sets are large enough to require fsms with a few thousand states, which amounts to very large memory requirements. In addition, fsms with a very large number of states often use a large number of different characters, which means that the Split Aho-Corasick architecture does not offer significant memory gains. A simple way to reduce the size of the required memory is to split large pattern groups into smaller subsets and associate a smaller fsm with each derived subset.

In order to clarify the benefits of rule set division, we present the following example: a relatively large pattern set is represented using the simple version of our Split Aho-Corasick architecture and the derived fsm has 512 states, while the character translation table uses 128 different values. Every cell of the character translation table requires 7 bits, which means that the character translation requires a total of 256*7 = 1.75 Kbits. Since we have a total of 512 states, each cell of the state lookup table is 9 bit wide and we need 128 cells for each fsm state due to the Split-AC architecture. As a result, the state lookup table requires a total of 512 * 128 * 9 = 576 Kbits, while the total memory requirements are 576+1.75=577.5 Kbits.

Now let's assume that this rule set can be divided into four smaller subset, with each having an fsm of 128 states and a character translation table of the same size as before. Following a similar logic as before, each character translation table will be 256*7=1.75 Kbits, while each state lookup table will require 128*128*7 = 112 Kbits of memory. This means that each fsm requires 113.75 Kbits of memory, with a total of 455 Kbits of memory for all four subsets. Compared to the large, undivided rule set we have a reduction of 21% in memory requirements.

This example is actually a rather pessimistic estimation of the benefits from rule set division since we assumed that the size of the character translation table remained the same. A decrease in the number of states in an fsm is usually accompanied by a reduction in the number of different characters, which leads to a drastic reduction of both the character translation and the state lookup tables. Returning to the example above, a more accurate estimation would be that the reduction of the number of states by a factor of 4, from 512 to 128, is accompanied by

a reduction of the different characters in each fsm by a factor of 2, from 128 to 64. In this case, each character translation table would require 256*6=1.5 Kbits while each state lookup table would require 128*64*7 = 56 Kbits, for a total of 4*57.5 = 230 Kbits for all four smaller fsms and a reduction of memory requirements by 61% in comparison to the large set.

However beneficial rule set division may be, it is not without limitations. The number of states in a finite state machine cannot be less than the length of the longest pattern in the corresponding rule subset. Even if we want the maximum number of states per fsm to be 32, a string of 120 characters can only fit in an fsm of at least 121 states (120 for each character of the string plus 1 for the initial state). As such, whenever a string is longer than the desired maximum number of states, the fsm it is inserted to will have a maximum number of states equal to the smallest power of two that can accommodate that string. I.e. a pattern of 120 characters would be inserted in a 128 state fsm.

4.3.1 Rule Set Division Algorithm

When dividing rule sets into subsets, the first step is to separate the case sensitive strings from the case insensitive ones, for the reasons described in section 3.3.4. After this initial separation, the strings are further divided into subsets using the following algorithm which, after some experimentation, we found to provide the best results.

Algorithm 4.1: *Division of a rule set into smaller subsets.*
Step 1. New subset. Create an empty fsm.
Step 2. Select the longest remaining pattern from the initial rule set and insert it in the fsm. Remove the pattern from initial rule set.
Step 3. If the length of the pattern is greater than the desired maximum number of states, the maximum number of states for this fsm only is the smallest power of two that can accommodate the pattern.
Step 4. Calculate the *distance* of each remaining string from the contents of the fsm.
Step 5. Select the pattern with the smallest *distance* that can fit in the fsm without going over the maximum number of states.
Step 6. If no pattern was found go to step 1.
Step 7. Insert the selected pattern in the fsm and remove it from the initial set. Go to step 4.

The above algorithm is repeated until we are at step 1 and there is no available pattern.

In the algorithm we mention a *distance* value that is associated with each pattern. This value represents the difference of each pattern from the contents of the fsm. A pattern's *distance* is defined in the following manner:

- Initially 0.
- Incremented by 1 for every character of the pattern that doesn't appear in any transition of the fsm.
- Decremented by the length of the common prefix that the pattern has with the contents of the fsm.

The *distance* allows us to select the pattern that will have the least possible cost when inserted in the fsm based on two criteria. The *distance* is increased for each character different from the contents of the fsm because different characters increase the required amount of memory for the Split-AC algorithm. On the other hand, when a pattern has some common prefix with the contents of the fsm then some states for the pattern have already been defined and we can use them "for free".

The algorithm we described above was our final selection. However, before that we had tried two other subset division algorithms. The first algorithm we tried relied only on pattern length: first, the longest pattern was selected and placed in an empty subset and afterwards we selected the longest pattern that could fit without going over the maximum number of states. The other subset division algorithm we tried was very similar to the final algorithm; the only difference was that we didn't take into account the common prefix length when calculating the *distance* of a pattern. A simple comparison of the three algorithms for a maximum fsm size of 16, chosen because it leads to the largest number of subsets, and a character occurrence frequency threshold equal to 0.01 is shown it Table 4.1.

TABLE 4.1

Subset division algorithm comparison showing the resulting number of subsets and the required memory size

Division Algorithms		
Algorithm	**# of Subsets**	**Total Memory**
1st : Length-Based	1029	4346.31 Kbit
2nd : *Distance*, Ignore Prefix	1033	4202.4 Kbit
Final : *Distance* with Prefix	864	3265.2 Kbit

These above results show that our algorithm selection is justified. However, it is interesting to note the differences between the first and second algorithms: while the first algorithm has less subsets and, as a result, less individual memories, in total it actually requires a little more memory. This is due to the fact that the second algorithm takes advantage of common characters between patterns, which leads to smaller state lookup tables.

CHAPTER 5 – IDS ARCHITECTURE USING THE SPLIT-AC ALGORITHM

5.1 Introduction

This chapter is dedicated to the description of the architecture we designed in order to implement the Split-AC pattern matching algorithm in a simple yet efficient hardware Intrusion Detection system. We also describe the manner in which the Split-AC algorithm and the IDS architecture utilizing it were implemented and tested.

We decided on implementing the algorithm in FPGA's instead of ASIC's because of the frequent updating of the rules used in IDS: traffic rules are updated very frequently, on a monthly if not weekly basis, and the modification of even one existing rule or the addition of a new one would require the creation of a new ASIC, with a corresponding cost in time and money. On the other hand, with FPGA's we need simply generate the appropriate code for the new traffic rule set and download it on the FPGA chip.

This chapter is divided into four different parts. In the first and largest part, we describe in detail the overall system architecture as well as the architecture of the various subsystems. In the second part we present a significant challenge we faced when trying to implement the Split-AC algorithm in FPGA's and the manner in which that challenge was overcome. The third part of the chapter focuses on the manner in which the required systems are implemented. Finally, the fourth part describes the testing and validation methods used for the various stages of this thesis.

5.2 System Overview

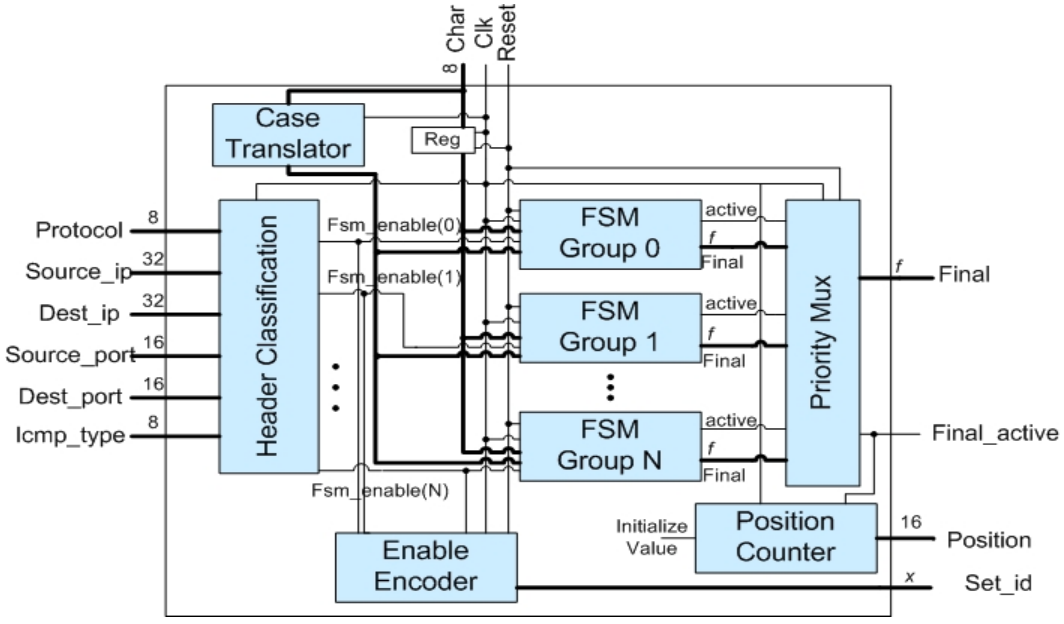

Figure 5.1: Overview for the initial split-AC algorithm (Revision 0).

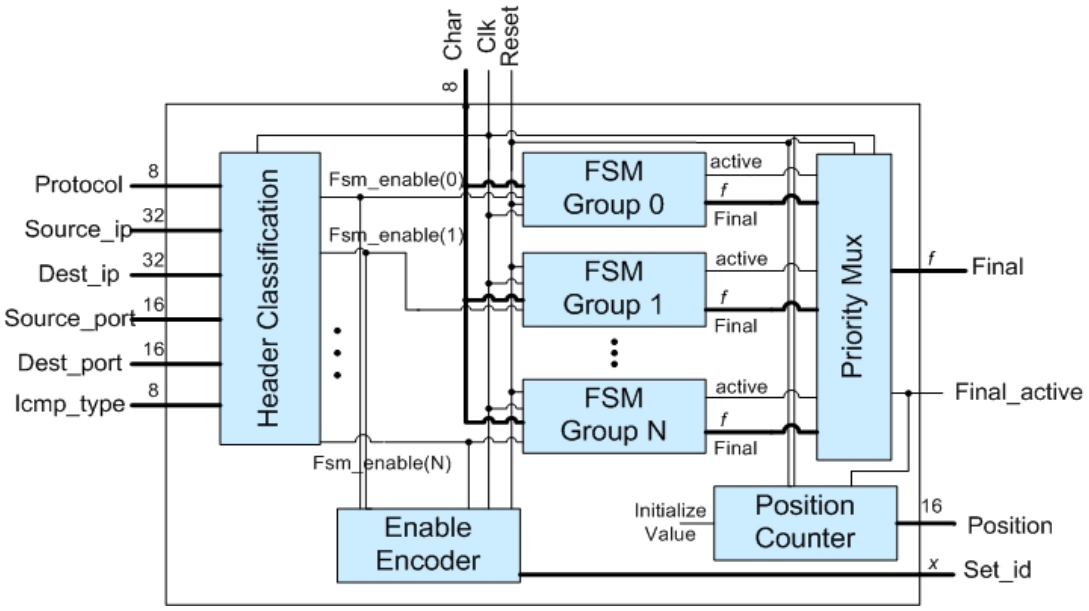

Figure 5.2: Split-AC Overview for algorithm Revisions 1 and 2

5.2.1 Description

The block diagram in Figure 5.1 presents a general overview of the Intrusion Detection System architecture which uses Revision 0 (original algorithm) of the Split-AC pattern matching algorithm, while the block diagram in Figure 5.2 shows the overview for algorithm Revisions 1 and 2.

The purpose of this system is to classify an incoming packet and examine its payload for the existence of specific strings. Specifically, when a new packet arrives, the inputs concerning packet header parameters receive new values. These new values are processed by the header classification subsystem, which in turn deactivates most fsm groups (by setting the corresponding enable signal to 0) and activates only the necessary fsm groups. The header classification process takes only 1 cycle to complete.

After the header classification is complete, the fsm groups start processing the payload of the packet at a rate of 1 byte per cycle. Whenever some strings are matched the appropriate output signals are activated. Also, the "fsm_enable" signals are encoded and the "set_id" output becomes equal to the serial number of the most specific rule set matching the packet. This information can be used by overlaying software systems to avoid the time-consuming process of packet header matching.

We assume the existence of a higher level of abstraction which cooperates with the system we propose. This abstraction level will extract the parameters from the header of the packet (i.e protocol, source_ip, icmp_type etc.), send them to the corresponding system inputs and raising the "reset" signal for two cycle in order to clear intermediate results, reinitialize the position counter and give the system time to process the header (the "Header Classification" module is not reset and requires two cycles to process the header data). This layer is also responsible of feeding the Split-AC system with one payload byte at every clock cycle, maintaining the header parameter inputs steady until the payload is fully processed and interpreting the "Final" output vector. Finally, this layer is responsible for the verification of other

rule parameters, the matching of other signatures for rules specifying more than one signature as well as taking appropriate action whenever rules are activated.

5.2.2 System Inputs

- **Protocol** (8 bits). The protocol number of the packet. Indicates if the packet uses TCP (0x06), UDP (0x11), ICMP (0x01) or other protocols.
- **Source_ip** (32 bits). The source IP address of the packet.
- **Dest_ip** (32 bits). Destination IP address.
- **Source_port** (16 bits). Source port of the packet. Used only for TCP or UDP protocols, in case of other protocol the value must be 0.
- **Dest_port** (16 bits). Destination port of the packet. Same as for source_port.
- **Icmp_type** (8 bits). The ICMP type of an ICMP packet. For all other protocols this value must be 0.
- **Char** (8 bits). Input character. In essence the next byte to be examined from the packet payload.
- **Clk** (1 bit). The system clock.
- **Reset** (1 bit). Reset signal. Clears all data and results, apart from the "header classification" subsystem. Reinitializes the "position counter".

5.2.3 System Outputs

- **Set_id** (*x* bits). The serial number of the rule set whose header matched that of the packet. If more than one rule set headers match then this output is equal to the smallest rule set serial number. *x* is the minimum number of bits that can represent all possible serial number values.
- **Final_active** (1 bit). A '1' indicates that the "final" output is active, meaning that in this cycle one or more signatures were matched in one or more fsms. Generated by OR-ing the "active" outputs of all the fsms.
- **Final** (*f* bits). Result vector showing the specific fsm and signature(s) that were matched. More details about the purpose of this vector can be found in Section 5.5.1 describing the fsms. Whenever the "Final_active" output is 0, this will also be 0. In case that two or more fsm groups have active "final" data in the same cycle, the data of the fsm group with the smallest serial number (highest priority) is selected; the others are lost.
- **Position** (16 bits). The payload byte containing the last character of a discovered signature.

5.2.4 Subsystems

- **Header Classification.** Uses the header parameters of the packet to enable the appropriate fsms for processing the payload. Described in detail in section 5.3.

- **FSM Group x.** Each block represents the fsms required to scan the payload for all the strings belonging to rule set with normalized id equal to *x*. As we have explained in section 4.3, one rule set may be divided into several subsets and, as such, require several fsms. For rule sets that include no strings to be matched, this block is omitted. The normalized id of a rule set is defined in the following manner:
  - TCP rule sets receive ids from 0 to N_{TCP}-1 (N_{TCP} = total TCP rule sets)
  - UDP rule sets receive ids from N_{TCP} to $N_{TCP} + N_{UDP}$ -1
  - ICMP rule sets have ids from $N_{TCP} + N_{UDP}$ to $N_{TCP} + N_{UDP} + N_{ICMP}$ -1
  - IP rule sets have ids from $N_{TCP} + N_{UDP} + N_{ICMP}$ to $N_{TCP} + N_{UDP} + N_{ICMP} + N_{IP}$ -1
- **Enable_Encoder.** Functions as a priority encoder for the "fsm_enable" signal vector. Described in detail in section 5.6.
- **Priority mux**. A priority multiplexer that uses the "active" results of the fsms to select the "final" output of the fsm group with the smallest serial number. Described in detail in section 5.7.
- **Case Translator.** Required only for Revision 0 of the Split-AC algorithm. A simple block of behavioral logic that converts small case letters into capital letters. This is done by subtracting 0x20 from the "char" input whenever its value is between 0x61 (the letter 'a') and 0x7A (the letter 'z'). The output vector of this block is 8 bits wide and is registered. This block is unnecessary for Revisions 1 and 2 because the case translation is handled by the Character Translation memories.
- **Position Counter.** A simple 16-bit counter created with behavioral VHDL that is incremented by 1 in each cycle. This counter is used to indicate the location in the payload where signatures were discovered. This counter is must be reset whenever we commence the payload scan of a new packet. When reset, the counter is initialized to a negative value equal to the number of cycles needed for a character to cross the entire pipeline, which is equal to 3 plus the pipeline stages of the priority multiplexer (for Revision 0 an extra delay cycle is required due to the "case translator"). In this manner, the counter will show 0 when the first payload byte has been completely processed and the results have arrived at the system output. This counter also contains an "output_enable" signal, which is wired to "Final_active": if the signal is cleared then the output of the counter is 0, while if the signal is set the counter outputs its value.

5.3 Header Classification

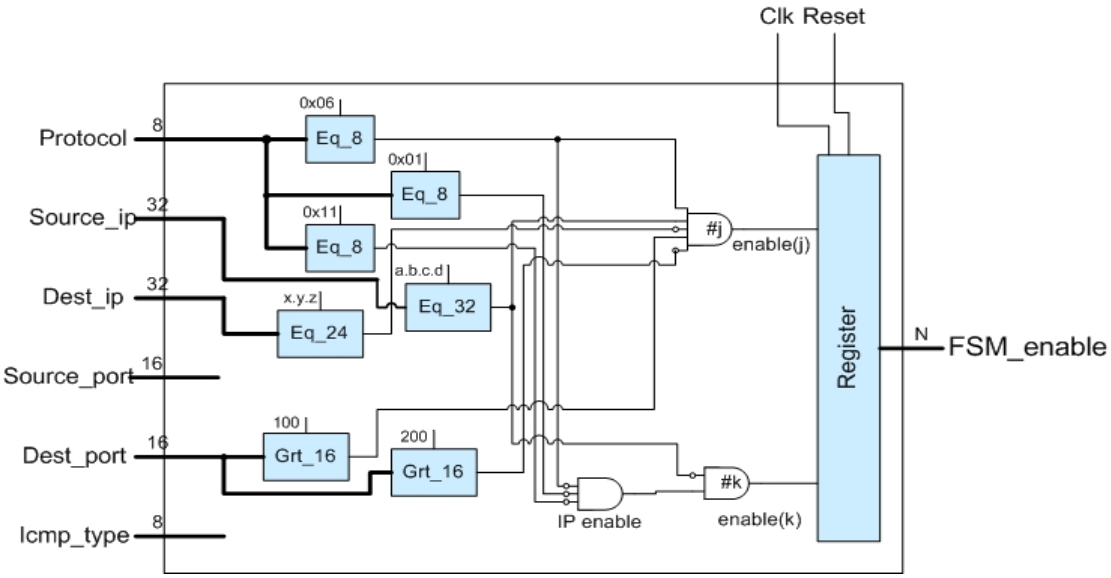

Figure 5.3: Header classification subsystem. The result of every comparator is registered (not shown).

5.3.1 Description

The block diagram of Figure 5.3 is intended to give a general idea of the "Header Classification" subsystem's architecture. We have presented an extremely efficient header classifier module in [16] which was, however, not used in this case due to the fact that it was easier to automatically generate the necessary code for this simpler architecture.

The purpose of the Header Classification subsystem is to check the header parameters of a packet and activate the rule sets that apply to the packet. As we described in section 4.2.2, the header classification process uses 3 to 5 parameters according to the protocol.

We use only equality and "greater-than" comparators to compare each header parameter to a number of specific values. Whenever a rule specifies a "!x" value we simply negate the result of an appropriate equality comparator. Likewise, if the value we want is "<y", we use a "greater-than" comparator with "y-1" as a stable input and negate the result. If the desired value is a range of numbers in the form of "10:200", we use two "greater-than comparators, one for '9' and one for '201', negate the result of the '201' comparator and AND the results of the two comparators together.

The architecture is pipelined and requires two cycles to output the results. Although not shown in the block diagram, the results of every equality and "greater-than" comparator are registered. These comparator registers along with the final, large output register make up the two-cycle pipeline delay.

The appropriate results of these comparators are combined using one AND gate for every rule set; if all parameters for a specific rule set are verified then the result of the AND gate will be set and the corresponding rule set will be enabled.

Figure 5.3 shows the logic for activating two different rule sets. The first set, denoted #j, is a TCP rule set with a source IP address of "a.b.c.d", a destination IP of "!x.y.z.0/24", "any" source port and destination port between 101 and 199. On the

other hand, the second set, denoted #k, is an IP rule set with "!a.b.c.d" source IP and "any" destination IP.

We minimize the required number of comparators by reusing as many results as we can, either in their normal form or negated. As can be seen in Figure 5.3, the comparator used to check if the source IP is "a.b.c.d" is used by both #j and #k sets, only #j uses the result in it's normal form while #j negates it.

Another manner in which we try to save logic is the IP comparators. Several rules specify "/24" or even larger subnets (remember: a smaller /x value translates to a larger IP subnet). In these cases we use the smallest possible equality comparator and compare only the required most significant bits while zero-padding any unnecessary LSB's. For example, if a rule specified a source IP address of "x.y.z.0/22" then we could use a 24 bit equality comparator, set the two LSB's of the comparator's input to 0 and the 22 MSB's to the corresponding MSB's of the Source_IP input.

One minor variation among the rule sets is encountered for IP rule sets. For TCP, UDP or ICMP rule sets, we use three equality comparators to check if the protocol is one of the three corresponding values. Since IP rules apply in all cases other than TCP, UDP or ICMP, we AND together the inverted results of the three protocol comparators to generate the IP_enable signal (as shown in Figure 5.3), which is then used for all IP rule sets.

5.3.2 Input Signals

- **Protocol** (8 bits). The protocol number of the packet. Indicates if the packet uses TCP (0x06), UDP (0x11), ICMP (0x01) or other protocols.
- **Source_ip** (32 bits). The source IP address of the packet.
- **Dest_ip** (32 bits). Destination IP address.
- **Source_port** (16 bits). Source port of the packet.
- **Dest_port** (16 bits). Destination port of the packet.
- **Icmp_type** (8 bits). The ICMP type of an ICMP packet.
- **Clk** (1 bit): The system clock.
- **Reset** (1 bit). Reset signal.

5.3.3 Output Signals

- **FSM_enable**(N bits). This output vector is as wide as the total number of different rule sets. When bit *i* is set then the rule set with a normalized id equal to *i* can be applied to this packet. When bit *i* is cleared, the corresponding rule set doesn't apply to this packet.

5.3.4 Subsystems

- **Eq_*X*.** *X*-bit wide equality comparator. Compares its input with a specific value, denoted with a vertical line and a numerical value at the top of the block in the

block diagram. The output bit is set when the two values are equal and cleared when they are not equal. We use 4 different sizes of equality comparators: 8, 16, 24 and 32 bits wide.

- **Grt_*X*.** *X*-bit wide "greater-than" comparator. Compares its input with a specific value, denoted with a vertical line and a numerical value at the top of the block in the block diagram. The output bit is set when the input signal is greater than the specific value and cleared if is equal or less. We use 2 different sizes of "greater-than" comparators: 8 and 16 bits wide.
- **Register**. *N*-bit wide register, where *N* is the total number of rule sets. In reality this is not a single register but a series of 32-bit registers set side-by-side.

5.4 FSM Group

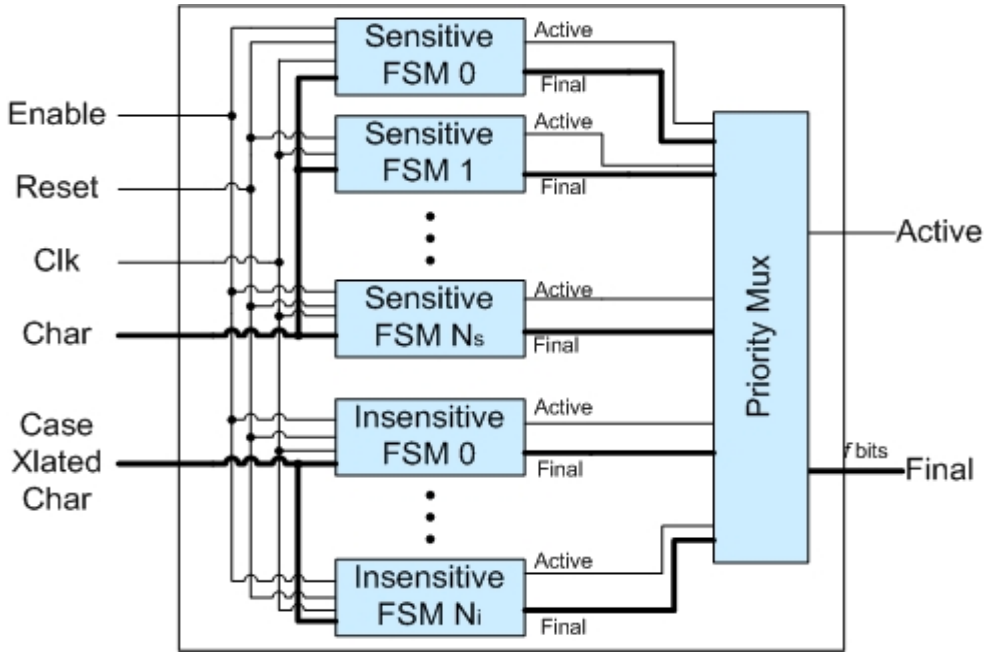

Fig.5.4: FSM Group subsystem for Revision 0. Revision 1 and 2 use "Char" wherever "Case Xlated Char" is used in Revision 0.

5.4.1 Description

The block diagram of figure 5.4 shows a typical "FSM Group" subsystem for Revision 0 of the AC algorithm. The only difference for Revisions 1 and 2 is that the subsystem has no "Case Xlated Char" input; in its place the "Char" input is used.

The FSM Group subsystem represents a collection of all the small finite state machines that are used to verify the rules of a specific rule set. Every FSM box in the diagram represents the fsm for a small subset of this rule set. The outputs of this subsystem are selected by the priority multiplexer in order to avoid conflicts should several fsms have active outputs in the same cycle.

5.4.2 Input Signals

- **Enable**(1 bit). Enable signal for the specific rule set.
- **Reset**(1 bit)**.**
- **Clk** (1 bit)**.** System clock
- **Char** (8 bits). The next payload byte to be processed.
- **Case Xlated Char** (8 bits). Required only for Revision 0. Whenever the "Char" input is a small letter, this input is equal to the equivalent capital letter. Used only by the case insensitive fsms.

5.4.3 Output Signals

- **Final_active** (1 bit). A '1' indicates that the "final" output is active, meaning that in this cycle one or more signatures were matched in one or more fsms.
- **Final** (*f* bits). Result vector showing the specific fsm and signature(s) that were matched. More details about the purpose of this vector can be found in Section 5.5.1 describing the fsms. Whenever the "Final_active" output is 0, this will also be 0.

5.4.4 Subsystems

- **FSM**. A finite state machine corresponding to a specific subset of this rule set. The distinction between "Sensitive" and "Insensitive" fsms shown in figure 5.4 is for clarification purposes. There is no functional difference between the fsms for case sensitive and those for case insensitive subsets; the only difference is their inputs, and even that applies only for Revision 0. More details in Section 5.5.
- **Priority mux**. A priority multiplexer. If more than one fsms have result data at the same cycle, this mux selects an fsm based on case sensitivity (sensitive fsms have priority) and then based on serial number (smaller numbers gain priority). In reality, this multiplexer isn't actually used at this level of the architecture: instead of having one priority multiplexer for the results of each fsm group and another to select between groups, we use only one, global priority multiplexer (the one shown in figure 5.1) to select among all subset results. The reason for showing the priority mux at this architectural level is to give the idea of how the fsm results are processed, as well as to simplify the architectural overview in figure 5.1 by using only two output signals per fsm group instead of $2*(N_s + N_i)$ signals.

5.5 FSM

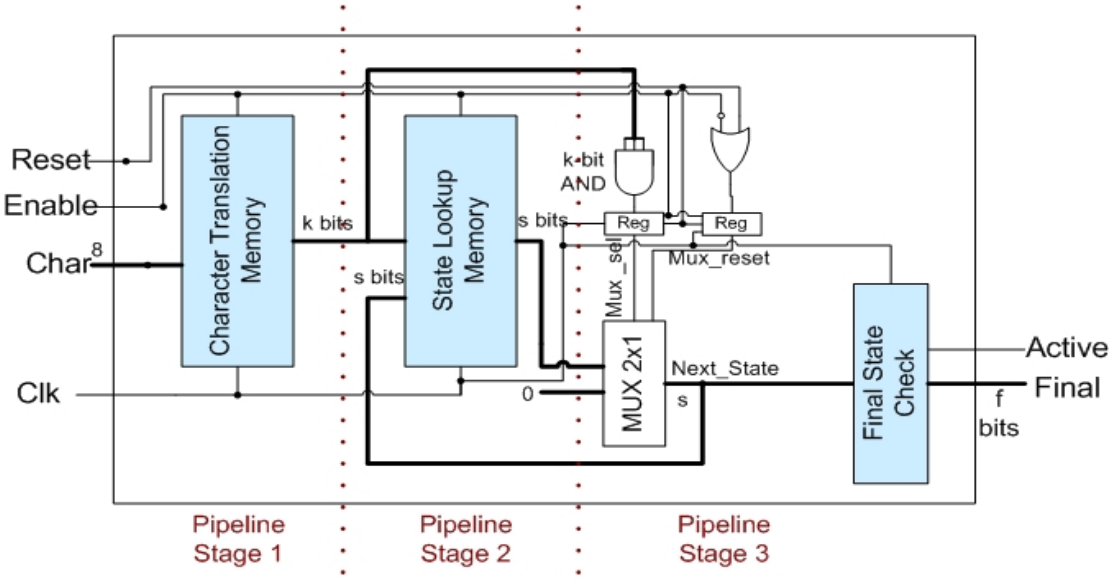

Fig. 5.5: FSM architecture in the trivial case where there are no infrequent characters

5.5.1 General FSM Description

The block diagram in figure 5.5 shows the trivial case of fsms with no uncommon characters and hence no CAM, while the block diagrams of figures 5.6, 5.7 and 5.8 show the fsm architecture for algorithm revisions 0, 1 and 2 respectively.

As we mentioned when describing the Split-AC algorithm in section 3.3.1, input characters are split into 3 groups.

- frequent characters: use a lookup table to find the next state
- infrequent characters: use a CAM-like structure to find the next state
- non-existent characters: the next state will be 0 (initial state).

The character input of the fsm subsystem is used to address a Character Translation memory, which informs us whether the character is frequent, infrequent or non-existent.

If the character is frequent, the translated character value and the current fsm state are used to address the State Lookup memory, which provides us with the next fsm state. For infrequent characters, we feed the current state and the character to the Infrequent Character CAM, and its output is the next state. Last, for non-existent characters the next state is always 0.

At any point, the value of the next state for the fsm, whether it will be set to 0 or given by the Lookup memory or the CAM, is controlled by a 2-to-1 multiplexer. The “select” and “reset” signals that control this mux are generated in each cycle by the result of the Character Translation memory. These signals are pipelined so as to arrive at the multiplexer when the processing of the corresponding character has been completed by the CAM and State Lookup memory.

The fsm components labeled “Character Translation Memory”, “Infrequent Character CAM”, “State Lookup Memory” and “Final State Check” all have

registered outputs. This fact along with the fsm architecture results in a 3-cycle pipeline. The operations corresponding to each pipeline cycle are as follows:

1) The "Character Translation Memory" translates the input character.
2) The "State Lookup Memory" and the "Infrequent Transition CAM" search concurrently for the next fsm state. Generate mux control signals.
3) Mux control signals used to select the next state. "Final State Check" block generates appropriate output signals.

Regardless of the algorithm revision, *k* is the minimum number of bits needed to describe the contents of the Character Translation memory while *s* is the minimum number of bits that can represent every state of this fsm.

Finally, the fsm informs us of any signatures it discovers via the *f*-bit wide "Final" output vector. The width (*f* bits) of this vector is the result of a $x+y+1+z$ function. The *x* most significant vector bits represent the serial number of the rule set that the fsm belongs to. The following *y* bits are the serial number of the particular subset to which the fsm corresponds. The next bit indicates the character sensitivity of the particular subset (1 for case sensitive subsets, 0 for case insensitive subsets). Finally, the *z* least significant bits represent the particular final state that the fsm reached, effectively showing us which signatures were matched. Note: the number *z* is not defined by the final states of this fsm but by the maximum number of final states in any fsm of the entire system.

5.5.2 Input Signals

- **Char** (8 bits). The next character (payload byte) to be processed.
- **Reset** (1 bit).
- **Clk** (1 bit). System clock
- **Enable** (1 bit). Indicates if this fsm is active. When cleared, the fsm doesn't process incoming data.

5.5.3 Output Signals

- **Active** (1 bit). If this signal is set then during the last cycle the fsm was in a state where one or more signatures match. If the signal is cleared, then the fsm was in a state were no signatures are matched.
- **Final** (*f* bits). This signal informs us of the rule set, the subset and the case sensitivity of the specific fsm, as well as the final state that the fsm has reached. This vector is non-zero only when the "active" output is 1 i.e. when the fsm reached a final state in the previous cycle.

5.5.4 Subsystems

- **Character Translation Memory**. A memory holding the data necessary to translate the incoming character into a value from 0 to 2^k-1, as required by the Split-AC algorithm. The memory output is registered.
- **State Lookup Memory**. We use the current state of the fsm and the translated value of the input character to address this memory. The data contained in that address is the next state of the fsm for that specific state and input character. The memory output is registered.
- **Final state check**. A comparator that checks if the current fsm state is between 0 and *F*, where *F* is the number of final states in the fsm. As we mentioned earlier in section 3.2.1, this comparison is enough to inform us that signatures were successfully matched because we rearrange the fsm states so that the final states are numbered 1 to *F*. When the comparison is true, this subsystem sets the "Active" and "Final" outputs.
- **Infrequent Character CAM**. This subsystem is essentially a series of equality comparators, with which we compare the fsm state and input character to specific values and report the next fsm state whenever the current state and character match an infrequent character transition. The behavioral VHDL code for describing a CAM subsystem looks something like this:

```
S <=  state( 4 DOWNTO 0) & char( 3 DOWNTO 0) ;
PROCESS ( S, reset )
BEGIN
      IF reset='1' THEN
            temp_state <= "00000";
      ELSE
            CASE S IS
                  WHEN "000101000" => temp_state <= "00011";
                  WHEN "000111100" => temp_state <= "00100";
                  WHEN "001001100" => temp_state <= "00101";
                  …
```

The results of the CAM are registered.

5.5.5 Revision 0 Specifics

The character input for the CAM is the same as that of the entire system, only delayed by 1 cycle so that it arrives at the same time as the translated character arrives at the Lookup memory.

In the Character Translation memory each frequent character is assigned a value from 0 up to $2^k - 2$. The value for all other characters, infrequent or non-existent, is $2^k - 1$ (i.e. "11…11").

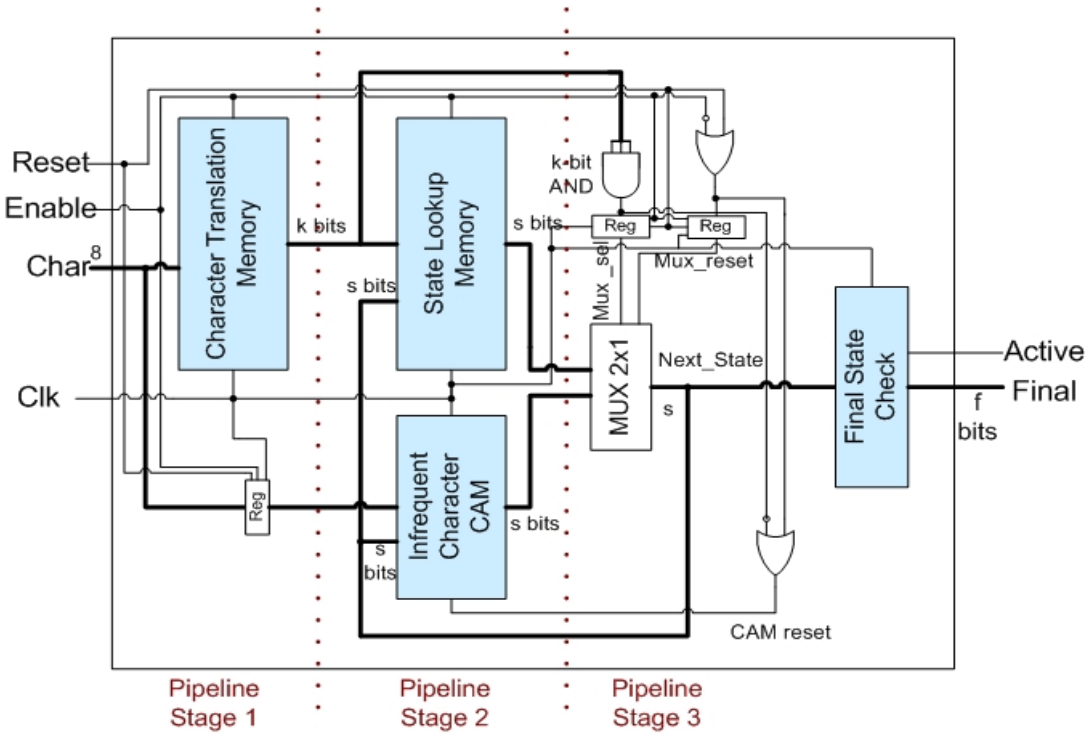

Fig. 5.6: FSM architecture for Split-AC Revision 0 (initial algorithm).

The "select" signal for the mux is generated by AND-ing together the *k* bits of the translated character. I.e. whenever the translated character is "11…11", the CAM provides us with the next state. The other control signal for the mux, "reset", is generated by OR-ing the "Reset" and "Enable" fsm inputs, i.e. whenever the entire fsm is reset or the specific rule set is not enabled.

CAM structures may be very useful, but aside from being resource demanding they also require significant amounts energy. As such it is beneficial, power-wise, for us to deactivate it whenever we can. The "CAM_reset" signal not only clears the CAM output, it deactivates the entire CAM structure. By OR-ing the mux reset and the inverted select signals, we deactivate the CAM whenever the mux selects the State Lookup Memory or none of the two. **Note**: for the CAM_reset signal we use the unregistered values of the mux control signals, in order to account for the delay cycle due to the registered CAM output.

5.5.6 Revision 1 Specifics

The Character Translation memory now contains data for both the common frequent characters and the infrequent ones. The values for frequent characters range from 0 to 2^{c1}-1, while the infrequent characters have values from 2^{c1} up to 2^{k}-2. The final value, 2^{k}-1 or "11…11" is used for non-existent characters. *c1* is always less than *k*.

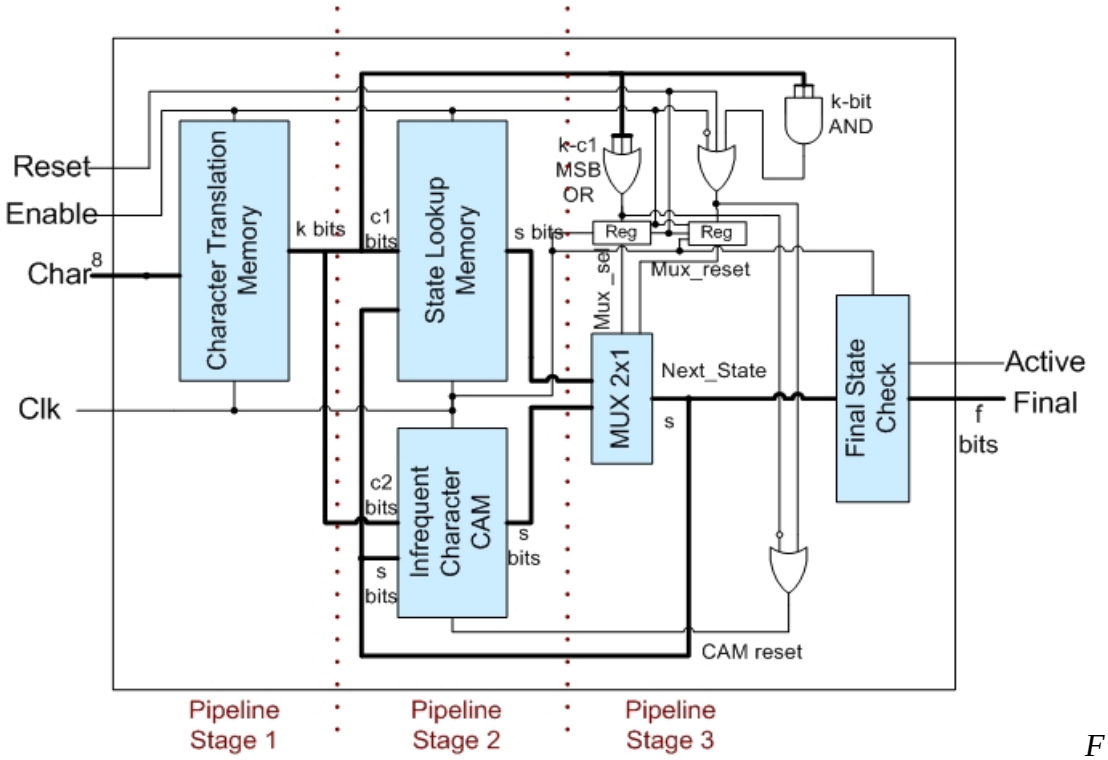

Figure 5.7: FSM architecture for Split-AC Revision 1.

The State Lookup memory now uses the *c1* LSBs of the translated character. The remaining *c1* MSBs are OR-ed together to generate the "select" signal for the mux. The reason for this can be clarified with a simple example: suppose that the Character Translation memory has 16 frequent characters numbered 0 to 15 (or "000000" to "001111") and 20 infrequent characters numbered 16 to 35 (or "010000" to "100011"). This means that k=6 and $c1$=4. For all frequent characters, the two MSBs of the translated value are 0. Conversely, whenever either of the two MSBs is 1, we can be certain that the Lookup Table won't be able to give us the next state.

The "reset" mux signal is generated by OR-ing the negated "Enable" input, the reset input and the k-bit logic AND between the translated character bits. If all k bits are set then the input character was non-existent, in which case the next state will be 0 and will not come from either the State Lookup memory or the Infrequent Character CAM.

The CAM_reset signal is generated in the same manner as for Revision 0, by OR-ing the "reset" and the negated "select" signals of the multiplexer.

Last, the character input of the CAM is now the *c2* least significant bits of the translated character, with $c2 \leq k$ being the number of bits required to describe how many *infrequent* characters are placed in the character translation mem. For example, the Character Translation mem has 32 frequent characters, numbered "000000" to "011111" and 10 infrequent ones numbered "100000" to "101001". The character input for the CAM will be equal to the 4 least significant bits, or "0000" to "1001".

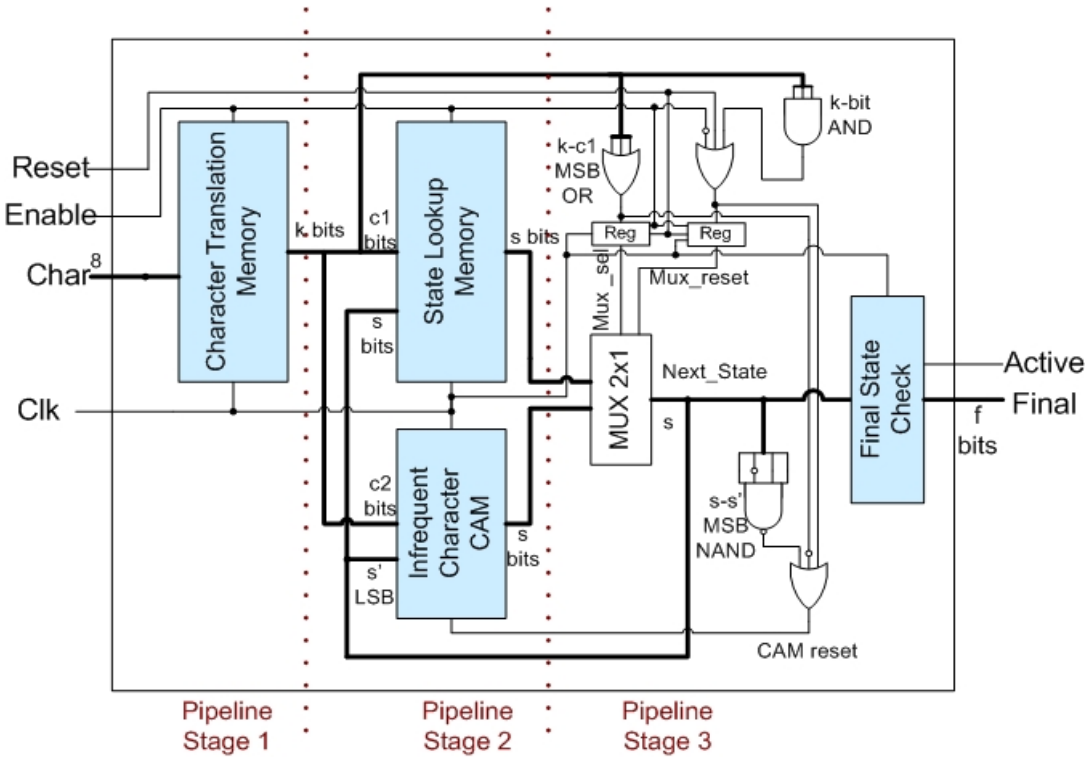

Fig. 5.8: FSM architecture for Split-AC Revision 2.

5.5.7 Revision 2 Specifics

The Character Translation memory and the State Lookup memory operate in exactly the same way as in Revision 1. The same is true for the manner that the multiplexer control signals are generated.

The difference of Revision 2 is that the states containing any infrequent characters are grouped together at the high end of the fsm. For example, if the fsm has a total of 120 states, numbered 0 to 119 (or "0000000" to "1110111"), out of which 10 states contain infrequent character transitions, those 10 states will be assigned id numbers from 110 to 119 (or "1101110" to "1110111"). It is apparent that all states containing infrequent character have the same 2 MSBs ("11"). We denote $s\text{-}s' \geq 0$ the number of common prefix bits for all states with infrequent characters. By itself, s' is the number of suffix bits necessary to differentiate between the states with infrequent characters. In this manner, whenever we are sure that the $s\text{-}s'$ most significant state bits have the specific prefix value, we can use only the remaining s' bits to address the CAM instead of using all the s state bits.

To this end, we perform a NAND among the $s\text{-}s'$ state MSBs and use the result to set the CAM_reset signal. Note that some inputs of the NAND may be inverted: if, for example, the common prefix for the states with infrequent characters was "101", then the second NAND input bit has to be negated.

5.6 Enable Encoder

5.6.1 Description

This subsystem is responsible for taking all the "FSM_enable" signals generated by the "Header Classification" component and encoding them in order to find the active rule set with the smallest id. Instead of using a very large, single cycle priority encoder (which would be very slow for the desired input sizes of approximately 400 signals) we used 16-to-4 encoders and 16-to-1 variable width multiplexers to create a tree-like, pipelined priority encoder shown in figure 5.9.

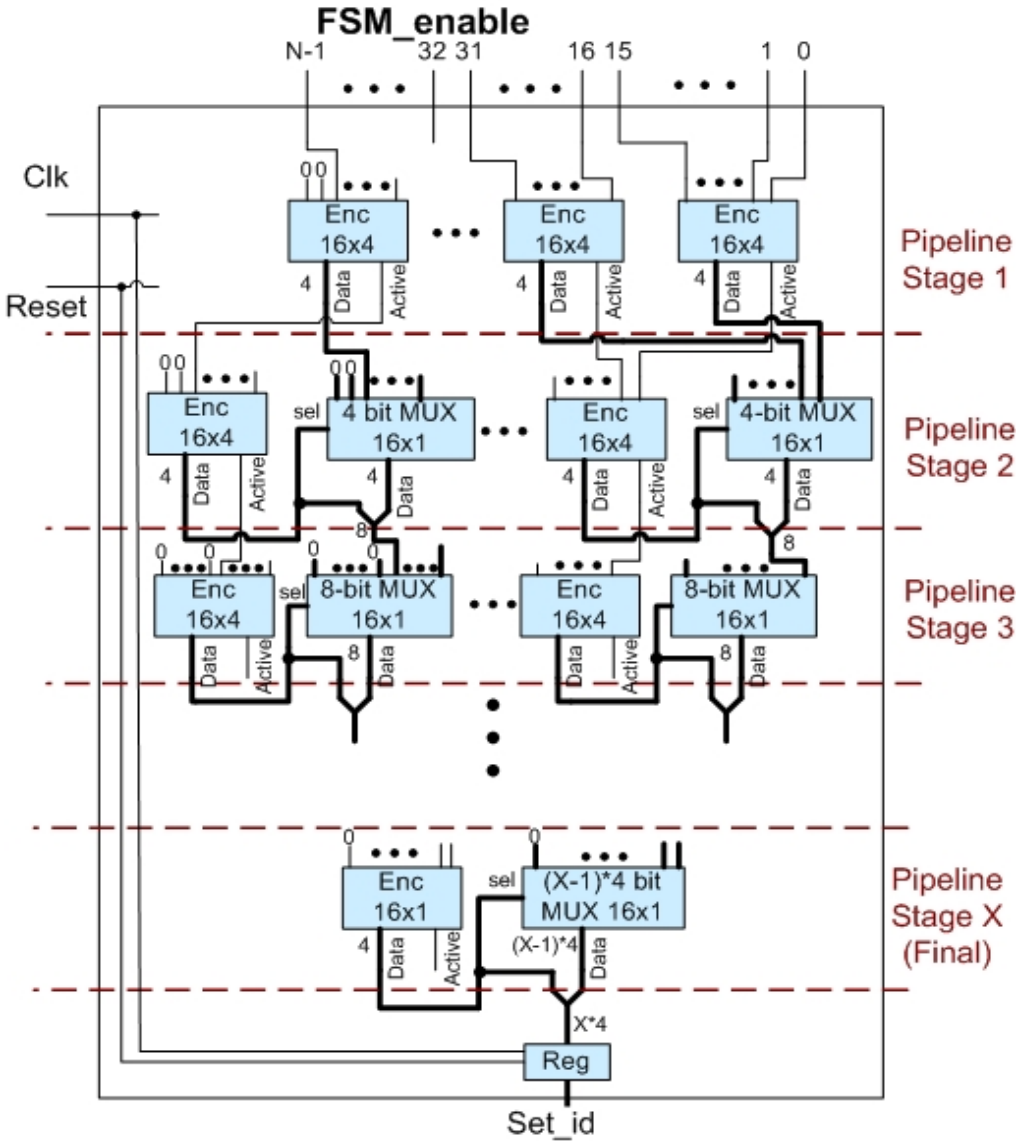

Fig 5.9: Architecture of the pipelined "Enable Encoder" subsystem.

The horizontal dashed lines shown in figure 5.9 indicate the different pipeline stages of the subsystem. In reality we use registers to hold the result values of every encoder plus the "Set_id" output, but we chose not to depict them in order to avoid

obfuscating the diagram. For that same reason we did not depict the "Clk" and "Reset" signal routing, but it is sufficient to say that the "Clk" signal goes to every register while the "Reset" signal goes to every register and multiplexer.

Since the "enable encoder" has N pipeline stages plus one output register, a total of N+1 clock cycles are needed for the results to reach the output.

The "enable encoder" is separated into different stages or levels. At the first level we have only encoders, each of which encodes 16 sequential input bits. When all the input bits of an encoder are set to zero, then the "Active" encoder output is also set to zero, otherwise it is set to 1.

From the second level and onward, there is also one multiplexer paired to every encoder. An encoder numbered *j* and belonging to level *x* is used to encode the "Active" outputs of the 16 encoders numbered *j**16 to *j**16+15 of the previous level *x*-1. The multiplexer of level *x* uses the data output of it's paired encoder as a "select" vector and selects among the corresponding 16 concatenations of encoder and multiplexer data of level *x*-1 (multiplexers of stage 2 select only from the encoder data, since there are no multiplexers in stage 1). Lastly, the final stage of the pipeline always holds only one encoder-multiplexer pair.

In every pipeline stage, the encoder (and paired mux, wherever one exists) responsible for the most significant bits of the previous stage (or, for the first stage, the "FSM_enable" input vector) will most likely have some zero inputs. This is done because it is very improbable for every pipeline stage to have a number of elements equal to a multiple of 16.

The diagram of figure 5.9 is, to some extent, more complex than actually needed. Current Snort rule distributions specify approximately 400 different rule sets, which can be accommodated by a 3 stage "Enable Encoder" that has only two encoder-multiplexer pairs at stage 2. However, this architecture is completely scalable and could be used to accommodate thousands of rule sets, should the need arise.

5.6.2 Input Signals

- **FSM_enable** (N bits). Bit vector indicating which rule sets are active and which are not.
- **Clk** (1 bit). System clock
- **Reset** (1 bit).

5.6.3 Output Signals

- **Set_id** ($\log_2(N)$ bits). The smallest rule set serial number among all active rule sets.

5.6.4 Subsystems

- **Enc_16x4**. Priority encoder 16-to-4. The Least Significant input bits have priority. Sets its "Active" output equal to 1 if at least one input bit is set, while the "Data" output is the position of the Least Significant set bit.
- **Mux_16x1**. Variable width (in 4-bit increments), 16-to-1 multiplexer.

5.7 Priority MUX

5.7.1 Description

This subsystem, whose internal design can be seen in figure 5.10, is used to forward the "final" and "active" results from each subset fsm to the corresponding outputs of the total system, as well as to avoid any possible conflicts. By "conflicts" we refer to the possibility that more than one subset fsm may have results to output at the same processing cycle. In such occasions, the multiplexer selects the output with the highest priority, which is the one with the smallest serial number.

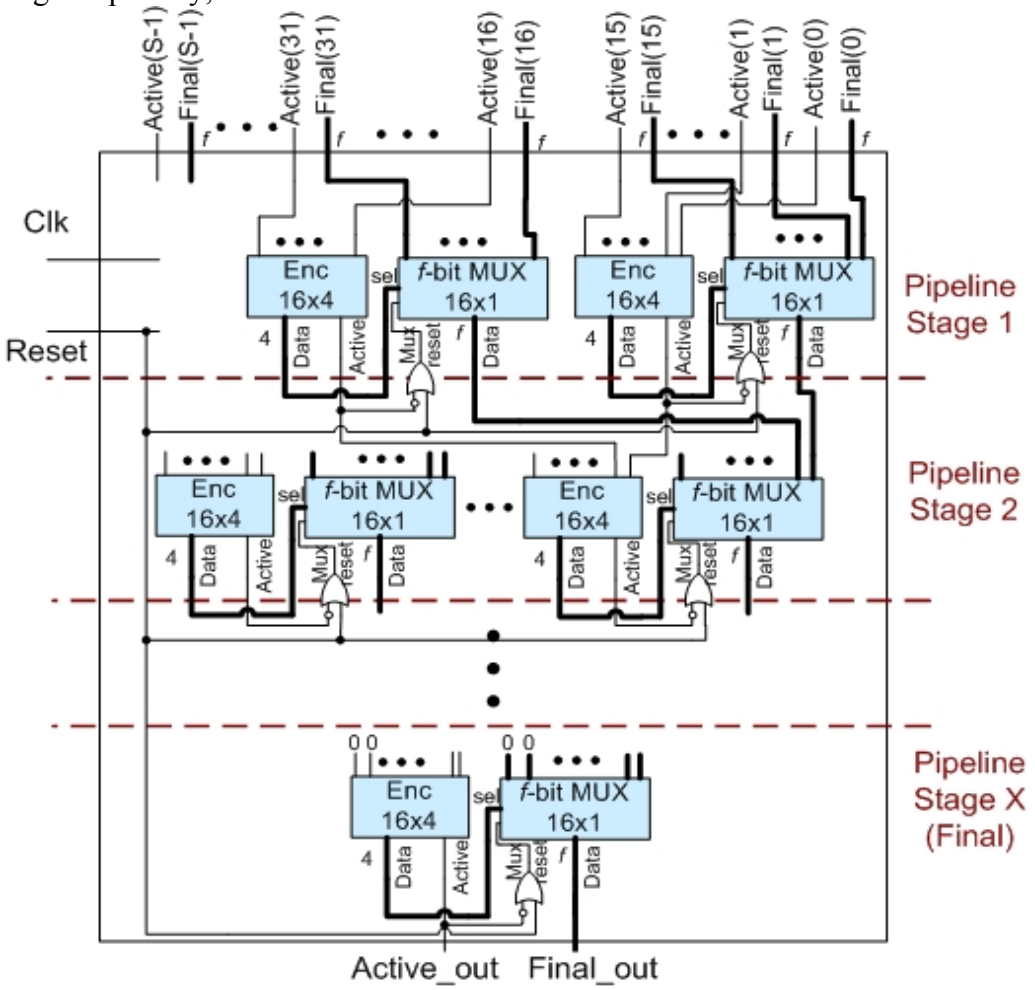

Fig. 5.10. Architecture of the "Priority MUX" subsystem.

Similarly to the "Enable Encoder" subsystem, we use a tree-like combination of small encoders and multiplexers to support the desired functionality. The entire structure is separated into different pipeline stages, as we show in figure 5.10 with the horizontal dashed lines. We use registers to hold the "Active" result of every encoder and the "Data" output of every 16x1 mux, although these registers are not shown in the block diagram in order to avoid further complicating it. One other element not shown (for reasons of simplicity) in the block diagram is the routing of the "Clk" and the "Reset" signals to the pipeline registers.

At any stage *x* of the pipeline we have a number of 16-to-4 priority encoders paired with 16-to-1 multiplexers. The priority encoder numbered *j* is used to encode the "Active" outputs of 16 sequential encoders, numbered *j**16 to *j**16+15, of stage *x*-1 or, if we are at the first stage, the corresponding "Active" input signals of the subsystem. The "Data" result of the encoder is used as a "select" signal by its pair mux, which in turn selects and forwards one of the "data" outputs of 16 stage *x*-1 multiplexers. For pipeline stage 1, the mux selects among the different "Final" input signals. The final pipeline stage always holds only one encoder-mux pair.

At every pipeline stage, some inputs for the encoder-mux pair responsible for the most significant bits may be missing. In these cases, all missing inputs are set to 0.

Aside from the pipeline registers, the "Reset" signal is also OR-ed with the "Active" result of each encoder and used to reset the mux paired to that encoder. This is done in order to reduce power consumption: when an encoder has all input bits equal to 0 then its paired mux has no data to forward. Taking into account the fact that most of the time only a handful of input signals are non-zero, it is apparent that only a small number of multiplexers will be active.

Lastly, the "Priority Mux" architecture was designed to be completely scalable. Although currently we require a priority multiplexer capable of handling from approximately 400 to little over 1000 different inputs, which can be handled by a 3-level architecture, the multiplexer could be easily extended to handle many thousands of signals.

5.7.2 Input Signals

- **Final(0)** to **Final(S-1)** (*f* bits each). The "final" output vector corresponding to a subset fsm. 'S' is the total number of subsets, or fsms, in the entire system. This signal carries information regarding which fsm located some signatures and what those signatures were. A description of what *f* signifies can be found in section 5.5.1, describing the fsms.
- **Active(0)** to **Active(S-1)** (1 bit each). The "active" output vector corresponding to a subset fsm. When an fsm has its "active" signal equal to 1, then it has located some signatures.
- **Clk** (1 bit). System clock.
- **Reset** (1 bit).

5.7.3 Output Signals

- **Active_out** (1 bit). If any "active(x)" input signal is 1, then this signal also becomes 1 after the entire pipeline has been traversed.
- **Final_out** (*f* bits). Out of all the "final(x)" inputs that have the corresponding "active(x)" bit set, the one with the highest priority is selected and sent here after the entire pipeline has been traversed.

5.7.4 Subsystems

- **Mux_16x1**. A 16-to-1 multiplexer, with *f*-bits wide inputs and outputs.
- **Enc_16x4**. A 16-to-4 priority encoder, where the Least Significant input bits receive priority.

5.8 FPGA Implementation Issues: Memory Allocation

In an ASIC implementation we would be able to freely allocate any memory, as large and wide as we require. This, however, is not yet the case with FPGA's. Cutting edge Virtex 4 FPGA chips contain a maximum of 552 dual-port memory blocks of 18 Kbits each. If we consider that the proposed Split Aho-Corasick architecture requires two memories for the fsm of each rule subset, one State Lookup memory and one Character Translation memory and that we have a minimum of 386 different rule subsets, we normally wouldn't be able to fit the entire design in a single FPGA.

This problem is surpassed by taking advantage of the characteristics of the Split-AC architecture. The first is that most required memories are small enough that we could fit several fsm memories in a single 18 Kbit memory block. However, in order to place the memories from different fsms the same memory block we must make certain that at most one of those memories is accessed at any given time.

This second constraint can be alleviated due to the fact that, usually, very few rule sets are active concurrently at any time. If we have two rule subsets that are mutually exclusive and can never be both active for the same packet, we can easily place them in the same memory block and choose whichever is necessary.

We also take advantage of the fact that FPGA memories are dual-ported, allowing us to read two different contents at the same time. Due of this, we can place the fsm memories from two subsets that are not mutually exclusive in the same block, provided that we access the data of each fsm from a different read port.

Rule sets are distinguished by their header parameters, which are the protocol and three (for ICMP and IP rule sets) or four (for TCP and UDP rule sets) other parameters. If two sets are completely disjoined in any one of those parameters then they can never be active at the same time. By default, rule sets of different protocols are mutually exclusive, while subsets of the same rule set are not.

In order to distribute the numerous required fsm memories among the available memory blocks, we initially calculate a mutual exclusion, or mutex, bitmap

for every rule set. This bitmap holds '1' for every rule set that is mutually exclusive with the current one and '0' for those that are not. The mutex bitmap of a rule set applies to all its subsets.

Each FPGA memory block is initially empty and also has two mutex bitmaps, one for each read port, both of which are initialized to all '1', meaning that the contents assigned to both read ports are mutually exclusive with every rule set. One final note is that the memory blocks used to store the State Lookup fsm memories are separate from those used for the Character Translation fsm memories. We tried using the FPGA blocks to hold both Translation and Lookup data but the memory allocation results were slightly worse.

After the initial processing is complete, we use the following algorithm to allocate the Character Translation and the State Lookup memories of each fsm to an FPGA memory block.

Algorithm 5.1: *Allocating fsm memory segments to FPGA memory blocks.*
Step 1. Find the rule set with the largest number of unallocated subsets. Select one of the unallocated subsets
Step 2. Find a State Lookup FPGA memory block, in which the data assigned to one or both read ports is mutually exclusive with the selected rule set and which can hold at least part of the selected subset fsm. Allocate as much fsm data as can fit.
Step 3. Perform a logic-AND operation between the rule set mutex bitmap and the FPGA memory block bitmap for the selected read port; the result is the new mutex bitmap for the specific read port of this memory block.
Step 4. If the selected memory block could only hold part of the fsm data, repeat the procedure from step 2 for the remaining data.
Step 5. Repeat the procedure from step 2 for the Character Translation memory of the selected subset.
Step 6. Mark this subset as allocated and go back to step 1.

Some clarification is in order as to how the memory allocation is performed in *step 2* when an fsm memory is too large to fit in a specific block. An fsm memory can be too large to fit in two different occasions:

a) The selected FPGA block enough free data bits but doesn't have a large enough address range. E.g. we have an fsm memory of 8K x 7 bits and the selected FPGA memory is 2K x 9 bits. In this case the fsm memory is partitioned length-wise, i.e. we place the first 2K addresses of the fsm memory in this FPGA block, the following 2K addresses in another block etc. All selected FPGA blocks must have the same number of address bits.
b) The selected FPGA has a large enough address range but not enough free data bits. In this case the fsm memory is partitioned breadth-wise; we store as many data bits of the fsm memory in this block as we can and allocate the remaining data bits in other blocks. All selected blocks must have an address range greater or equal to the fsm memory's range.

Another significant detail is that the 18 Kbit FPGA memory blocks are constrained as to how wide the resulting memory can be. The supported configurations are 512 lines of 36 data bits, 1K lines of 18 bits, 2K lines of 9 bits, 4K lines of 4 bits, 8K lines of 2 bits and 16K lines of 1 bit. Since 2K data bits are lost when the address range is greater than 2K lines, we utilize only the first three configurations for maximum space utilization.

5.8.1 Memory Allocation Examples

A simplified example of the allocation of several small fsm memories in a single FPGA memory block can be seen in figure 5.11.

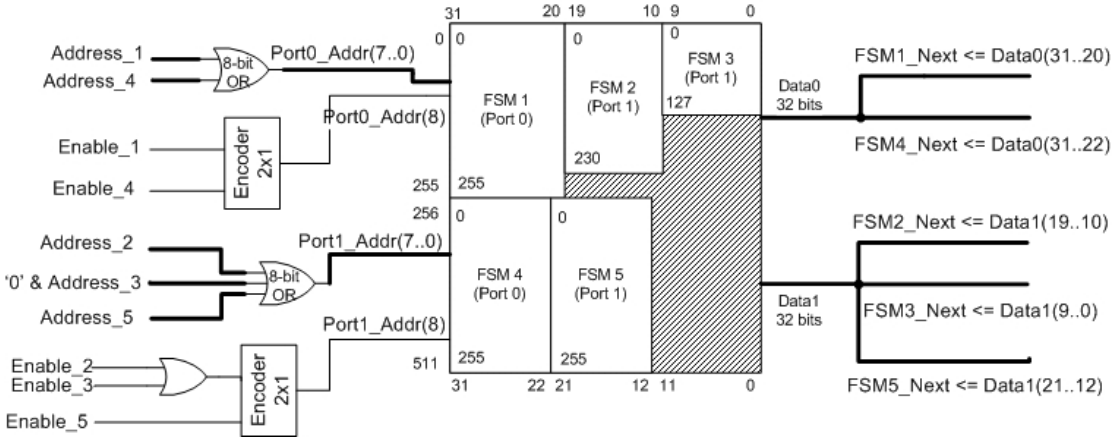

Fig 5.11: Allocation of 5 small fsm memories in a single FPGA memory block.

In this example we have 5 small fsm memory segments, numbered FSM 1 to FSM 5, allocated in a single FPGA block. Right after the fsm number we have written the number of the port which is responsible for the access of that specific data. The numbers written next to the vertical sides of the FPGA block represent addresses, while the numbers on the horizontal sides represent data bits. We can see that memory FSM 1 is allocated to port 0 and has 256 lines of 12 data bits, FSM 2 is allocated to port 1 and has 230 lines of 10 data bits etc. The grayed out areas of the FPGA block represent unused space. The OR gates and encoders necessary to calculate the read address of a port represent the *ingress logic* of that port.

Signals "Address_#" are the requested read addresses from each fsm. Signals "FSM#_Next" represent the data to be sent back to the corresponding fsms. Signals "Port0_Addr" and "Port1_Addr" are the read addresses of the memory blocks two ports, while signals "Data0" and "Data1" are the results of the read operation for each of the two ports.

In the previous section we described how the rule subsets (and, by extension, the memory segments) allocated to each port are mutually exclusive, meaning that at most one will be active at any time. By extension, at most one "Address_#" signal for each port will be nonzero, which is why we can simply OR together the "Address_#" signals. A slight complication arises for memory segments that require less address bits than their neighbors. In our example, segment FSM 3 has only 128 lines (requiring 7 address bits) while all other segments need 8 address bits. The answer to this problem is to zero-pad the beginning of the input address for each smaller segment so as to have the exact number of bits as the largest segment contained in the FPGA memory block.

Since the minimum address range for an FPGA block is 512, whenever a block is used to hold memory segments with 256 or fewer lines we divide it horizontally into "segment rows", as is the case in this example. The number of segment rows is equal to 512 divided by the address range of the largest memory segment (2 in our example). In order to select between different segment rows, we use the "enable" signals of the different fsms and an appropriate encoder (a 2x1 encoder if we have two segment rows, 4x2 for four rows etc.). The "enable" signals of all memory segments that belong to a specific segment row are OR-ed together and the

result goes to the segment row's equivalent input of the encoder. Since at most one of the "enable" signals will be non-zero, the encoder's output provides the most significant bits of the read address for that specific port. The required OR gates and encoders for a specific memory block's read port represent the *ingress logic*.

In our example, for the memories allocated to read port 1, FSM 2 and FSM 3 are in the first segment row (FPGA block addresses 0 to 255) while FSM 5 is in the second segment row (FPGA block addresses 256 to 511). Signals Enable_2 and Enable_3 are OR-ed together and led to input 0 of the encoder, while signal Enable_5 is led to input 1 of the encoder. As a result, if FSM 2 or FSM 3 is active the corresponding "Enable" signal will be set to 1, the encoder's output (which is also the read address' MSB) will be 0 and the first segment row will be selected. In a similar manner, whenever FSM 5 is active the second segment row will be selected.

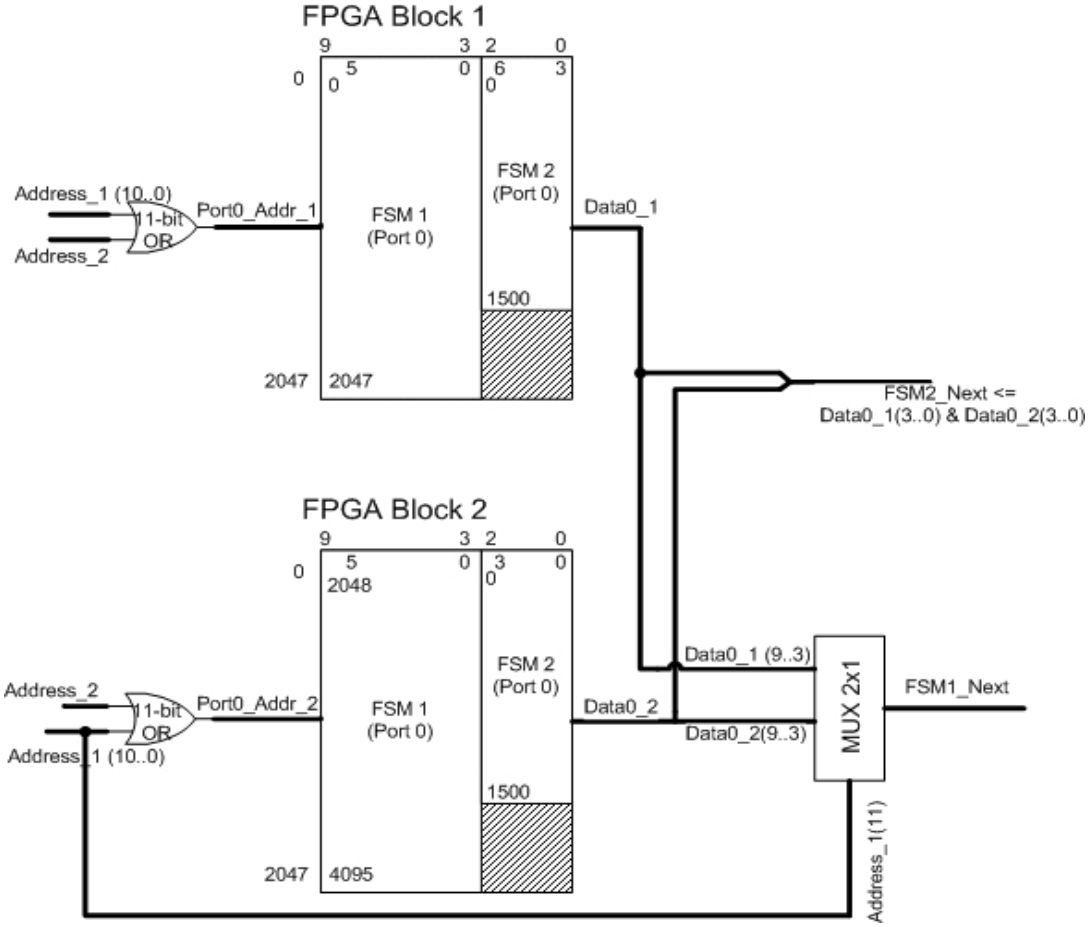

Fig 5.12. Length-wise and Breadth-wise memory distribution. FSM 1 is distributed length-wise, while FSM 2is distributed breadth-wise.

Figure 5.12 shows a different memory allocation example, in which we have both length-wise and breadth-wise distribution of fsm memory segments that are too large to fit in a single FPGA memory block.

In this example, the memory of the first fsm has 4096 lines of 6 bits and has been distributed length-wise. This memory is split across two blocks, with the first block holding lines 0 to 2047 and the second block holding lines 2048 to 4095. This memory requires 12 bits to be completely addressed. The 11 lower bits are OR-ed with the address bits for the FSM_2 memory segment (same reason as in the previous example) and used as a read address in both FPGA blocks. The MSB is used as a control signal in a 2x1 multiplexer: when the MSB is 0, the data that goes to

FSM1_Next is the data read from the first FPGA block, but when the MSB is 1 then the data of the second block is forwarded to the output.

On the other hand, the second fsm memory has 1500 lines of 6 bits and has been distributed breadth-wise. The 3 MSB's of each line (bits 5 to 3) are stored in bits 2 to 0 of the first FPGA Block, while the 3 LSB's (bits 2 to 0) are stored in bits 2 to 0 of the second FPGA block. In this way, whenever we access a line from the FSM 2 memory, the same line from both FPGA blocks is read and the result is the concatenation of the appropriate data bits.

5.9 Implementation

In order to implement the proposed architectures, we relied on a properly modified T-Gate [15][16] platform. The final platform we used is written entirely in C++ code and spans 7 files for a total of approximately 16,000 lines of code.

First of all, the modified T-Gate platform contains a software implementation of the Split-AC algorithm, in order to test the correctness of the algorithm.

The second major operation of the platform is to read all the Snort rule files and classify the rules into different sets based on their headers, as described in section 4.2.2.

Another function of the platform is the breaking-down of large rule groups into smaller subsets in such a manner that none of the Aho-Corasick fsms corresponding to the subsets has more states than a user-specified maximum. The rule group division process is described in detail in section 4.3.

After dividing rule sets into subsets, the platform performs the FPGA memory allocation procedure described in the previous section. Essentially, this procedure aims at assigning as many mutually exclusive fsm memories as can fit in a single FPGA memory block.

Finally, the platform generates the necessary VHDL code for implementing the non-stable components of the architecture presented in chapter 5. By "non-stable" we refer to the components whose contents vary based on the set of rules being implemented and the user-defined parameters. There are only three user-defined parameters: the maximum number of states per fsm, the character occurrence frequency threshold T_f and the revision of the Split-AC algorithm that will be used (0 to 2). The generated non-stable architecture components are stored in the following files:

- **fsm_*i*_(in)sensitive_*j***: the fsm which corresponds to the case (in)sensitive subset *j* of rule group *i*. Also contains the CAM logic for the specific subset. Described in section 5.5.
- **LUX_*x*_ingress_(0,1)**: the ingress control logic corresponding to port 0 or 1 of state lookup memory block *x* (see section 5.8).
- **xlate_*y*_ingress_(0,1)**: the ingress control logic corresponding to port 0 or 1 of character translation memory block *y* (see section 5.8).
- **enable_encoder**: the priority encoder for encoding the enable signals of active fsms and selecting the smallest. Described in section 5.6.
- **final_priority_mux**: the priority multiplexer used to select among all the active results of all fsms. Described in section 5.7.

- **header_classifier**: the component which activates all rule groups that can be applied to an incoming packet. Described in section 5.3.
- **split_central**: the top level of the architecture, as described in section 5.2.

Aside from the above non-stable components, there are several other stable components that are used. The VHDL code for most of these components was written by hand. The only components not written by hand were the memory components: these were generated by using the Core Generator functionality of the Xilinx ISE 7.1.04.

- **comp_eq_(8,16,24,32)**: equality comparator for two 8, 16, 24 or 32 bit wide values.
- **comp_grt_(8,16)**: greater-than comparator for two 8 or 16 bit wide values.
- **counter_16_bit**: 16 bit incremental counter. Used as the "Position Counter" described in section 5.2.4.
- **encoder_(2,4,8,16)_to_(1,2,3,4)**: four priority encoders of different size. In these encoders, the most significant bits have priority. Used for the ingress logic of the FPGA memory blocks and, occasionally, for selecting among state lookup data from different FPGA blocks for fsms with very large state lookup tables.
- **inverse_encoder_16_to_4**: a 16-to-4 priority encoder in which the least significant bits receive priority. Used inside the "final_priority_mux" and "enable_encoder" components.
- **mux_2x1**: single bit wide 2-to-1 multiplexer.
- **mux_n_(2,4,8,16)x1**: four different, variable width multiplexers.
- **reg_(1,4,32)**: registers for holding a single bit, four bits or 32 bits.
- **reg_n**: variable width register component.
- **char_xlator**: this component is necessary when implementing Revision 0 of the Split-AC algorithm. Corresponds to the "Case Translator" subsystem mentioned in section 5.2.4.
- **mem_(512,1024,2048)_to_(36,18,9)**: dual-port memory components generated by using the Xilinx Core Generator. Three different configurations: 512 entries of 36 bits, 1024 entries of 18 bits and 2048 entries of 9 bits. Each configuration corresponds to a single FPGA memory block.

5.10 Validation & Testing

The first thing that needed to be verified was the correctness of the Split-AC algorithm. To that end, we implemented the algorithm is software as part of the T-Gate platform and for every split-AC fsm that was created we also created a standard Aho-Corasick automaton. When testing the algorithm with some synthetic input texts, we used both the split-AC and the regular fsm and found that both performed exactly the same transitions and matched the same patterns.

Afterwards, we verified that the subset division process (i.e. dividing rule groups into subsets) was correct. To accomplish that we created an Aho-Corasick fsm for the initial rule set and one fsm per generated subset. When scanning synthetic inputs, we recorded which patterns were matched by the initial rule set fsm and which

where matched by *all* the subset fsms. Both matched exactly the same patterns in the same text positions.

The last thing we needed to verify was the correctness of the VHDL code we used. All VHDL files were tested using the Modelsim 6.0 simulator. Every component whose code was hand-written was also tested and found to be working correctly. Components that have only one variation in the architecture and whose code was generated by the software platform (e.g. “enable_encoder”, “priority_mux”) were also tested. For “non-stable” components that have many different variations generated in a similar manner (e.g. “fsm_*i*_(in)sensitive_*j*”), testing every single instance would be a waste of time; instead, we tested a couple of the largest and most complex generated instances. Since the large and complex instances functioned correctly, it is only logical that the smaller and simpler instances are also correct.

CHAPTER 6 – RESULTS

6.1 Introduction

In this chapter we present the various results regarding the behavior of the Split-AC algorithm as well as the IDS architecture in which it was implemented. In the first part of the chapter we present the rule configuration that was used. The second part presents numerous interesting results. Finally, the third part is dedicated to potential future work that could arise from this thesis.

6.2 Rule Configuration

The rule set that we used was the unregistered version 2.4 of the Snort rule set [10]. We opted to use the unregistered version since it is freely available: a registered version of the rule set, which contains more rules, is available after a free registration process. We used all the rule files included in that specific rule set and from every rule with more than one patterns we selected the first one. The total number of rules with patterns is 2271, while the total number of characters is 24033.

The proper configuration of the symbolic addresses IP used in the snort rules (e.g. $HOME_NET) is of significant importance, since it directly influences the header classification process and, by extension, the rule grouping process. For example, if every symbolic address was set to "any", the resulting number of rule groups would be much smaller that if every symbolic address is different.

To that end, we configured the symbolic addresses so that we would have a substantially fine-grained rule grouping. Specifically, the IP addresses assigned to each symbolic address were:

- **$HOME_NET**: set to 147.27.3.0/24 (the /24 means that we are interested only in the first 24 bits of the IP address)
- **$EXTERNAL_NET**: set to !HOME_NET, i.e. ! 147.27.3.0/24.
- **$HTTP_SERVERS**: set to 147.27.4.0.
- **$TELNET_SERVERS**: set to 147.27.5.0.
- **$SMTP_SERVERS**: set to 147.27.6.0.
- **$DNS_SERVERS**: set to 147.27.7.0.
- **$SQL_SERVERS**: set to 147.27.8.0.
- **$SNMP_SERVERS**: set to 147.27.9.0.
- **$AIM_SERVERS**: set to the same values used by Snort. These are IP addresses 64.12.24.00/23, 64.12.28.00/23, 64.12.161.0/24, 64.12.163.0/24, 64.12.200.00/24, 205.188.3.0/24, 205.188.5.0/24, 205.188.7.0/24, 205.188.9.0/24, 205.188.153.0/24, 205.188.179.0/24 and 205.188.248.0/24.
- **$HTTP_PORTS**: set to 80.
- **$SHELLCODE_PORTS**: set to !80.
- **$ORACLE_PORTS**: set to 1521.

6.3 Rule Grouping and Subset Division Results

In table 6.1 we present the results from dividing the original rule sets into subsets based on the desired maximum number of states per fsm, as described in section 4.3. The "unrestricted" column corresponds to an unlimited maximum number of states per fsm. Finally, the "Total rule sets before division" value is the number of different rule sets generated by our fine-grained rule grouping method described in section 4.2.2.

TABLE 6.1
Results from rule set division into subsets.

Total rule sets before division: 400							
Initial rule sets with patterns: 322							
Maximum States per fsm	**16**	**32**	**64**	**128**	**256**	**512**	**Unrestricted**
Total Subsets	864	749	599	457	408	386	374
Maximum subsets in one set	124	124	115	42	19	9	2

The reason for having a maximum of 2 subsets even with an unrestricted number of states per fsm is due to the separation of case sensitive and insensitive patterns in two subsets wherever both are present. Also, the fact that the total number of subsets is less than the initial number of sets is because only rule sets with patterns are divided: as a result, even though we initially have 400 rule sets, 400-322=78 rule sets don't contain any patterns to match and are ignored in the division process.

Table 6.2 shows the distribution of subset sizes for several values of the desired maximum number of states per fsm. At this point we should remind that the desired maximum number of states can not always be satisfied: subsets containing patterns which are longer than the maximum value are assigned a number of states equal to the smallest power of 2 that can contain that pattern. The value at the top of each column indicates that all cells in that column have more states than the previous column and up to the number of this column. For example, column '64' holds the number of subsets that have more than 32 and up to 64 states.

TABLE 6.2
Distribution of the subset size in powers of two. "Max X states" indicates that the desired maximum number of states per fsm is X.

	Number of fsms corresponding to every power of 2.													
	2	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**Total fsms**
Max 16 states	27	60	99	301	226	148	3	0	0	0	0	0	0	864
Max 32 states	27	59	88	103	321	148	3	0	0	0	0	0	0	749
Max 64 states	27	59	86	91	67	266	3	0	0	0	0	0	0	599
Max 128 states	27	59	85	88	59	34	105	0	0	0	0	0	0	457
Max 256 states	27	59	85	86	58	30	18	45	0	0	0	0	0	408

Max 512 states	27	59	85	86	58	29	13	8	21	0	0	0	0	386
Undivided	27	59	85	86	58	29	11	8	7	2	1	0	1	374

One interesting observation that can be made from the values of table 6.2 is that in the case of unrestricted maximum number of states per fsm, 344 out of 374 subsets, or 92% of the subsets, require fsms with 64 states or less, while 257 out of 374 subsets, or 68.7% of the subsets, require no more than 16 states per fsm! These conditions make the split-AC algorithm a very promising solution, since it performs best when the fsms have a small number of states and, by extension, use a small number of characters.

A second observation is that when the maximum number of states is set to the very small value of 16, 377 out of 864 subsets, or 43.6% of the subsets, need an fsm of at least 32 states, i.e. larger than the maximum number of states. This is due to the fact that long patterns cannot fit in fsms with a smaller number of states that the pattern length. In essence, this result indicates that a large number of rule sets cannot be divided into as small fsms as we would desire.

6.4 Split-AC Behavior

In this section we present in detail the behavior of the various aspects of the Split-AC algorithm under varying parameters. We show the simulation results for several different values of the maximum number of states per fsm as well as the character occurrence frequency threshold T_f. We use several values for the T_f parameter, which range from 0.01 to 0.99.

We should remind that the character occurrence frequency threshold T*f* is the parameter controlling which characters are considered as "frequent" and which characters are "infrequent": a character appearing in transitions in more than T*f**100% of the fsm's states is "frequent", while those below the threshold are "infrequent". A large threshold value means fewer "frequent" and more "infrequent" characters, resulting in larger CAMs and smaller memories. As such, the 0.01 threshold value means that almost all characters appearing in an fsm are "frequent", while the 0.99 threshold means that only characters appearing in practically every state are "frequent".

6.4.1 Split-AC Memory Requirements

Figure 6.1 shows the minimum required memory for the initial split-AC algorithm (revision 0), under the condition that there is no limitation on the number or size of memories that we use (i.e. we can have any number of memories, as small or as large as we like). This could be the case if we were implementing the algorithm in ASIC.

Figure 6.1 shows clearly that as the maximum number of states per fsm decreases, i.e. as we break the rule sets into smaller and smaller subsets, the required

memory also decreases. This decrease, however, seems to taper out at 32 states per fsm, since the difference in memory requirements when going from a maximum of 32 to 16 states per fsm is very small (1.3% to 1.8%).

Figure 6.1 also verifies the expected behavior of the Split-Ac algorithm with regards to the character occurrence frequency threshold T_f. As T_f increases, fewer characters fall into the "frequent character" category, which results in smaller character translation memories and state lookup memories with fewer transitions per state. Of course, the tradeoff for this memory decrease is that a larger amount of logic will be required to handle the "infrequent" characters.

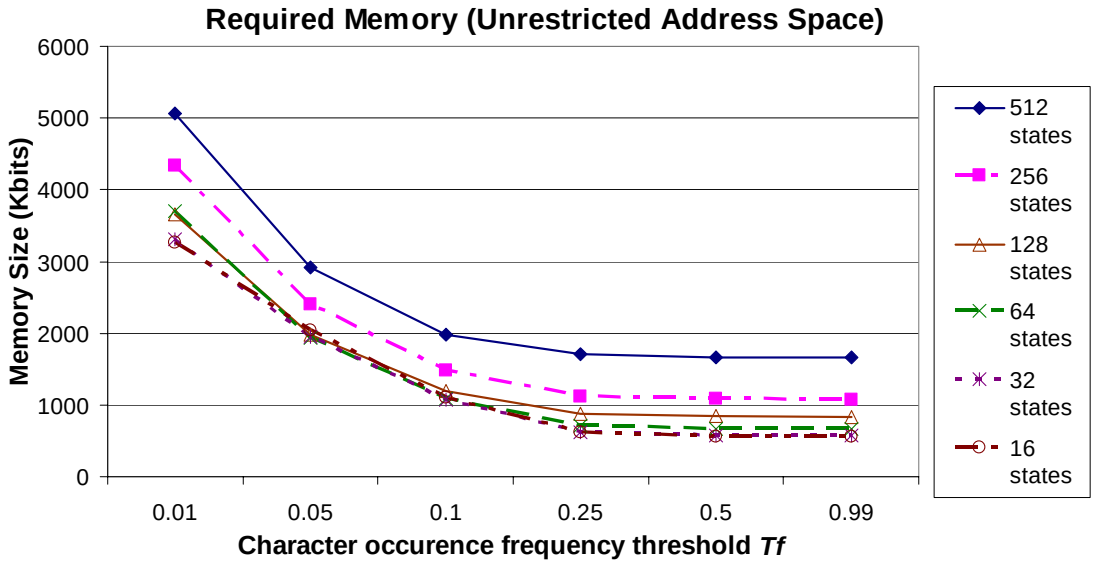

Fig. 6.1. Original Split-AC memory requirements when the address space of the memories is not restricted to powers of 2

One final observation of interest is threshold T_f values between 0.25 and 0.99 have a very small impact in memory requirements, while there is a very large difference in memory for values between 0.01 and 0.1.

The values in figure 6.1 assume that the memories we use can have any number of addresses, for example 150 addresses, and are not limited to powers of 2. The corresponding results in the case where the address space for every memory is a power of 2 can be seen in figure 6.2.

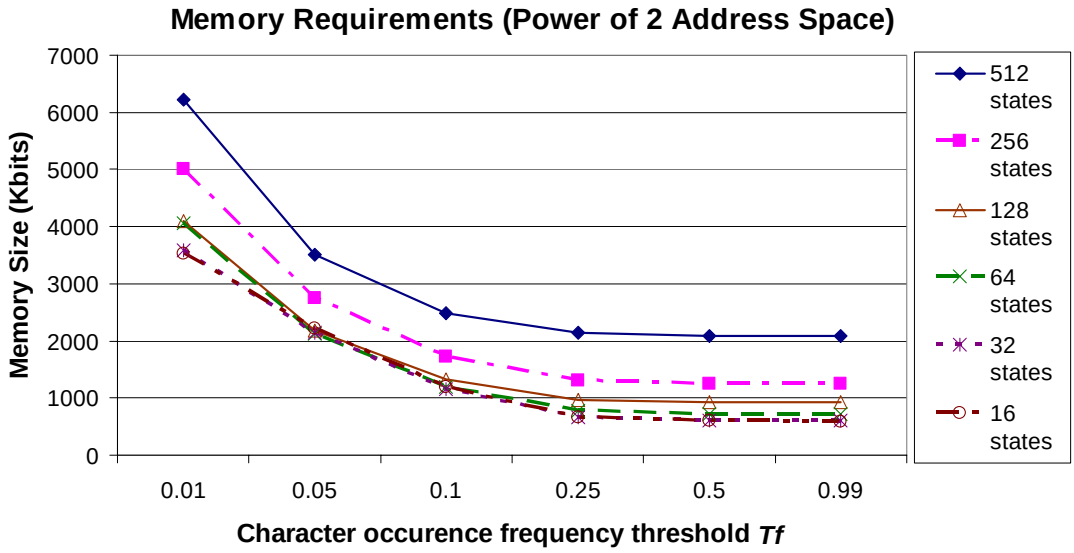

Fig. 6.2: Original Split-AC memory requirements when the address space of the memories is restricted to powers of 2.

We can see in figure 6.2 that, even though the required memory increases in every case, the behavior of the algorithm remains the same. At this point we must mention that this increase in memory size is owed solely to the "state lookup" memories, because the "character translation" memories are always aligned to a power of two since they have 256 entries. On the other hand, a state lookup memory may, for example, contain 100 states with 32 transitions each: if the address pace is unrestricted then a memory with 100*32 = 3200 addresses can be used, but when limited to powers of two the memory would have $2^{12} = 4096$ addresses. The precise percentage of memory increase for every case is shown in table 6.3

TABLE 6.3
Percentage of memory increase when the address space for every memory must be a power of 2.

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	22.87	20.58	25.13	25.65	25.42	25.46
256 states	15.59	14.36	16.32	16.59	16.21	16.25
128 states	11.84	10.32	10.58	10.23	10.32	10.35
64 states	9.9	9.65	9.46	8.2	8.1	8.25
32 states	8.67	9.25	8.9	6.85	6.62	6.6
16 states	8.32	8.68	8.27	6.28	6.04	6

Table 6.3 shows that the total required memory can be increased by 6% to 25.65% when the address space for every memory is restricted to powers of 2. One other observation is that the percentage of memory increase becomes smaller as the maximum number of states per fsm decreases. The logical explanation for this behavior is that by using smaller fsms we have smaller chunks of unused space.

6.4.2 Character Translation and State Lookup Memories

Table 6.4 shows the exact size of the character translation memories for different parameter values, while tables 6.5 and 6.6 respectively show the size of the state lookup memories when the address space is unrestricted and restricted to powers of 2.

TABLE 6.4
Total size of character translation memories (in Kbits).

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	321.25	293	247.25	164.25	143.5	140.75
256 states	348.5	314.25	261.5	175	154.25	151.5
128 states	406.25	358	291.25	196.75	175	172.25
64 states	575.25	485.25	373	247	221.75	219
32 states	707	610.5	461	289.5	258.5	254.75
16 states	802	720	527.75	315.75	279	274.5

When observing the behavior of the character translation memory size, we can see that the total required memory actually decreases as the maximum number of states increases. The reasons for this become apparent with a simple example: if we have two subsets of the same set and each has 31 characters in the character translation table, the required space for the character two translation tables is 2*256*5=2560 bits. If these two subsets are joined into one set with a maximum of 31+31=62 characters in the translation table, the required character translation space now becomes 256*6=1536 bits. The subset division process may decrease the space required for state lookup memories, it nevertheless increases the space required for character translation memories.

The second observation that can be made from table 6.4 is that the space required for character translation decreases as the frequency threshold increases. This behavior is to be expected: a larger threshold value means that fewer characters are considered frequent and, by extension, the character translation tables are smaller.

TABLE 6.5
Total size of state lookup memories when the address space is not restricted to powers of 2 (in Kbits).

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	4736.98	2617.57	1734.39	1543.41	1517.72	1516.67
256 states	3987.65	2093.06	1215.7	944.221	929.08	928.025
128 states	3258.65	1620.2	907.775	684.916	665.053	663.998
64 states	3124.44	1447.9	712.553	465.484	442.968	441.873
32 states	2600.29	1345.05	608.871	336.625	315.961	314.244
16 states	2462.47	1321.06	574.602	307.346	286.141	284.143

TABLE 6.6
Total size of state lookup memories when the address space is limited to powers of 2 (in Kbits).

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	5893.67	3216.67	2232.29	1981.43	1939.92	1938.7
256 states	4663.67	2438.67	1456.79	1129.93	1104.67	1103.45
128 states	3692.67	1824.42	1034.67	775.121	751.73	750.512
64 states	3490.61	1634.36	815.23	523.918	496.777	495.559
32 states	2887.01	1526.01	704.137	379.543	353.98	351.777
16 states	2734.23	1498.23	665.855	346.499	320.262	317.684

Tables 6.5 and 6.6 show the consistent behavior of the algorithm for varying parameter values. Specifically, we observe that for a decreasing maximum number of states per fsm the required state lookup memory size also decreases, as expected. The state lookup memory size also decreases when the threshold is increased, which is owed to the fact that more characters are removed from the state lookup tables and assigned to the infrequent character logic.

The data presented in the tables above allows us to plot the percentage in which state lookup memories contribute to the total memory requirements, which is shown in figure 6.3. The contribution percentage of the character translation memories in each case can be found by subtracting the state lookup percentage from 100.

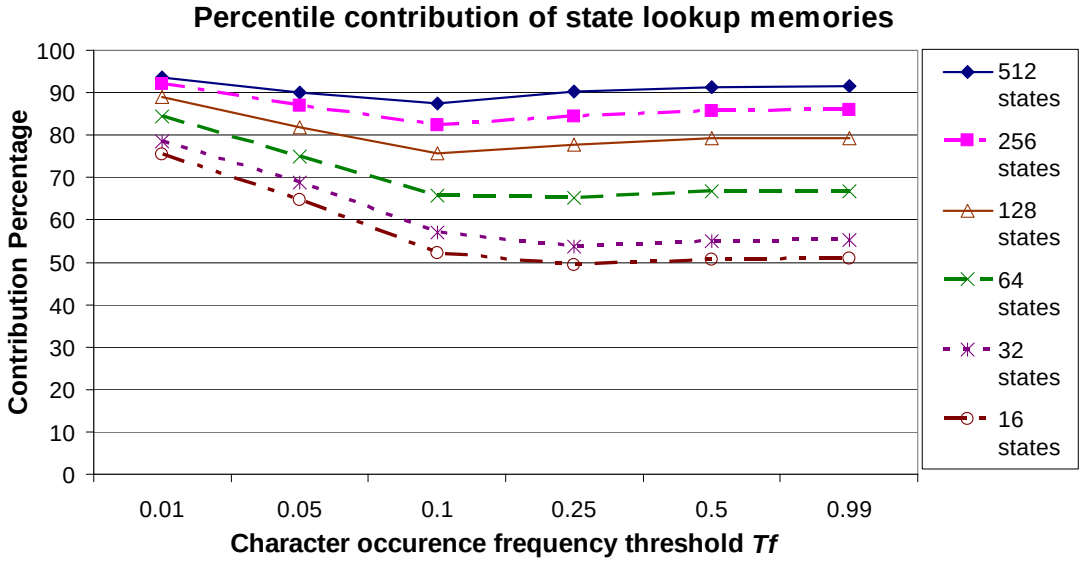

Fig 6.3: The percentage in which state lookup memories contribute to the total memory requirements of Split-AC

It becomes apparent from figure 6.3 that the percentile contribution of state lookup memories increases along with the maximum number of states per fsm. We also observe that the percentage initially drops as the T_f threshold increases from 0.01 to 0.1 and then increases again as the threshold increases from 0.1 to 0.99. The initial percentage decrease is owed to the fact that, for those threshold values, the state

lookup memory size decreases more rapidly than the character translation memory size, a situation that is later reversed for larger threshold values.

The percentile contribution doesn't change significantly when we limit the memory address space to powers of 2. The only difference is a 1% to 3% increase in the contribution percentage of the state lookup memories

6.4.3 Frequent Characters per FSM State

Figure 6.4 shows the average number of frequent characters per fsm state, while figure 6.5 shows the corresponding maximum number of frequent characters in an fsm state. We remind that the transition data for frequent characters is held in the state lookup table.

It is interesting to note that for a threshold value of 0.01, i.e. when practically every character is considered frequent and its transition data is maintained in the state lookup table, the average number of frequent characters per fsm is no more than 13. This indicates that, on average, the state lookup tables will hold only 16 transitions (the power of 2 immediately greater than 13) per instead out of the 256 transitions that the original Aho-Corasick stores. This represents an approximate reduction of required space by 93.75 %. On the other hand, if we were to select a threshold value of 0.99 then the average number of frequent characters is no more than 2.8, which translates to storing 4 transitions per state instead of 256 or an approximate state lookup space reduction of 98.4375 %.

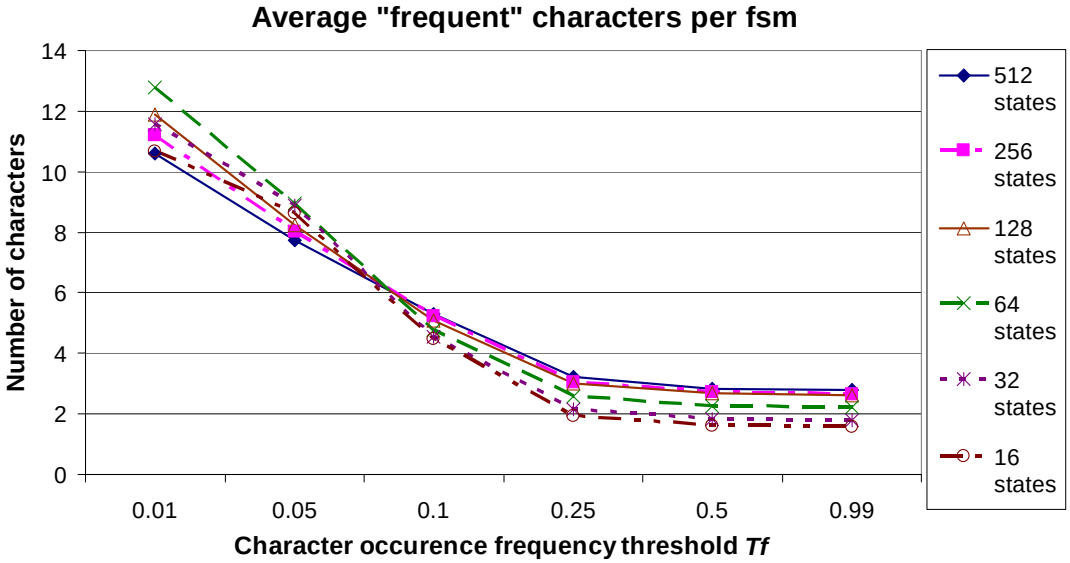

Fig. 6.4: Average number of frequent characters per fsm.

Figure 6.5 shows the maximum number of frequent characters found in any fsm. The figure shows clearly that the maximum number of frequent characters decreases along with the maximum number of states per fsm. It is also interesting to note that in the worst case (i.e. the case with the most frequent characters) we have no more than 63 frequent characters, which means that the largest state lookup table holds no more than 64 transitions per state and, as such, needs approximately 75% less space than the traditional AC algorithm. On the other end, for a threshold value of 0.99, the largest state lookup table will store from 32 to as little as 8 transitions per

fsm state, resulting in an approximate state lookup space reduction from 87.5 % up to 96.875 %.

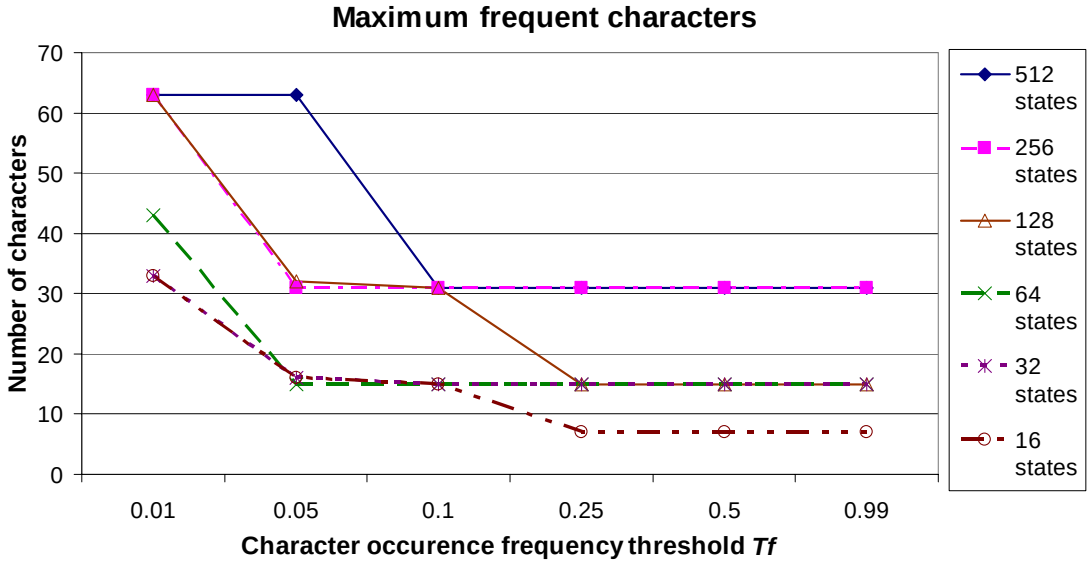

Fig. 6.5: Maximum number of frequent characters in any fsm.

6.4.4 Infrequent Character Requirements

Figure 6.6 shows the average number of infrequent character transitions per fsm, while figure 6.7 shows the maximum number of infrequent character transitions in any fsm. Note that we are counting the infrequent transitions, not simply the infrequent characters. For example, if an infrequent character *c'* appears in 3 different transitions in an fsm then we count 3 infrequent transitions and not one infrequent character.

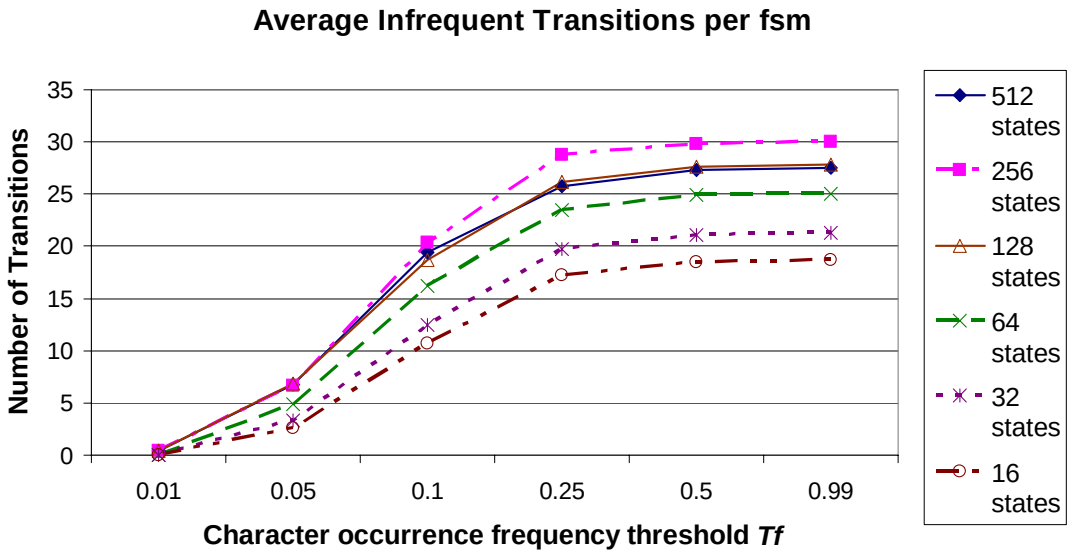

Fig. 6.6: Average number of infrequent transitions per fsm

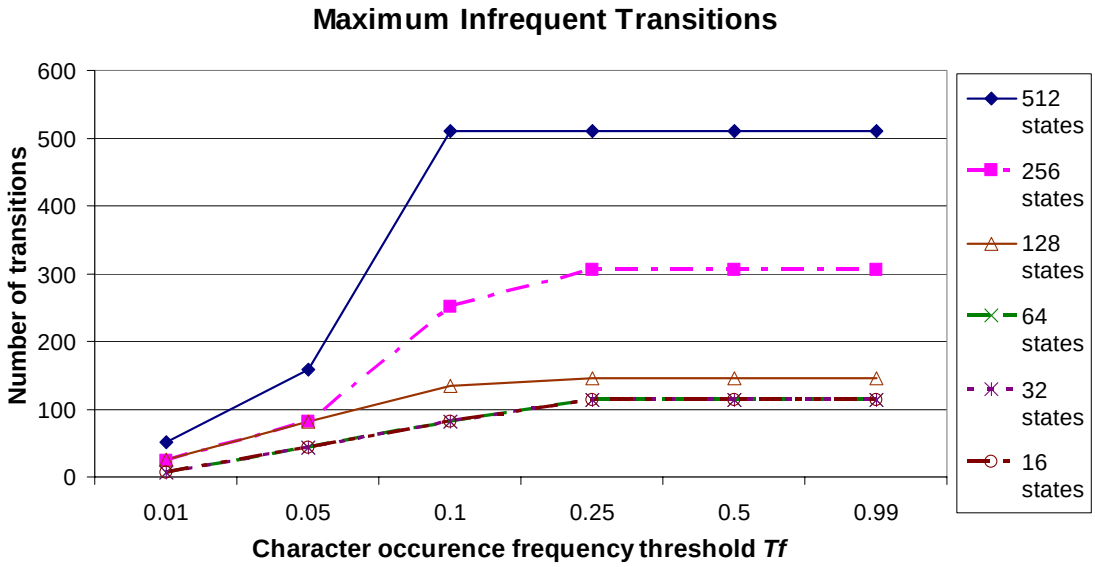

Fig. 6.7: Maximum number of infrequent transitions in an fsm.

We can observe that, in general, both the average and the maximum number of infrequent transitions increases along with the number of states. If we assume that the total number of infrequent transitions is roughly independent of the total number of subsets, then as the maximum number of states increases, the same number of infrequent transitions is divided across fewer subsets, resulting in an increase of infrequent transitions.

The assumption we made may be simplistic but it is not far from the truth: when the subsets get larger, a few characters that were previously infrequent in the small subsets may now appear in a larger frequency, thereby becoming frequent and being removed from the infrequent transitions. However, this decrease of infrequent transitions is usually smaller in proportion to the decrease of subsets. The only case where this phenomenon becomes apparent is the average number of infrequent transitions when going from a maximum of 256 states to 512 states per fsm.

What is of more interest is the variation of the average number of infrequent transitions for different frequency threshold values. For a threshold of 0.01, the number of infrequent transitions is between 0.016 (16 states max.) and 0.425 (512 states max.) transitions per fsm, a very small value. Of course, this comes at the cost of large state lookup memories. For a threshold value of 0.05, the average number of infrequent transitions is between 2.56 and 6.75 transitions per fsm, a relatively manageable number. As the threshold further increases and approaches 1, the number of transitions reaches 18.66 to 30.03 transitions per fsm: the logic cost to support such a number of infrequent transitions will be significant.

In our architecture for implementing the Split-AC algorithm (section 5.5) we decided on using a CAM like structure to handle the infrequent transitions. As we mention in section 3.4, the resource cost of a CAM is roughly proportionate to the total number of CAM entries times the input bits being compared. The total CAM tag size required for each configuration of the input parameters is shown in table 6.7.

TABLE 6.7

Total CAM input (tag) size for the initial version of the Split-AC algorithm (in Kbits).

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	2.66	40.6172	117.684	151.72	160.393	161.506
256 states	2.33	40.6768	125.359	174.152	179.604	180.717
128 states	2.51	45.1631	122.306	169.357	178.11	179.224
64 states	0.21	40.1309	131.806	189.569	200.465	201.578
32 states	0.21	32.9727	124.14	194.557	206.748	208.55
16 states	0.21	29.6084	121.298	193.513	206.852	208.946

From the values of table 6.7, it is interesting to note that while for small threshold values, 0.01 to 0.05, the required CAM size increases along with the maximum number of states, for large threshold values the situation is reversed. It is also interesting to note that for a threshold of 0.01 a minimal amount of logic is required, from 0.21 to 2.66 Kbits, while CAM size for threshold values larger than 0.1 increases to as much as 209 Kbits. The threshold value of 0.05 seems to be the middle ground, since only a few tens of CAM Kbits are required in that case.

Aside from the CAM input or tag bits, we also felt it necessary to measure the required number of CAM data size, i.e. the bits needed to store the result for every CAM entry. While not as critical a resource as the CAM tag bits, the CAM data bits should also be taken into consideration. The size of the CAM data for every case is shown in table 6.8, in which we can see that the total CAM data size follows a similar behavior to the total CAM input size.

Table 6.8

Total CAM data size (in Kbits)

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	1.4	20.3	59	74.2	78.1	78.5
256 states	1.2	19.6	60.4	82.4	84.6	85
128 states	1.2	20.7	55.7	75.9	79.6	79.9
64 states	0.1	17.1	55.9	79.5	83.9	84.3
32 states	0.1	13.6	51.1	78.1	83.6	84.2
16 states	0.1	12.3	49.4	77.3	82.3	83

6.5 Algorithm Revision Impact

Figures 6.8, 6.9 and 6.10 show the effect of the algorithm revisions 1 and 2, described in sections 3.5.1 and 3.5.2, on the total required memory and the total CAM input size.

In figures 6.8, 6.9 and 6.10, “Rev. 1” of the algorithm is when the infrequent characters are also translated and “Rev. 2” is when the states with infrequent transitions are grouped together and the common state prefix is ignored. We remind

that the changes for revision 2 are made on top of revision 1 and, as such, include the benefits and disadvantages of revision 1. All results are normalized to 1, where 1 is the corresponding memory or CAM requirements of the initial algorithm (revision 0). Any value less than 1 represents an improvement over the original algorithm.

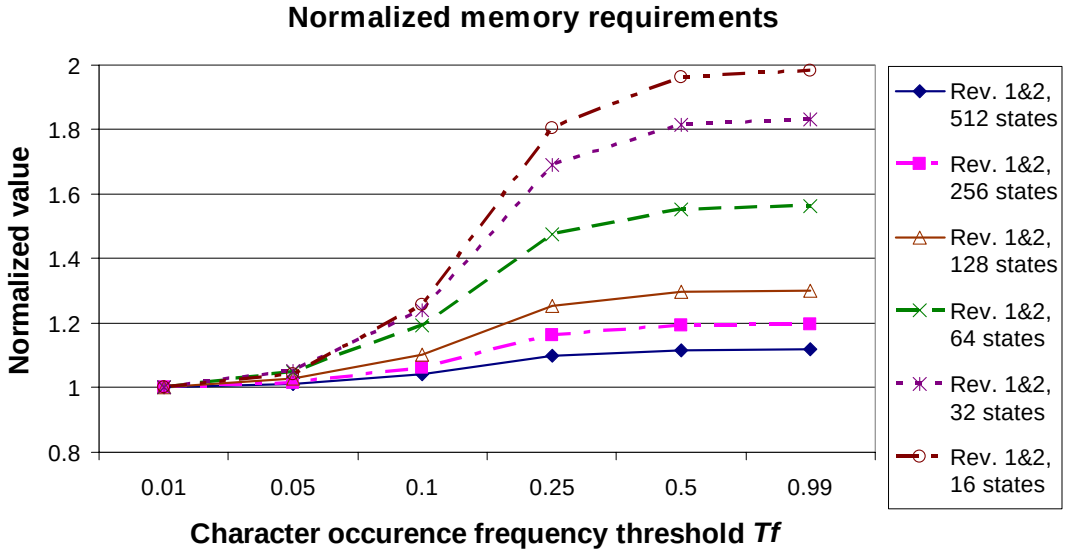

Fig. 6.8: Memory requirements for revisions 1 and 2. Normalized against the CAM requirements of the original Split-AC algorithm

Revision 1 is based on including the infrequent characters in the character translation tables. This may help decrease the CAM size, but it also increases the size of the character translation tables, especially for large values of the frequency threshold: for such values, the character translation tables in the original Split-AC algorithm would be very small. In figure 6.8 we can observe exactly how the total required memory increases along with the threshold value, to a point of nearly twice the initial required memory when the maximum number of states is 16.

We also observe that the increase in memory becomes less profound as the maximum number of states per fsm increases: from a near doubling of required memory for 16 states, we go to a simple 11.7% increase for 512 states. This is owed to the fact that, for a small number of states per fsm, the character translation tables have a larger contribution to the total memory requirements. In figure 6.3, we can see that for 512 states per fsm, the character translation memories contribute only 8.5%, while the contribution percentage for 16 states per fsm is 40.1%. Therefore, it is only logical that an increase in the size of character translation memories will have a greater impact as the maximum number of states decreases.

However, it is interesting to note that for very small threshold values, the memory increase is minimal: for a 0.01 threshold the memory increase is less than 0.2%, while a threshold of 0.05 leads to a memory increase between 1.2% and 5.1%.

Finally, both revision 1 and 2 have the exactly the same memory requirements because algorithm revision 2 utilizes the changes made in revision 1 but doesn't require any additional memory.

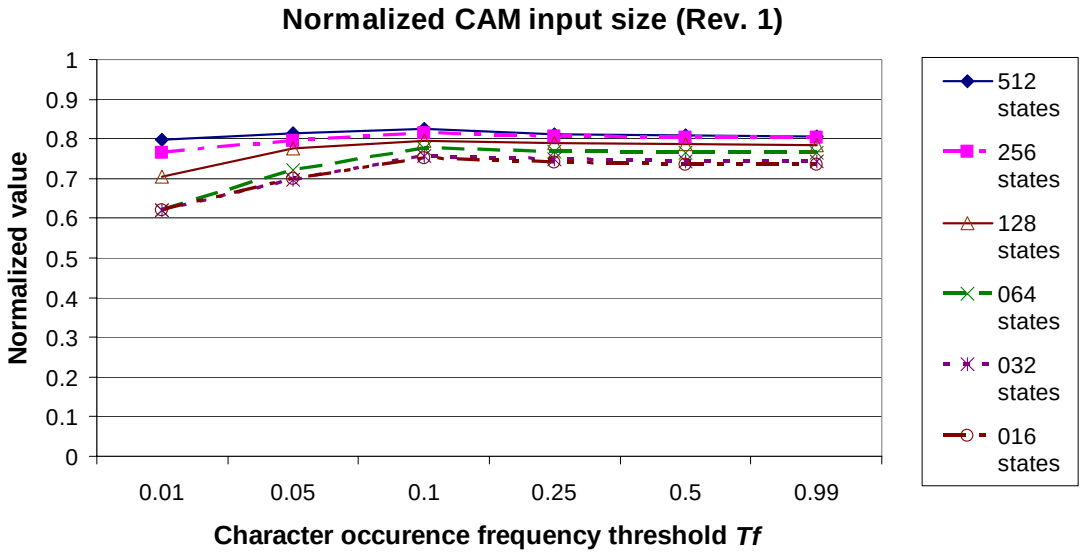

Fig. 6.9: CAM input size for algorithm revision 1 (infrequent character translation). Normalized against the CAM requirements of the original Split-AC algorithm

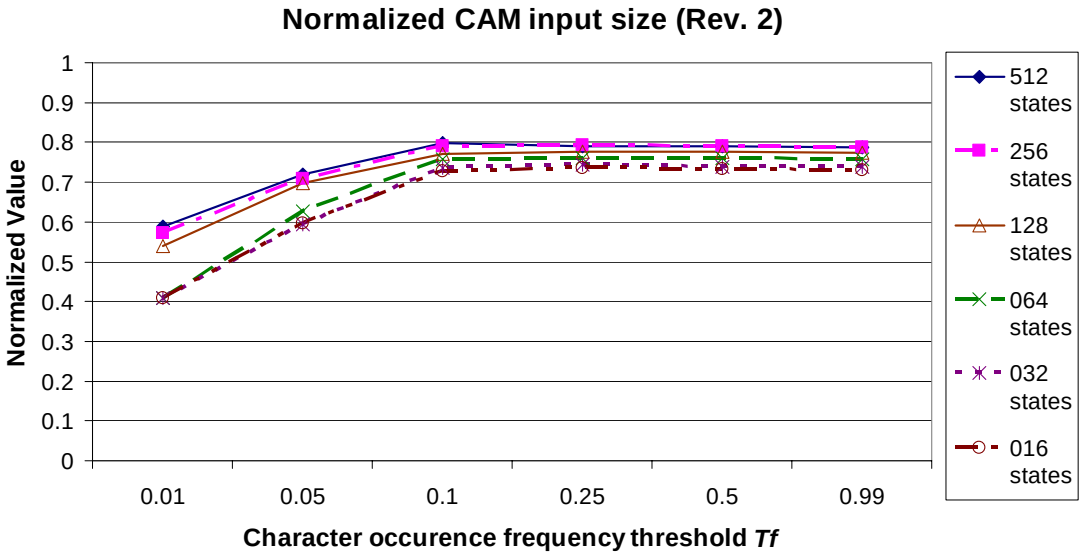

Fig. 6.10: CAM input size for algorithm revision 2 (regrouping states with infrequent transitions). Normalized against the CAM requirements of the original Split-AC algorithm

In figure 6.9 we can notice that the required CAM size is reduced in all cases by a minimum of 17.5% (for 512 states) to 25% (for 16 states). In case that revision 2 is used, as shown in figure 6.10, that reduction percentage respectively becomes 20.3% to 26.6%.

Another observation is that the CAM size reduction from both algorithm revisions increases as the maximum number of states decreases. Also, we can see that for threshold values of 0.1 and higher, revision 2 doesn't offer significant improvement over revision 1.

The impact of both algorithm revisions becomes apparent for small values of the frequency threshold, specifically for 0.01 and 0.05. For a threshold value of 0.01,

algorithm revision 1 offers a CAM size reduction of 20% (512 states) to 38% (16 states), while revision 2 reduces CAM size from 32% (512 states) to as much as 59% (16 states). On the other had, when the threshold is 0.05 the CAM space saved by revision 1 is 19% to 30%, while the CAM space saved with revision 2 ranges from 28% to 40%.

In short, the algorithm revisions prove to be a lucrative solution for small threshold values, since they significantly reduce required CAM space without increasing memory requirements by much. For larger values of threshold they can reduce CAM size by 20-25% but with an occasionally substantial increase in required memory. We decided on using revision 2 of the algorithm in our FPGA implementation, because we consider that reducing CAM size is more important than the increased memory trade-off.

Table 6.9 shows the exact size of the required memory when using the revised algorithms, while tables 6.10 and 6.11 show the size of the CAM for algorithm revision 1 and 2 respectively.

TABLE 6.9
Total required memory for algorithm revisions 1 and 2 (in Kbits).

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	5061.48	2943.07	2061.64	1874.16	1852.97	1852.42
256 states	4340.15	2446.81	1571.2	1303.221	1292.58	1292.025
128 states	3670.9	2034.2	1323.525	1104.166	1088.803	1088.248
64 states	3700.44	2026.65	1293.803	1050.484	1032.468	1031.873
32 states	3308.04	2056.05	1326.871	1058.625	1042.461	1041.494
16 states	3265.22	2126.81	1385.1	1125.35	1109.14	1107.89

TABLE 6.10
Total CAM input (tag) size for algorithm revision 1 (in Kbits).

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	2.125	33.0508	97.126	123.042	129.535	130.169
256 states	1.78125	32.3887	102.139	140.543	144.326	144.96
128 states	1.77	35.0205	97.3486	133.516	139.927	140.561
64 states	0.13	28.9385	102.285	145.729	153.327	153.961
32 states	0.13	22.9375	94.1006	145.807	153.878	154.894
16 states	0.13	20.7041	91.009	143.428	152.214	153.409

TABLE 6.11
Total CAM input (tag) size for algorithm revision 2 (in Kbits).

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	1.566	29.1826	93.8066	119.986	126.59	127.233

256 states	1.336	28.8	98.8916	137.961	141.702	142.346
128 states	1.35645	31.4541	94.4404	131.511	138.136	138.79
64 states	0.086	25.1152	99.7227	144.325	152.246	152.89
32 states	0.086	19.6064	91.3994	144.516	152.921	153.957
16 states	0.086	17.6895	88.3018	142.17	151.34	152.566

6.6 FPGA Implementation

From the FPGAs available by Xilinx, the Virtex4 family was chosen due to the large amount of resources they contain and the increased speed they offer. Specifically, device vc4vlx160 was selected from the Virtex 4 family because it provides a large number of memory blocks (288) and the best speed grade (-12). The other device types, fx and sx, where not considered because we didn't require the DSP or PowerPC capabilities they offer.

6.6.1 FPGA Memory Allocation

As we mentioned earlier in section 5.8, memory allocation was a significant problem challenge when trying to implement our IDS architecture in FPGAs. Figure 6.11 shows the actual amount of FPGA memory used when implementing revision 2 of the Split-AC algorithm.

For a character occurrence frequency threshold of 0.01 we can see that we get the smallest number of memory blocks when the maximum number of states is 16 or 32. However, for larger threshold values we can see that the smallest memory requirements are for 128 states per fsm which is contrary to the behavior of the ideal memory requirements.

The reason for this abnormal behavior is that a small maximum number of states per fsm means that large sets must be divided into a large number of subsets. Since subsets of the same set are never mutually exclusive between themselves they must be allocated in different FPGA memory blocks. As a result, we end up having several near-empty FPGA memory blocks where each holds only a small amount of useful memory, hence the observed increase in required memory blocks.

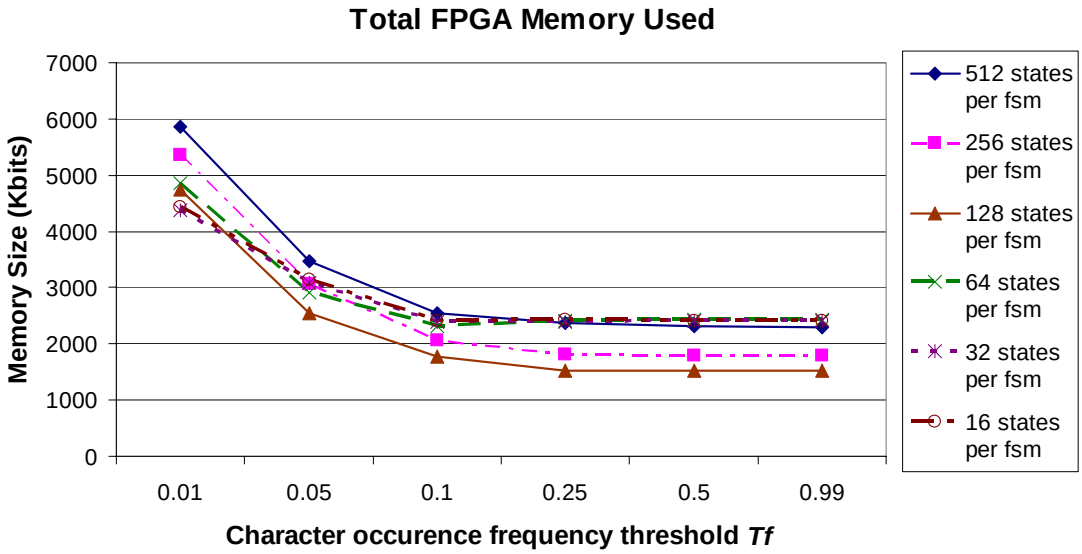

Fig 6.11: Total used FPGA memory when implementing algorithm revision 2.

When the maximum number of states changes from 128 to 256, we can see that the required memory again increases. This indicates that the increased memory requirements due to the increase in array size are greater than any possible waste of memory from "incompatible" subsets. These results lead us to the conclusion that a maximum of 128 states per fsm is in most cases the optimum solution when it comes to used memory.

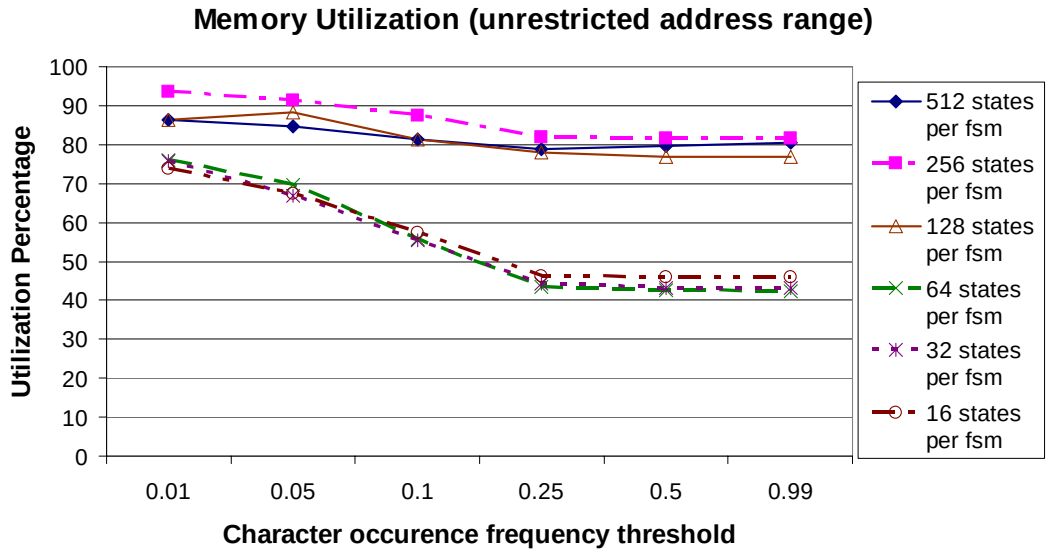

Fig 6.12: Memory utilization percentage when comparing with the absolute minimum of required memory.

Figure 6.12 shows the memory utilization percentage. This is calculated by dividing the required memory when there are no address space restrictions with the actual needed FPGA memory. When the number of states is 64 or less, we observe

that the utilization percentage starts off at approximately 75% for a threshold of 0.01 and later reaches a relatively low value, between 40% and 45%. On the other hand, when we have 128 or more states per fsm the utilization percent is always more than 76%, reaching a maximum of 93.5 % for 256 states and 0.01 frequency threshold.

This figure also shows that the utilization percentage declines as the frequency threshold increases and practically levels off when the threshold surpasses 0.25. This is to be expected if we consider the impact of the frequency threshold: a large threshold value means that few characters are considered "frequent" and so the fsm memories are small, while a small threshold value has the opposite effect. If the fsm memories are small, then many could fit in a single FPGA memory block, which means that we must find many mutually exclusive rule subsets (no small feat!). On the other hand, when each fsm memory is (relatively) large, we need find only a couple of mutually exclusive subsets to fill each FPGA memory block.

An interesting phenomenon canbe observed when we use the restricted-to-powers-of-2 memory requirements, as shown in figure 6.13: when having a maximum of 512 states per fsm, the memory utilization percentage can actually surpass 100% and reach up to 105%!

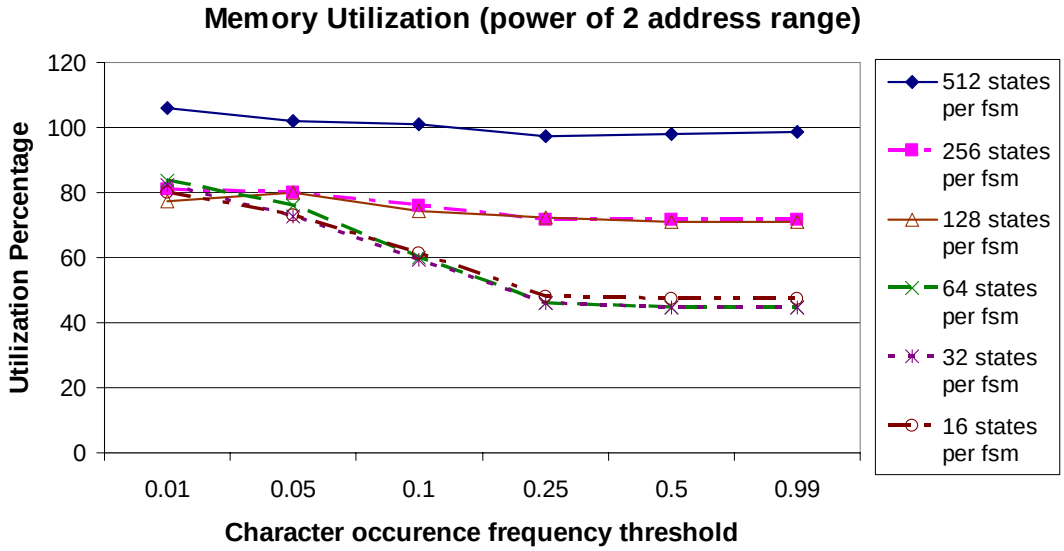

Fig. 6.13: Memory utilization percentage by comparing used FPGA memory with the memory requirements when address range for all memories is restricted to powers of 2.

This artifact is not due to some error on our behalf; it can be explained with a simple example. Consider an fsm with 300 states and 64 transitions per state. If the memory address space is restricted to powers of 2, then a memory with 512*64=32,768 entries would be allocated, even though this state lookup memory actually needs only 300*64=19,200 entries. On the other hand, when partitioning state lookup memories across FPGA memory blocks we allocate chunks of 2048 state lookup memory entries per FPGA block. In this way, the entire state lookup memory can be partitioned among 10 FPGA blocks containing a total of 20,480 entries, a significant improvement over the theoretic memory requirements of 32,768 entries.

Finally, table 6.12 the difference in used FPGA memory when using the initial verion of the algorithm instead of revision 2. In that table, a negative value means that the initial algorithm (revision 0) uses less FPGA memory than revision 2.

TABLE 6.12
Difference in used FPGA memory between Revision 2 and the original Split-AC algorithm (in Kbits).

	Frequency Threshold T_f					
Max number of states per fsm	**0.01**	**0.05**	**0.1**	**0.25**	**0.5**	**0.99**
512 states	-18	-36	-72	-162	-162	-162
256 states	0	-36	-90	-180	-180	-180
128 states	-18	-36	-36	-36	-36	-36
64 states	0	0	0	0	0	0
32 states	0	0	0	0	0	0
16 states	0	0	0	0	0	0

The first conclusion arising from table 6.12 is that for 64 or less states per fsm, the amount of used FPGA memory is the same for the original Split-AC algorithm and revision 2. In the case of 128 states per fsm, the difference is no more than 36 Kbits or 2 memory blocks. For a larger maximum number of states, the difference between the two versions becomes more profound and reaches a maximum of 180 Kbits.

Since we get the optimum amount of used FPGA memory for a maximum of 128 states per fsm or less and the difference of used memory among the two versions of the algorithm is so small, our choice of implementing revision 2 of the Split-AC algorithm proves to be sound.

6.6.2 FPGA implementation Results

Table 6.13 shows the results of the FPGA implementation of the architecture described in chapter 5 for several different parameter combinations. All results are generated from the Xilinx ISE version 7.1.04i and represent post place and route values for revision 2 of the Split-AC algorithm. The following list shows the most important ISE parameters that where changed from their default values:

- **Synthesis effort**: set to "high"
- **Optimization goal**: set to "speed"
- **Place & route effort**: set to "high"
- **Additional post place & route effort**: set to "normal"

We tested four different threshold values (0.01, 0.05, 0.5 and 0.99) when the maximum number of states per fsm is 128 because that number of states requires, on most occasions, the smallest amount of memory. We also implemented the 32 state-0.01 threshold pair, since it provides the global minimum of required memory.

TABLE 6.13
Post Place & Route results for different parameter values for Split-Ac Revision 2.

Max States per FSM	**32**	**128**	**128**	**128**	**128**
Threshold T*f*	**0.01**	**0.01**	**0.05**	**0.5**	**0.99**
Min. Period (nsec)	8.631	8.723	8.763	9.867	9.307

Max. Frequency (MHz)	115.861	114.639	114.116	101.348	107.446
Max. Throughput (Gbps)	0.927	0.917	0.913	0.811	0.860
# Logic Cells	16,980	12,341	23,629	59,983	60,061
# slice Flip-Flops	4,162	3,224	4,215	6,157	6,190
# Slices	8,970	6,602	12,492	31,821	31,860
# Memory Blocks	243	264	141	85	85
Min. Virtex4 Model (Lx/Sx/Fx)	160/55/100	160/55/100	60/35/40	80/-/100	80/-/100

The "Maximum Throughput" value in table 6.13 is calculated from the fact that the architecture we have implemented processes one payload byte (or 8 bits) per cycle.

We can observe that as the character occurrence frequency threshold increases, the maximum attainable frequency declines. Examining the generated timing reports reveals that the critical path of our design is the second pipeline stage in the fsms, specifically the CAM block responsible for the infrequent character transitions. As the frequency threshold increases, more and more characters become infrequent, resulting in larger and wider CAMs that require more time to be traversed and lead to the observed increase in the maximum clock frequency.

We can also see that for the minimal frequency threshold of 0.01, the amount of required logic resources varies significantly between 32 states per fsm and 128 states per fsm. Specifically, for 32 states we require 30.9% more logic cells and 31.1% more slices. This increase is owed to the increase in the number of fsms, which increases the overhead for control logic. We did not calculate the required resources for different values of the threshold with a maximum of 32 states per fsm due to time constraints; nonetheless, we are confident that a similar behavior will be observed.

Another observation is that the threshold increase leads to an enormous increase in the number of used logic cells: a threshold of 0.99 requires nearly triple the logic cells than a threshold of 0.01. This large difference reflects the significant amount of logic required to implement the CAM blocks responsible for the infrequent character transitions. By comparing the resource use for the initial Split-AC algorithm and algorithm revision 2, we would expect to see a significant decrease in required logic for revision 2 of the algorithm. However unexpectedly, this is not always the case as we can see in table 6.14.

Table 6.14

Post Place & Route result comparison between the initial algorithm and revision 2. We have boldfaced the vest results for each case.

Max States per FSM	**32**		**128**		**128**	
Threshold T*f*	**0.01**		**0.01**		**0.5**	
Algorithm	**Rev 0**	**Rev. 2**	**Init.**	**Rev. 2**	**Init.**	**Rev. 2**
Min. Period (nsec)	**8.426**	8.631	**8.242**	8.723	11.923	**9.867**
Max. Frequency (MHz)	**118.680**	115.861	**121.330**	114.639	83.872	**101.348**
Max. Throughput (Gbps)	**0.949**	0.927	**0.971**	0.917	0.671	**0.811**
# Logic Cells	17,554	**16,980**	13,007	**12,341**	**52,152**	59,983

# slice Flip-Flops	4,554	**4,162**	3,632	**3,224**	6,504	**6,157**
# Slices	9,412	**8,970**	6,986	**6,602**	**28,497**	31,821
# Memory Blocks	**243**	**243**	**263**	264	**83**	85

Initially, let's consider the two cases with a frequency threshold of 0.01, for which the required CAM logic is minimal. In both cases we see that revision 2 requires fewer logic resources, as expected. We also observe a slight increase in the minimum clock period for revision 2, which is most likely due to the additional control logic required in the revised algorithm.

While the results for those two cases proved to be as expected, what we did *not* expect were the results for a frequency threshold of 0.5, which corresponds to a CAM-hungry configuration. While revision 2 requires approximately 37.7K (or 22%) fewer CAM input bits than revision 0, the actual number of logic cells for revision 2 is 15% higher and the number of slices is 11.6% higher than that of the initial algorithm.

This paradoxical result is owed to the fact that in revision 0 the CAM of every fsm uses the uncompressed 8-bit character as an "infrequent" character input, which is the same for every fsm, while in revision 2 the CAM uses the character after it has been compressed for that specific fsm. As a result, 256 comparators are enough to satisfy every possible "infrequent" character input of *every* fsm for revision 0, while revision 2 requires one comparator per infrequent character per fsm. A simple graphical example of this fact is shown in figure 6.14.

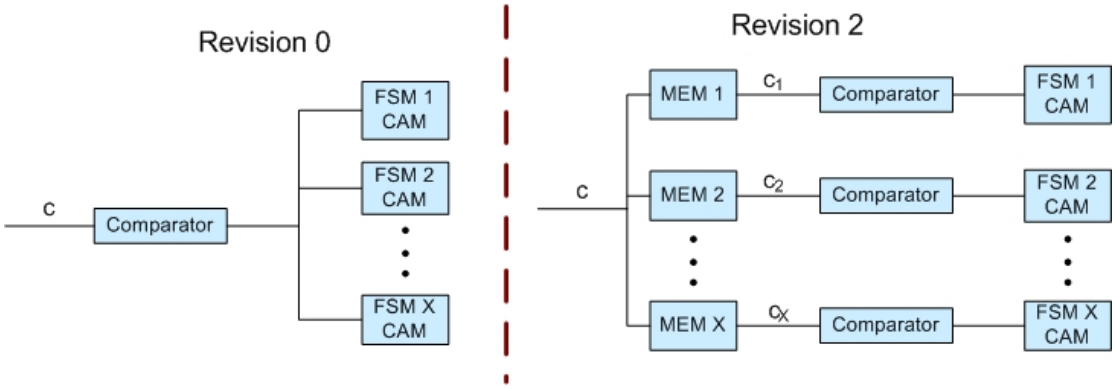

Fig. 6.14: A simple explanation of the paradox between the revisions.

Even though our revisions failed to reduce the amount of required logic, they provided an unexpected benefit; they increase the maximum clock frequency in the CAM-heavy case by 20.8%. This improvement can be attributed to two facts: first, the CAM tags in revision 2 have a smaller width than in revision 0 and, as a result, require smaller and faster comparators. The second reason is that revision 2 needs a different comparator per "infrequent" character per fsm, which can be placed very close to the rest of the CAM logic. Revision 0, on the other hand, needs only 256 for all the fsms, making it impossible to place every comparator next to every CAM that will use its result.

6.6.3 Split-AC Against Bit-Split

The bit-split state machine algorithm described in section 2.4.3 is currently regarded as the smallest, in terms of memory, deterministic Aho-Corasick variation. The authors show that the bit-split algorithm supports the entire Snorts rule set of 12,812 characters with only 0.4 Mbytes (3.2 Mbits) of memory. Table 6.15 shows the results when comparing the implementations of the Bit-Split with three different Split-AC configurations.

Table 6.15
Comparing the Bit-Split algorithm implementation with the Split-AC algorithm implementation

	Bit-Split	**Split-AC #1 32 states 0.01 T*f***	**Split-AC #2 128 states 0.05 T*f***	**Split-AC #3 128 states 0.99 T*f***
Total Memory (Mbits)	3.2	4.27	2.48	1.5
# characters	12,812	24,033	24,033	24,033
Bits/character	261.9	186.4	108.1	65.2

The first configuration represents the best memory-heavy configuration, where almost all characters are stored in lookup memories and only a small amount of CAM logic is used. The second case represents a balanced case, which requires a medium sized memory and a relatively small CAM. Finally, the third case is a CAM-reliant configuration.

We can see that aside from the memory-heavy case, the Split-AC algorithm requires a smaller amount of memory than the Bit-Split algorithm. However, that in itself is not particularly accurate since the Bit-Split rule set is nearly half of the rule set we used. For that reason, we calculate the required number of bits per pattern character by dividing the required memory with the total number of pattern characters. When comparing the bits per character, the Split-AC algorithm is up to 75.1% smaller than the Bit-Split algorithm. Even in the first test case, which relies heavily on memory and has only minimal CAM support (only 0.086K tag bits and 0.1K data bits) the Split-AC algorithm required 28.8% less bits per pattern character.

When it comes to performance, however, the scales weigh heavily against the Split-AC algorithm: Bit-Split has been shown to achieve a processing throughput of up to 36 Gbps while the FPGA implementation of Split-AC almost achieves a meager 1 Gbps. However disappointing Split-AC may be performance-wise, we still managed to achieve our motivating target which was to create the smallest, in terms of required memory, variation of the Aho-Corasick algorithm.

6.7 Result Summary

This chapter contains a large volume of results, which we attempt to summarize at this point. First of all, we showed detailed results of the rule set division into subsets, which can generate from 386 subsets (when the maximum number of states per fsm is 512) up to 864 subsets (for 16 states per fsm).

Afterwards, we showed the ideal Split-AC memory requirements when we have no restrictions in memory address range and when the memory address range is restricted to powers of two. We saw that in the first case, with unrestricted memories, the required memory ranged from 559 Kbits to 5058 Kbits. In the case of memories with an address range restricted to powers of two, the required memory is between 592 Kbits and 6215 Kbits. The difference between restricted and unrestricted memory requirements varied between 6% (for a maximum of 16 states per fsm) and 25.7% (for a maximum of 512 states per fsm). We observed that, in general, memory requirements decrease as the maximum number of states decreases and the frequency threshold increases.

The next thing we showed was a detailed analysis of state lookup memory requirements and translation memory requirements. We also calculated the percentage in which each memory type contributes to the total memory requirements and found that the state lookup memories contribute from 50% to 75% when we have 16 states per fsm maximum, a percentage that increases to between 87% and 94% for a maximum of 512 states per fsm. In general, the contribution of state lookup memories decreases as the threshold Tf increases.

Our next point of focus was the number of "frequent" characters per fsm, which is a direct indication of the required size of the state lookup memories. We found that for a threshold of 0.01 the average number of frequent characters per fsm is roughly between 10.6 and 12.8, values which decrease as the threshold increases to become 1.7 and 2.8 characters per fsm for a 0.99 threshold. The maximum number of "frequent" characters in a single fsm is between 33 and 63 characters for a 0.01 threshold and between 7 and 31 characters for a 0.99 threshold.

We then calculated the CAM requirements for the Split-AC algorithm. We found that the fsms have an average of nearly 0 CAM entries for a 0.01 threshold and roughly 18.5 to 30 CAM entries for a 0.99 threshold. On the other hand, the maximum number of CAM entries in a single fsm is 7 to 52 entries for a 0.01 threshold and increases to between 113 and 510 CAM entries for a 0.99 threshold. When calculating the total CAM tag size required for the initial version of Split-AC, we found that we need 0.21K to 2.66K tag bits for a 0.01 threshold, a value which increases to between 161.5K and 208.9K tag bits for a 0.99 threshold. Correspondingly, the required CAM data size is from 0.1K to 1.4K data bits when the threshold is 0.01 and increases to between 78.5K and 85K for a 0.99 threshold.

Afterwards, we calculated the difference in memory requirements between the initial Split-AC algorithm and the two revisions, which require more memory but reduce CAM tag size. We found that the revised algorithms can require 1 to 2 times the memory of the initial algorithm, with the exact value being almost 1 for a 0.01 threshold and increasing as the threshold increases and the maximum number of states decreases. However, revision 1 reduces CAM tag size by a percentage of 17.5% up to 38%, while revision 2 has an ever larger reduction percentage ranging from 20.3% to 59%. For both revisions, the reduction percentage generally increases when the threshold or the maximum number of states is decreased.

Our next step was to implement the Split-AC algorithm on Xilinx FPGAs. Due to the fact that the FPGA contains only 18 Kbit memory blocks, the actual amount of required memory was greater that the estimated values, between 1530 Kbits and 5868 Kbits (85 to 326 FPGA memory blocks). When comparing these results with the ideal memory requirements in order to calculate the memory utilization percentage, we found that in the case of unrestricted memories the utilization percentage was between 42.5% and 93.5%, while in the case of memories

with address ranges restricted to powers of 2 the utilization percentage was increased and ranged from 44.7% to 106%! Also, when comparing the FPGA memory requirements of the initial Split-AC algorithm with those of revision 2, we found that revision 2 required at most 180 Kbits (or 10 FPGA blocks) more than the initial algorithm.

After performing the "Synthesis" and "Place & Route" procedures for a few specific Split-AC configurations, we found that the maximum achieved throughput is between 0.811 Gbps and 0.971 Gbps, while the required number of logic cells ranges from 12,341 to 60,061. One object of special interest was the result comparison between the implementation of the initial algorithm and revision 2, which showed us that the revised algorithm can require 22% *more* logic cells than the initial Split-AC but can provide a 20.8% performance improvement.

Finally, we compared the Split-AC algorithm with the Bit-Split algorithm, which is currently regarded as the smallest Aho-Corasick variation in terms of memory, and found that Split-AC requires from 28.8% to 75.1% fewer bits per pattern character, a significant improvement.

CHAPTER 7 – CONCLUSION

To summarize, chapter 2 provided a background in pattern matching algorithms. In chapter 3 we have presented the algorithm in detail as well as two revisions which provide a tradeoff between required memory and CAM tag size. Chapter 4 was dedicated to the rules which are used in the IDS as well as the manner in which those rules are divided in groups for maximum split-AC efficiency. Chapter 5 presented an IDS-on-a-chip architecture utilizing the Split-AC algorithm along with fine-grained header classification. Finally, chapter 6 provided detailed results about the general behavior of the algorithm as well as the place & route results for some specific FPGA configurations of the algorithm.

The results of our work have shown that the proposed Split-AC algorithm represents the smallest, in terms of required memory, variation of the classic Aho-Corasick algorithm. The entire Snort rule set can be implemented in an FPGA using as much as 0.547 Kbytes of memory with minimal additional logic for "infrequent characters" or as little as 0.191 Kbytes with significant additional logic for "infrequent characters". When compared with the Bit-Split algorithm, which is currently the smallest Aho-Corasick based algorithm, Split-AC requires 28.8% to 75.1% fewer bits per character.

Although promising in terms of memory use, the Split-AC algorithm for now suffers from performance issues, with the maximum achieved throughput being little under 1 Gigabit per second. This limitation stems from the processing of a single character per cycle, a feature inherited from the original Aho-Corasick algorithm. As such, an interesting topic for further research would be to find if and how the Split-AC can be modified to process more than one character per cycle, as well as how to improve the current design in order to attain higher clock frequencies.

The idea of Character Translation described in section 3.3.1, which forms the heart of the Split-AC algorithm, could also be applied to many other pattern matching algorithms in order to compress the character and reduce its width. Since the impact of Character Translation process is tied to the number of different characters in a pattern group, the rule set division algorithm described in section 4.3.1 can also be used to break large pattern groups into smaller subsets which use common characters. In this way, it would be interesting to see how other algorithms can be improved by using Character Translation. A specific example we believe would be of great interest is using Character Translation with the bit-split fsms described in [9].

We have also presented an Intrusion Detection System architecture utilizing the Split-AC algorithm that can fit on a single FPGA chip, supports fine-grained packet header classification and rule group selection as well as full pattern matching. This architecture can cooperate in a large degree with a software-based IDS system by providing the software with header classification information as well as matching a single pattern per rule. In this way, we have taken several steps towards the goal of combining the versatility of a software IDS platform with the parallel processing capabilities of hardware.

However, this proposed Intrusion Detection System architecture is not bound specifically to the Split-AC algorithm. It could easily be modified to implement some other, potentially faster or more resource efficient algorithm. In this way, another

interesting topic for future research would be finding the fastest and cheapest algorithm with which to implement the pattern matching capabilities of the IDS.

The scientific area of Intrusion Detection Systems presents a fertile field for research, not simply for academic purposes but also due to the growing demand for fast, effective IDS. To that end, we can only hope that our work will prove to be a useful tool for future efforts.

References

[1] R.Boyer and J. Moore. A Fast String Matching Algorithm. *Commun. ACM*, 20(10): 762-772, October 1977.

[2] R. N. Horspool**.** Practical Fast Searching in Strings, *Software - Practice & Experience*, 10(6):501-506, 1980.

[3] C.J Coit, S. Staniford, and J. McAlerney. Towards Faster Pattern Matching for Intrusion Detection, or Exceeding the Speed of Snort. *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, 2002.

[4] A. Aho, and M. Corasick. Fast Pattern Matching. An aid to bibliographic search. *Comm. ACM*, 18(6):333-340, June 1975.

[5] S.Wu, and U. Manber. A fast algorithm for multi-pattern searching. *Technical Report TR-94-17*, University of Arizona, 1994.

[6] M. Norton. Optimizing Pattern Matching for Intrusion Detection

[7] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection.

[8] W. Eatherton. Hardware-Based Internet Protocol Prefix Lookups. Washington University Electrical Engineering Department (MS Thesis), May 1999.

[9] L. Tan, and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA 05),* IEEE CS Press, 2005, pp 112-122

[10] www.snort.org

[11] Mark Norton and Dan Roelker of Sourcefire. Snort 2.0 Rule Optimizer. www.sourcefire.com/technology/whitepapers.htm, June 2004

[12] Mark Norton and Dan Roelker of Sourcefire. Snort 2.0 Multi-Rule Inspection Engine. www.sourcefire.com/technology/whitepapers.htm, June 2004

[13] Mark Norton and Dan Roelker of Sourcefire. Snort 2.0 Detection Revisited. www.sourcefire.com/technology/whitepapers.htm, June 2004

[14] Snort Users Manual.

[15] V. Dimopoulos. Evaluating Hardware Assisted Network Intrusion Detection Software. Graduate thesis.

[16] V. Dimopoulos, G. Papadopoulos, and D. Pnevmatikatos. On the Importance of Hardware in HW/SW Network Intrusion Detection Systems. *Panhellenic Conference on Informatics* (*PCI*), 2005.