A High Throughput String Matching Architecture for Intrusion Detection and Prevention

Lin Tan Timothy Sherwood
Computer Science Department
University of California, Santa Barbara
{tan,sherwood}@cs.ucsb.edu

Abstract

Network Intrusion Detection and Prevention Systems have emerged as one of the most effective ways of providing security to those connected to the network, and at the heart of almost every modern intrusion detection system is a string matching algorithm. String matching is one of the most critical elements because it allows for the system to make decisions based not just on the headers, but the actual content flowing through the network. Unfortunately, checking every byte of every packet to see if it matches one of a set of ten thousand strings becomes a computationally intensive task as network speeds grow into the tens, and eventually hundreds, of gigabits/second.

To keep up with these speeds a specialized device is required, one that can maintain tight bounds on worst case performance, that can be updated with new rules without interrupting operation, and one that is efficient enough that it could be included on chip with existing network chips or even into wireless devices. We have developed an approach that relies on a special purpose architecture that executes novel string matching algorithms specially optimized for implementation in our design. We show how the problem can be solved by converting the large database of strings into many tiny state machines, each of which searches for a portion of the rules and a portion of the bits of each rule. Through the careful co-design and optimization of our our architecture with a new string matching algorithm we show that it is possible to build a system that is 10 times more efficient than the currently best known approaches.

1 Introduction

Computer systems now operate in an environment of near ubiquitous connectivity, whether tethered to a Ethernet cable or connected via wireless technology. While the availability of always on communication has created countless new opportunities for web based businesses, information sharing, and coordination, it has also created new opportunities for those that seek to illegally disrupt, subvert, or attack these activities. With each passing day there is more critical data accessible over the network, and any publicly accessible system on the Internet is subjected to more than one break in attempt per day. Because we are all increasingly at risk there is widespread interest in combating these attacks at every level, from end hosts and network taps to edge and core routers.

Given the importance of protecting information and services, there is a great deal of work from the security community aimed at detecting and thwarting attacks in the network [19, 28, 6]. Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) have emerged as some of the most promising ways of providing protection on the network, and the market for such systems is expected to grow to $918.9 million by the end of 2007 [18]. Network based intrusion detection systems can be categorized as either misuse based or anomaly based. Both systems require sensors that perform real time monitoring of network packets, either by comparing network traffic against a signature database or by finding out-of-the-ordinary behavior, and triggering intrusion alarms. A higher level interface provides the management software used to configure, log, and display alarms generated by the lower level processing. These two parts, working in concert, alert administrators of suspicious activities, keep logs to aid in forensics, and assist in the detection of new worms and denial of service attacks. But it is at the lowest level, where data is actually inspected, that the computational challenge lays.

To define suspicious activities, most modern network intrusion detection/prevention systems rely on a set of rules which are applied to matching packets. A *rule* consists at minimum of a type of packet to search, a *string* of content to match, a location where that string is to be searched for, and an associated action to take if all the conditions of the rule are met. An example rule might match packets that look like a known buffer overflow exploit in a web server; the corresponding action might be to log the packet information and alert the administrator. Rules can come in many forms, but frequently the heart of the rule consists of strings to be matched *anywhere* in the payload of a packet. The problem is that for the detection to be accurate, we need to be able search every byte of every packet for a potential match from a large sets of strings. Searching every packet for all of these strings requires significant processing resources and memory. The problem is that for the detection to be accurate, we need to be able to search every byte of every packet for a potential match from a large set of strings [1]. Searching every packet for all of these strings requires significant processing resources, both in terms of the amount of time to process a packet, and the amount of memory needed. In addition to raw processing speed, a string matching engine must have bounded performance in the worst case [2]. Due to the fact that rule sets are constantly growing and changing as new threats emerge, a successful design must have the ability to be updated quickly and automatically all the while maintaining continuous operation.

In order to address these concerns, we take an approach that relies on a simple yet powerful special purpose architecture working in conjunction with novel string matching algorithms specially optimized for that architecture. The key to achieving both high performance and high efficiency is to build many tiny state machines, each of which searches for a portion of the rules and *a portion of the bits of each rule.* Our new algorithms are specifically tailored towards implementation in an architecture built up as an array of small memory tiles, and we have developed both the software and the architecture in concert with one another. The result of our efforts is a device that maintains tight worst case bounds on performance, can be updated with new rules without interrupting operation, has configurations generated in seconds instead of hours, and is ten times more efficient that the existing best known solutions.

Specifically, this paper makes the following research contributions:

- We describe a novel configurable String Matching Architecture that can store the entire Snort rule set in only 0.4 MB and can operate at upwards of 10 Gbit/sec *per instance.*
- We present a novel String Matching Algorithm that operates through the conjunction of many small state machines working in unison that reduces the number of required out-edges from 256 to as low as 2.
- Our machine is configured by a *Rule Compiler* that partitions and *bit-splits* a finite state machine (FSM) representation of the strings into a set of small implementable state transition tables. The compiler takes only on the order of seconds to complete.
- We compare our design to the state of the art in string matching algorithms and hardware based designs. The key metric is the efficiency (performance/area) and we beat the best existing techniques by a factor of 10 or more.
- We propose a replacement update model that allows non-interrupting rule update which can complete in the order of seconds while FPGA based methods generally require days or months to recompile rules.

[1]The rule set from Snort has on the order of 1000 strings with an average length of around 12 bytes.

[2]so that a performance based attack cannot be mounted against it [10]

The rest of the paper is laid out as follows. In Section 2 we begin with a description of the string matching architecture which implements the many state machines and the way in which the algorithm runs. The actual method of generating the state machines from a given rule set, the tradeoffs and heuristics used to do so, and the details of our Rule Compiler implementation are all described in Section 3. Section 4 presents an analysis of design in terms of performance and efficiency and compares our work to past efforts in the area. In Section 5 a discussion of the related work is presented, and finally we conclude with Section 6.

2 Architecture

Intrusion Detection/Prevention Systems (IDS or IPS) play an increasingly important role in network protection and at the core of most Network IDSs is a computationally challenging problem because network intrusion systems, require a deep packet inspection. Every byte of every packet must be examined which means gigabytes of data must be searched each and every second of operation. In this section we begin by briefly describing the requirements that have driven our design, the main ideas behind our string matching technique, and the details of our architecture.

2.1 IDS/IPS Requirements

In designing our system we have identified the following requirements for Intrusion Detection/Prevention Systems (IDS/IPS).

Worst Case Performance: In order to check incoming packets in real time, without degrading the total throughput, Intrusion Detection/Prevention Systems need string matching algorithms that can keep up with this speed. More specifically, a robust Intrusion Detection Systems should require that its string matching algorithm have stringent worst case performance, otherwise the worst case may be exploited by an adversary to either slow down the network or to force the systems to not inspect some packets, which may include an attack. Neither of these two choices is desirable.

Non-Interrupting Rule Update: Currently the Snort rule set is updated roughly monthly but researchers are currently working on systems that will provide a real-time response to new attacks and worms. In addition to performance requirements, we also want an architecture that can be updated quickly and that can provide continuous service even during an update.

High Throughput per Area: The advantages of small area are twofold. A design that is small enough to be fit completely on chip consumes less resources and can operate much faster than one that relies on off chip memory. Furthermore, many designs use replication to boost performance, and in these cases efficiency becomes performance because of the sheer number of copies you can fit onto a single die.

2.2 String Matching Engine

At a high level, our algorithm works by breaking the set of strings down into a set of small state machines. Each state machine is in charge of *recognizing* a subset of the strings from the rule set. The details of the algorithm are presented in Section 3, but we begin with a description of our architecture.

Our architecture is built hierarchically around the way that the sets of strings are broken down. At the highest level is the full device. Each *device* holds the entire set of strings that are to be searched, and each cycle the device reads in a character from an incoming packet, and computes the set of matches. Matches can be reported either after every byte, or can be accumulated and reported on a per-packet basis. Devices can be replicated, with one packet sent to each device in a load balanced manner, to multiply the throughput, but for our purposes in this paper we concentrate on a single device.

Inside each device is a set of *rule modules*. The left side of Figure 1 shows how the rule modules interact with one another. Each rule module acts as a large state machine, which reads in bytes and outputs string match results. The rule modules are all structurally equivalent, being configured only through the loading of their tables, and each module holds a subset of the rule database. As packet flows through the system, each byte of the packet is broadcast to *all* of the rule modules, and each module checks the stream for an occurrence of a rule in its rule set. Because throughput, not latency, is the primary concern of our design the broadcast has limited overhead because it can be deeply pipelined if necessary.

The full set of rules is partitioned between the rule modules. The way this partitioning is done has an impact on the total number of states required in the machine, and will hence have an impact on the total amount of space required for an efficient implementation. Finding an efficient partitioning is discussed in Section 3. When a match is found in one or more of the rule modules, that match is reported to the interface of the device so that the intrusion detection system can take the appropriate actions. It is what happens inside each rule module that gives our approach both high efficiency and throughput.

If we look into what goes into each rule module, we find that each is made up of a set of tiles. The right hand side of Figure 1 shows the structure of each and every tile in our design. Tiles, when working together, are responsible for the actual implementation of a state machine that really recognizes a string in the input. If we just generated a state machine in a naive manner, each state may transition to one of potentially 256 possible next states at any time. If we were to actually keep a pointer for each of these 256 possibilities, each node would be on the order a kilobyte. A string of length l requires l states [3], and then if you multiply that by the total number of rules you quickly find yourself with far more data than is feasible to store on-chip. So the trade-off is either store the state off-chip and loose your bounds on worst case performance, or find a way to *compress* the data is some way. Past techniques have relied on run length encoding and/or bit-mapping which have been adapted from similar techniques used to speed IP-lookup [27]. Our approach is different in that we split the state machines apart into a set of new state machines each of which matches only some of the bits of the input stream. In essence each new state machine acts as a filter, which is only passed when a given input stream *could be* a match. Only when *all* off the filters agree is a match declared. While we briefly describe the way the algorithm runs for the purpose of describing our architecture here, a full description can be found in Section 3.

Each tile is essentially a table with some number of entries (256 entries are shown in Figure 1), and each row in the table is a state. Each state has two parts. It has some number of next state pointers, which encode the state transitions (4 possible next states are shown and each is indexed by a different 2 bits from the byte stream), and it has a partial match vector. The partial match vector is a bit-vector that indicates the potential for a match for every rule that the module is responsible for. If there are up to r rules mapped to a rule module, then each state of each tile will have a partial match vector of length r bits (Figure 1 shows a module with $r = 16$). By taking the AND of each of the partial match vectors we can find a full match vector, which indicates that

[3] Some states can be shared by different strings, the total number of states is however on the same order of magnitude.

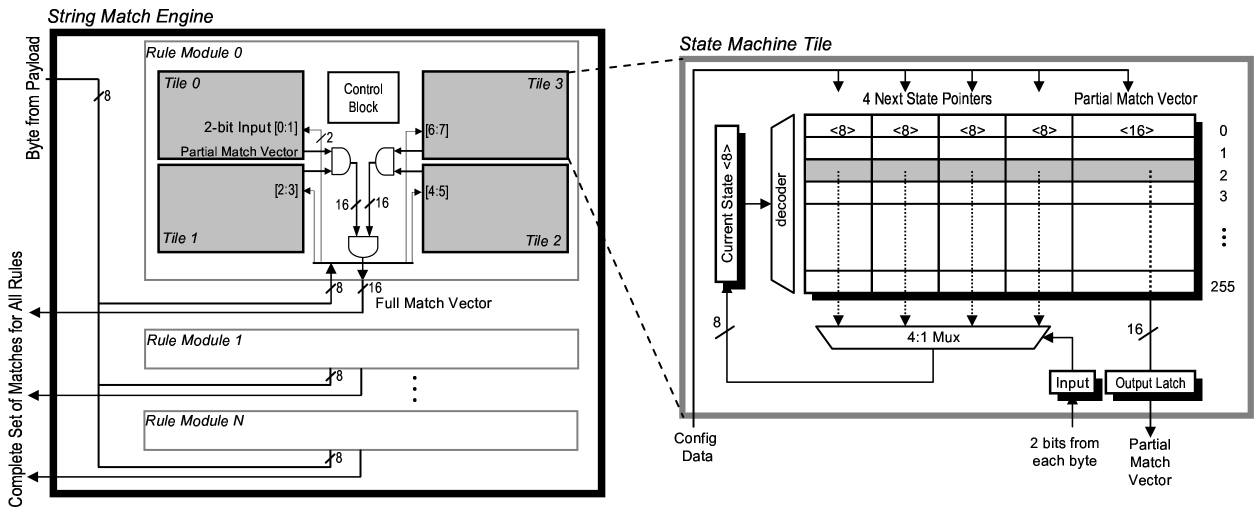


Figure 1: The String Matching Engine of the High Throughput Architecture. The left side is a full Device, comprised of a set of Rule Modules. Each rule module acts as a large state machine and is responsible for a group of rules, r rules. Each rule module is made of a set of tiles (4 tiles are shown in this figure). The right side shows the structure of a tile. Each tile is essentially a table with some number of entries (256 entries are shown in this figure) and each row in the table is a state. Each state has some number of next state pointers (4 possible next states are shown) and a partial match vector of length r. A rule module takes one character (8 bits) as input at each cycle and output the logical AND operation result of the partial match vectors of each tile.

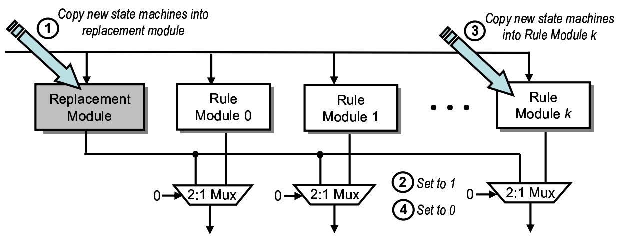

1 Copy new state machines into replacement module
3 Copy new state machines into Rule Module k
Replacement Module
Rule Module 0
Rule Module 1
Rule Module k
2:1 Mux
2 Set to 1
4 Set to 0

Figure 2: Non-interrupting Update

all of the partial match vectors are in agreement and that a true match for a particular rule has been found.

Before accepting any input characters and at the beginning of each packet, all tiles are reset to start from state 0. On each cycle, the input byte is divided into groups of bits (in the example the 8-bits are divided into 4 groups of 2). Each tile then gets its own group of bits. Each tile uses its own internal state to index a line in the memory tile, and the partial match vector is read out along with the set of possible state transitions. The input bits are used to select the next state for updating, and the partial match vector is sent to an AND unit where it is combined with the others. Finally all full match vectors for all modules are concatenated to indicate which of the strings was matched.

2.3 Architectural support for Non-Interrupting Update

A major weaknesses of past techniques which relied on FPGA reconfiguration to encode the strings to be matched is that when the rule database is to be updated, the device needs to go offline. The Snort database, and other proprietary signature databases, have been changing at a rate of more than one rule every days [27]. It is simply unacceptable to the end user to have their network traffic either uninspected or undelivered for minutes or even hours while the rule database is recompiled and transfered to the device. This problem is only going to grow in importance in the coming years as more attacks are unleashed and as automated systems are put in place that can detect new worms and denial of service attacks and generate useful signatures at real time. Our architecture can easily support this functionality through the addition of a temporary tile used for updates.

Figure 2 shows the addition of an new rule module which acts as a temporary state machine. The rule set is already partitioned into several smaller parts that each fit onto a rule module. To replace the contents of one rule module, the rules are first updated and copied to the temporary rule module. At the start of the next packet, the control bit for the module about to be overwritten is set to override with the results from the replacement rule module. The main rule module can then be written (with the same contents as the replacement module) with no stop in service. When the main rule module is completely updated, the control bit is switched back and the replacement module is wiped clean and overwritten with the state machine for the next module in line. Writing to an entire rule module will take on the order of 1.6 microseconds, and to finish an entire update would take less than 108 microseconds. While for our architecture the procedure for updating rules is very straightforward, this is by design and most other techniques we have examined require at least some amount of downtime to performance an update.

3 Algorithm Mapping

In Section 2 we presented the architectural issues in implementing a high speed string matching engine, and in this section we describe the software system, also referred to as the rule compiler, which makes it work.

Readers may already be familiar with efficient algorithms for string matching such as Boyer-Moore [7], which are designed to find a *single* string in a long input. Our problem is slightly different, as we are searching for one of a *set* of strings from the input stream. While simply performing multiple passes of a standard one-string matching algorithm will be functionally correct, it does not scale to handle the tens of thousands of strings that are required by modern intrusion detection systems. Instead, the set of strings that we are looking for can be folded together into a single large state-machine. This method, the Aho-Corasick algorithm [2], is what is used in the `fgrep` utility as well as in some of the latest versions of the `Snort` [19] network intrusion detection system.

3.1 The Aho-Corasick Algorithm

The essence of the Aho-Corasick algorithm involves a pre-processing step which builds up a state machine that encodes all of the strings to be searched. The state machine is generated in two stages. The first stage builds up a tree of all the strings that need to be identified in the input stream. The root of the tree represents the state where no strings have been even partially matched. The tree has a branching factor equal to the number of symbols in the language. For the Snort rules, this is a factor of 256 because snort can specify any valid byte as part of a string [4]. All the strings are enumerated from this root node, and any strings that shares a common prefix will share a set of parents in the tree. The left hand side of Figure 3 shows an example Aho-Corasick state machine constructed for keywords "he", "she" "his", and "hers". To match a string, you start at the root node and traverse edges according to the input characters observed. The second half of the preprocessing is inserting failure edges. When a string match is not found it is possible for the suffix of one string to match the prefix of another. To handle this case failure edges are inserted which shortcut from a partial match of one string to a partial match of another. In Figure 3 we show the full state machine with failure edges (however failure edges that point back to the root node are not shown for clarity).

Let us suppose that the input stream is "hxhe", which would match the string "he". Traversal starts at state 0, and then proceeds to state 1 (after reading "h"), 0 (after reading "x"), back to 1 (after reading "h"), and finally ending at state 2. State 2 is an accepting state and matches the string "he". In the Aho-Corasick algorithm there is a one-to-one correspondence between accepting states and strings, where each accepting state indicates the match to an unique string.

3.2 Implementation Issues

The Aho-Corasick algorithm has many positive properties, and perhaps the most important is that after the strings have been preprocessed the algorithm always runs in time linear to the length of the input stream, regardless of the number of strings. It is impossible for a crafty adversary to construct an input stream that will cause an IDS to lag behind the network resulting in either reduced traffic speed or uninspected data. The problems with the algorithm lie in realizing a practical implementation, and the problems are two-fold. Both problems stem from the large number of possible out edges that are directed out of each and every node. Implementing those out edges requires a great deal of next pointers, 256 for each and every node to be exact. In our simple example, we only have 4 possible characters so it is easier, but in reality encoding these potential state transitions requires a good deal of space. If we were just to encode the state transitions as 32-bit pointers the size of the rule database would balloon to 12.5 megabytes, far larger than what could economically fit on a chip. This brings us to the second problem which is the serial nature of the state machine. The determination of which state we are to go to is strictly dependent on that state that we are currently in. The determination of the next state from the current state forms a critical loop, and because that next state could be one of 256 different memory locations throughout a large data structure it is very difficult to make this fast. While in [27] Tuck et al. show how these structures could be compressed, they still take on the order of megabytes and the compression greatly complicates the computation that needs to be performed.

To examine the behavior of string matching on real data, we generated the Aho-Corasick state machine for a set of strings used for actual intrusion detection and packet filtering. For this we used the default string set supplied with `Snort`, which includes, as part of its rule base, a set of over 1000 suspicious

[4]this feature can be used to identify a particular 4 byte IP address for example

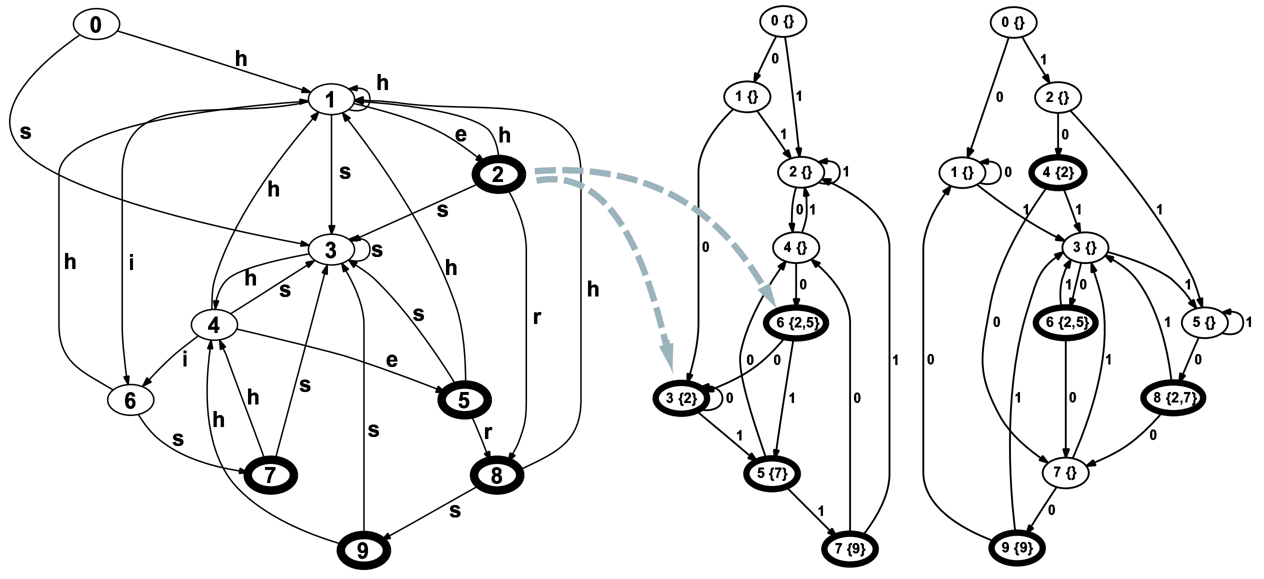


Figure 3: Extracting bit-level parallelism from the Aho-Corasick algorithm by splitting the state machine into 8 parallel state machines. The leftmost state machine is the Aho-Corasick state machine (D) constructed for strings "he", "she", "his" and "hers". Next state pointers pointing back to State 0 is not shown in the graph because it is unrealistic and also unclear to show all of the 256 next state pointers for each state. The other two state machines are two binary state machines B_3 and B_4 among the eight state machines, B_0, B_1 ... , and B_7, split from D. State machine B_3 is only responsible for Bit 3 of any input character, while state machine B_4 is only responsible for Bit 4 of any input character.

strings resulting in an Aho-Corasick state machine with around 10,000 nodes.

3.3 Splitting Apart the State Machines

While Aho-Corasick state machines can be searched in constant time per character, a real implementation requires large amounts of storage and requires a memory reference for each character searched. Storing each state as an array of 256 next pointer is wasteful. Furthermore there is a high variation in the number of next pointers that any given state needs. Nodes near the root of the tree need more than 200 next pointers, while nodes near the leafs need only 1 or 2. We need a way of breaking this problem into a set of smaller problems each of which has more regular behavior.

To solve this problem, we *split the state machines apart* into a new set of 8 state machines. (8 is not optimal which we will show in Section 4.) Each state machine is then responsible for *only one of the eight bits of an input character.*

Three advantages of this technique are:

- The split machines have exactly two possible next states (not a large and variable number as in the original design). This is far easier to compact into a small amount of memory.
- The 8 state machines are loosely coupled, and they can be run independently of one another (assuming we can merge the results back together).
- Each state machine is essentially a binary tree with back edges. This means we can speed the tree up by traversing multiple edges at a time (as in a multi-bit trie [23]).

From the state machine D constructed in Aho-Corasick Algorithm, each bit of the 8-bit ASCII code is extracted to construct its own *Binary State Machine*, a state machine whose alphabet contains only 0 and 1. Let B_0, B_1, ..., B_7 be these state machines (1 per bit). For each bit position i we take the following steps to build the binary state machine B_i. Starting from the start state of D, we look at all of the possible next states. We partition the next states of D into two sets, those that come from a transition with bit i set to 1, and those which transition with bit i set to 0. These sets become two new states in B_i. This process is repeated until we fill out all of the next states in the binary state machine in a process analogous to subset construction (although our binary state machines can never have more states that D). Each state in B_i maps to one or more states in D.

After the construction, the mapping to non-output states of D are not needed any more and so can be eliminated from the resulting state machines. On the other hand, the mapping to output states of D still needs to be stored for all states. Because each output state in D corresponds to a string in the rule set, these lists of output states for a state a in binary state machine indicate strings matched when these states are visited. A resulting state in B_i is an accepting state if it maps back to any of the accepting states of D. A small bit-vector is kept for each state in binary state machines, indicating which of the strings *might* be matched at that point. Only if *all* of the bit-vectors agree on the match of at least one string has a match actually occurred.

Figure 3 shows two of the binary state machines generated from the state machine on the left. The state machine in the middle is state machine B_3 which is only responsible for bit 3 of the input and the state machine on the right is state machine B_4. As you can see, state 2 in the original state machine maps to state 3 and 6 in B_3 and state 4, 6, and 8 in B_4.

Now let us see how a binary state machine is constructed from an Aho-Corasick state machine by constructing B_3 in this concrete example. Starting from State 0 in D, which we call D-State 0, we construct a State 0 for B_3, which is called B_3-State 0, with a state set {0}. Numbers in a state set are D-State numbers. We examine all states kept in the state set of B_3-State 0, which is D-State 0 in this example, and see what D-States can be reached from them reading in input value "0" and "1" in bit 3 respectively. For example, D-State 0 and D-State 1 are reachable from D-State 0 reading in input value "0". A new state, B_3-State 1, with state set {0,1,} is then created. Similarly, B_3-State 2 with state set {0,3} is created as the next state for B_3-State 0 for input value "1". Then B_3-State 3 with state set {0,1,2} is created as the next state for B_3-State 1 for input value "0". The next state for B_3-State 1 for input value "1" is an existing state B_3-State 2, then there is no need to create a new state. B_3 is constructed by following this process until next states of all states are constructed. After the construction, non-output states kept in state sets, such as 0, 1 and 3, are eliminated, resulting in B_3 shown in the middle of Figure 3.

3.4 Finding a Match

Let us examine the search processes in both the original Aho-Corasick state machine and in the corresponding binary state machines for the example input stream "hxhe" used before. Reading in "hxhe", D will be traversed in the order of State 0, State 1, State 0, State 1 and State 2. The last state traversed, namely State 2, indicates the match of string "he". Because each state machine takes only one bit at a time, we will need the binary encoding of this input shown in Table 1. Binary state machine B_3 will see *only* the 3rd bit of the input sequence, which will be 0100. Looking to binary state machine B_3, the state traversal for this input will be State 0, State 1, State 2, State 4 and State 6. State 6 maps to states $\{2, 5\}$ in D. Similarly, the binary state machine B_4 will see the input 1110, and will be traversed in the order of

Char	0	1	2	3	4	5	6	7
h	0	1	1	0	1	0	0	0
x	0	1	1	1	1	0	0	0
h	0	1	1	0	1	0	0	0
e	0	1	1	0	0	1	0	1

Table 1: Binary Encoding of input stream "hxhe"

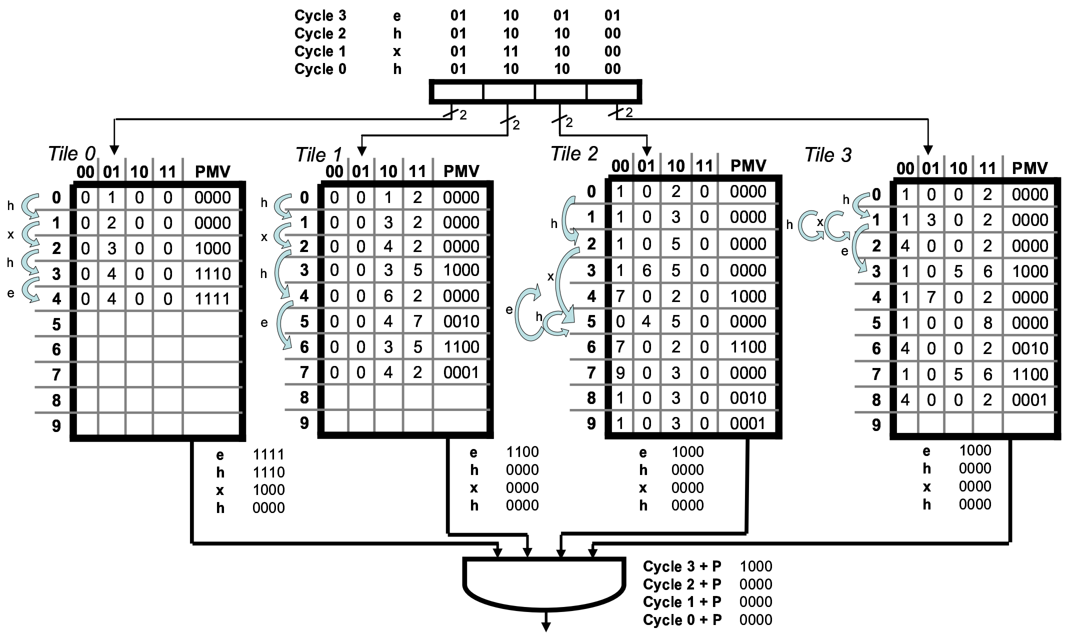
State 0, State 2, State 5, State 5 and State 8, whose state set is $\{2, 7\}$. The actual output state is the intersection of state sets of all 8 binary state machines. In this example, the intersection is State 2, which is the same as the result of Aho-Corasick. In the architecture described in Section 2 this intersection step is completed by taking the logical AND of bit vectors in the on chip interconnect.

The intersection of state sets can be empty, which means there is no actual output but there is partial output for some binary state machines. Let us take input "xehs" for example. The ASCII encoding of bit 3 and bit 4 of "xehs" is 1001 and 1010 respectively. For state machine B_3, the state machine in the middle of Figure 3, the traversal of states is State 0, State 2, State 4, State 6 and State 5, whose state set is $\{7\}$. For state machine B_4, the rightmost state machine in Figure 3, the resulting state set is $\{2, 5\}$ of State 6. The intersection of these two sets are empty, hence no string is matched.

3.5 Partitioning the Rules

If we put all of the more than 1,000 strings into a big state machine and construct the corresponding bit-split state machines, a partial match vector of more than 1,000 bits, most of which are zeros, will be needed for each entry in tiles described in Section 2. This is a big waste of storage. Our solution to this problem is to divide the strings into small groups so that each group contains only a few strings, e.g. 16 strings, so that each partial match vector is only 16 bits. In this way each tile will be much smaller and thus faster to be accessed.

Many different grouping techniques can be used for this purpose and can result in various storage in bits. In order to find the best dividing methods, we want to consider the following constraints. The number of bits of partial match vector determine the maximum number of strings each tile can handle. In addition, each tile can only store a fixed number of states, i.e. 256 states. We want to make full use of the storage of both partial match vectors and state entries, which means we want to pack as many strings in without going over 16 strings or 256 states. Otherwise, we will have to divide this group into two to let the number of states fit, resulting in wasted partial match vectors. By analyzing the distribution of strings in Snort rule set, we find that generally 16 strings require approximately 256 states. And a more detailed discussion



Cycle 3	e	01	10	01	01
Cycle 2	h	01	10	10	00
Cycle 1	x	01	11	10	00
Cycle 0	h	01	10	10	00

Tile 0

	00	01	10	11	PMV
0	0	1	0	0	0000
1	0	2	0	0	0000
2	0	3	0	0	1000
3	0	4	0	0	1110
4	0	4	0	0	1111
5					
6					
7					
8					
9					

Tile 1

	00	01	10	11	PMV
0	0	0	1	2	0000
1	0	0	3	2	0000
2	0	0	4	2	0000
3	0	0	3	5	1000
4	0	0	6	2	0000
5	0	0	4	7	0010
6	0	0	3	5	1100
7	0	0	4	2	0001
8					
9					

Tile 2

	00	01	10	11	PMV
0	1	0	2	0	0000
1	1	0	3	0	0000
2	1	0	5	0	0000
3	1	6	5	0	0000
4	7	0	2	0	1000
5	0	4	5	0	0000
6	7	0	2	0	1100
7	9	0	3	0	0000
8	1	0	3	0	0010
9	1	0	3	0	0001

Tile 3

	00	01	10	11	PMV
0	1	0	0	2	0000
1	1	3	0	2	0000
2	4	0	0	2	0000
3	1	0	5	6	1000
4	1	7	0	2	0000
5	1	0	0	8	0000
6	4	0	0	2	0010
7	1	0	5	6	1100
8	4	0	0	2	0001
9					

	Tile 0	Tile 1	Tile 2	Tile 3
e	1111	1100	1000	1000
h	1110	0000	0000	0000
x	1000	0000	0000	0000
h	0000	0000	0000	0000

Cycle 3 + P	1000
Cycle 2 + P	0000
Cycle 1 + P	0000
Cycle 0 + P	0000

Figure 4: The state transitions of input stream "hxhe" on the rule module for strings "he", "she", "his" and "hers". Only the first 4 bits of 16-bit Partial Match Vectors (PMV) are shown in this diagram because the rest of 12 bits are all zeros for only 4 strings are mapped onto the Rule Module. Here instead of splitting into 8 state machines, we split the Aho-Corasick state machine into 4 state machines, each of which is responsible for 2 bits of an input byte. The Full Match Vector output on Cycle 3+P, 1000, shows that by this cycle string "he" is matched.

about selecting the size of each tile can be found in Section 4.

A good solution is therefore to sort all strings lexicographically and then divide them sequentially into groups so that all the common prefixes can share states in state machines and thus use less states in total. While this is not the optimal solution, it beats the two alternatives, dividing by length and dividing randomly. The dividing by length method would consume 21.9% more states and 13.6% more groups than the method we use, and the random grouping technique would use 12.1% more states and 4.5% more groups.

3.6 Filling the Tables

Until now, we have shown how to break a rule set into a set of groups, the way to construct Aho-Corasick state machines for each group, and the algorithm to split these Aho-Corasick state machines into new sets of state machines. The final step to mapping a rule set onto our architecture is then filling the tables in all modules. As described in Section 2.2 , each entry in a table is for one state. The next state pointers and the partial match vector for state x is stored in entry x. Figure 4 shows an example of 4 state machines split from the Aho-Corasick state machine in Figure 3 mapped onto our architecture. Here instead of splitting into 8 state machines, we split the Aho-Corasick state machine into 4 state machines, which is optimal in terms of storage which we will show in Section 4. Each of these 4 state machines is responsible for 2 bits of an input byte. Still taking "hxhe" as an example input stream, the transitions of all of the 4 state machines starting from state 0 are shown by arrows. At each cycle, a partial match vector is produced by each tile, and the logic AND of these partial match vectors are outputted. According to different requirements of Intrusion Detection/Prevention Systems, our architecture can output only after an entire packet is scanned instead of at each cycle.

4 Analysis of Design

Now that we have presented our string matching architecture and the algorithm used to construct its configuration from a set of strings, we now present an analysis of several important design options and compare against prior work.

4.1 Theoretical Optimal Module Sizes and Group Sizes

As we mentioned in the prior section, we can divided the Aho-Corasick state machine into 8 binary state machines, each of which processes only 1 bit at a time. While 8 binary state machines is the easiest to understand, we could also split the Aho-Corasick algorithms into 4 state machines, each of which processes 2 bits at a time, or 2 state machines that process 4 bits at a time. A different way to divide up the original state machine is to partition the strings into groups of different sizes, such as 8, 16, 32, 64 and

n	Fanout	Storage in bits $T_{n,g}$
2	16	$n\lceil\frac{S}{g}\rceil * 2^p * (g + g + 32p)$
4	4	$n\lceil\frac{S}{g}\rceil * 2^p * (g + 3g + 16p)$
8	2	$n\lceil\frac{S}{g}\rceil * 2^p * (g + 7g + 16p)$

Table 2: Optimal Module Sizes. $\lceil log_2(gL)\rceil$ is denoted by p for clarity.

128 strings per group. We would like to know which combination of the two parameters, module size n (the number of state machines per rule module) and group size g (the number of strings per group), is the best in terms of total storage in bits among all possible combinations.

Given a rule set of S strings, each of which has L characters per string on average (length), the total number of bits our architecture requires is approximately,

$$T_{n,g} = n\lceil\frac{S}{g}\rceil 2^{\lceil log_2(gL)\rceil}(\lceil log_2(gL)\rceil 2^{\frac{8}{n}} + g)$$

From this formula, we can see that the smaller the group size g is the smaller $T_{n,g}$ is.

The effect of n on $T_{n,g}$ is not that direct from the formula above. We can see this effect more clearly if we plug numerical n into $T_{n,g}$. The concrete results after variable n is plugged in are shown in Table 2. Fanout is the number of next state pointers for each state.

We can see from Table 2 that $T_{4,g}$ is minimum when the constraint $g < 8p$ is satisfied, which is always true in practice when r is not very big, say less or equal to 64.

4.2 Practical Optimal Module Sizes and Group Sizes

We have obtained the theoretical optimal parameters for our architecture from the previous section. We are now going to confirm some of these results and point out some of the problems of them and obtain the optimal parameters in practice.

There are three problems with the theoretical analysis above. First, the approximation of the number of rule modules used in $T_{n,g}$, $\lceil\frac{S}{g}\rceil$, is for the ideal case and in reality more rule modules may be needed. Second, $p = \lceil log_2(gL)\rceil$ is used as the approximation of the number of bits to encode each state. If the longest string is longer than gL, requiring more than gL states, more bits than p will be needed to do the encoding. The length of the longest string in the Snort rule set we use is between 64 and 128, which means at least 7 bits are needed. In short, p values that is not large enough to accommodate the longest string have to be eliminated. Finally, the total storage consists of the total number of bits and some circuit overhead, e.g. decoder and multiplexer. The more groups the strings are divided into, the more overhead the entire system will have.

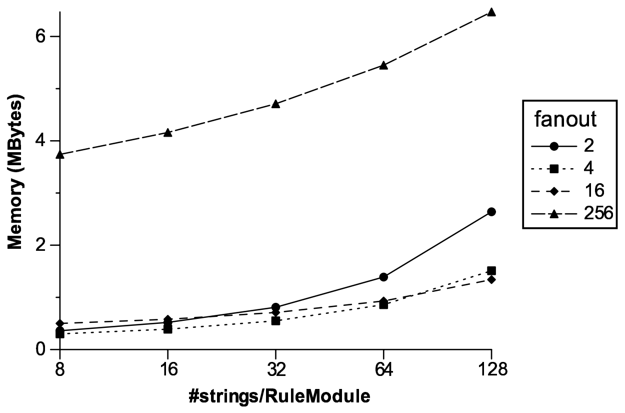


Figure 5: Practical Memory Comparison for Different Fan-out and Group Sizes. X-axis is the number of strings per rule module, also referred to as group sizes, on a logarithmic scale. The four lines correspond to data for four different fanout.

We tune the two parameters on the real Snort rule set and these practical results are shown in Figure 5. The X-axis is the group sizes g on a logarithmic scale, and the y-axis is memory in megabytes. Four lines in the figure correspond to data for four different fanout. We can see from the graph that the line for fanout 256 is prominently high above in the graph, which indicates that the traditional way with state machines with 256 next state pointers use way more storage than our bit-split state machines. Even if the group size is as small as 8, 3.74MB are needed, which is more than 7 times of the storage of the other fanout. The fact that all lines increase monotonically confirms that the smaller the group size the smaller the total memory needed. We can see that the two best points are for fanout 4 and group size 8 and group size 16, which only use 0.4MB to store the entire Snort rule set. We chose group size 16 which allows for longer strings, with the concern that string length is growing and larger size causes less overhead.

4.3 Detailed Throughput and Area Comparison

As we mentioned in Section 2, IDS/IPS have three main requirements on string matching algorithms, which are worst case throughput, non-interrupting update and area efficiency. So here we compare these three requirements on our design and a number of other designs. In Section 2, we have described our architectural support for incremental and non-interrupting update, therefore we concentrate on the other two requirements, the worst case throughput and area efficiency, as well as performance per area (Throughput*Characters/Area) in the rest of this section.

From Table 3, we can see that our design can achieve worst case throughput of over 10 Gbit/sec even if only 1 byte is read in at each cycle time, while the best of all FPGA-based methods we examined can only achieve throughput over 3 Gbit/sec with this read-in rate. Even the smallest throughput of our design is over 8 Gbit/sec with a great increase in area efficiency and performance per area. In addition to throughput, we do actual area efficiency (in char/mm^2) comparison among different designs. We explore tradeoffs in SRAM memory bank sizes using a modified version of CACTI 3.2 [21]. Area results of FPGA-based methods are calculated from the number of LUTs and area needed by each LUT and are normalized to the same technology (0.13 μm). Our design achieves area efficiency of 320.972 characters/mm^2, which is more than 4 times of that of the best FPGA-based designs examined. The performance per area of our design is near 12 times of that of the best examined FPGA-based methods.

Figure 6 shows the efficiency comparison of our bit-split FSM (Finite State Machines) design and FPGA-based designs. The X-axis is the area efficiency, the number of characters per square millimeter can support. The Y-axis is the throughput in Gbit/sec. All points on the same dashed line in the figure have the same performance per area value. Dashed lines on the upper right part of the figure have higher performance per area value. So the points on the upper right part denote more efficient designs. We can clearly see that even the least efficient configuration of our bit-split FSM design beats the best FPGA-based designs examined and most of the bit-split FSM design are far better than these FPGA-based designs.

Our method is also better than the best software method we examined. Tuck et al. [27] optimized the Aho-Corasick algorithm by looking at bitmap compression and path compression to reduce the amount of memory needed to 2.8MB and 1.1MB respectively, which are still at least about 3 times of that of our design, which is only 0.4MB.

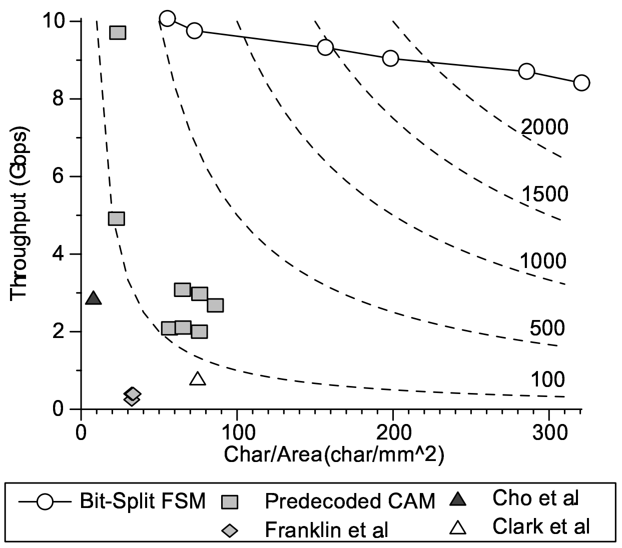


*Figure 6: Efficiency (Throughput*Char/Area) Comparison of String Matching Designs. Each dashed line is aggregation of all points that have the same performance per area value. The points on the upper right part denote more efficient designs. We can clearly see that even the least efficient configuration of our bit-split FSM design beats the best FPGA-based designs examined and most of the bit-split FSM design are far better than these FPGA-based designs.*

5 Related Work

Recently there has been a flurry of work related to string matching in many different areas of computer engineering. This work can be broadly broken down by the target of its intended implementation, either in software, or in an FPGA. While we could not hope to provide a comprehensive set, we attempt to contrast our work with several key representatives from each area.

Software-based: Most software based techniques concentrate on the reducing the common case performance. Boyer-Moore [7] is a prime example of such technique, as it lets its user search for strings in sub-linear time if the suffix of the string to be searched for appears rarely in the input stream. While Boyer-Moore only searches for one string at a time, Fisk and Varghese [12] present a multiple-pattern search algorithm that combines the one-pass approach of Aho-Corasick with the skipping feature of Boyer-Moore as optimized for the average case by Horspool. The work by Tuck, et al. [27] take a different approach to optimizing Aho-Corasick by instead looking at bitmap compression and path compression to reduce the amount of memory needed.

FPGA-based: The area that has seen the most amount of string matching research is in the reconfigurable computing community [9, 15, 22, 13, 4, 5, 8, 11, 3]. Proponents of the work in this area argue intrusion detection is a perfect application of reconfigurable computing because it is computationally intensive, throughput oriented, and the rule sets change overtime but only relatively slowly. Because FPGAs are inherently reconfigurable, the majority of prior work in this area focuses on efficient ways to map the a given rule set down to a specialized circuit that implements the search. The configuration (the circuit implemented on the FPGA) is custom designed to take advantage of the nature of a given specific rule set, and any change to the rule set will require the generation of a new circuit (usually in a hardware description language) which is then compiled down through the use of CAD tools. The work of Sourdis and Pnevmatikatos [22] describes an an approach that is specifically tuned to the hardware resource

Description	Throughput (Gbps)	Char/Area ($1/mm^2$)	Throughput* Char/Area ($Gbps/mm^2$)	Notes
Bit Split FSM	10.074	55.219	556.306	Bank size 64B
	9.759	72.592	708.424	Bank size 128B
	9.326	156.569	1460.092	Bank size 256B
	9.042	198.442	1794.316	Bank size 512B
(Group Size 16)	8.706	285.676	2487.194	Bank size 1024B
	8.408	320.972	2699.210	Bank size 2048B
Sourdis and Pnevmatikatos [22]	9.708	23.482	227.968	4B/cc, Virtex2-6000
	4.913	22.682	111.434	4B/cc,Spartan3-5000
	3.080	64.990	200.170	Virtex2-3000, g=64
Pre-decoded	2.975	76.035	226.203	Virtex2-3000, g=128
CAMs	2.678	86.076	230.510	Virtex2-3000, g=256
	2.086	56.709	118.295	Spartan3-1500, g=64
	2.107	65.350	137.693	Spartan3-1500, g=128
	2.000	75.851	151.703	Spartan3-1500, g=256
Hutchings et al. [15]	0.248	32.496	8.059	1B/cc, Virtex-1000
Regular	0.400	32.496	12.998	1B/cc, Virtex-1000
Expressions	0.396	33.353	13.208	1B/cc, Virtex-2000
Cho et al. [8] Dis. Comparators	2.880	$\sim$ 7.911	$\sim$ 22.785	1B/cc, Altera EP20K
Clark et al. [9] NFAs-Shared Decoders	0.800	$\sim$ 74.733	$\sim$ 59.787	1B/cc, Virtex-1000

Table 3: Detailed Comparison of Our Bit Split FSM Design and FPGA-based Designs. g = group size. $1B/cc$ = read in one byte per cycle time.

available to devices available from Xlinix to provide near optimal resource utilization and performance. Because they demonstrate that there mapping is highly efficient, and they compare against prior work in the domain of reconfigurable computing, we compare directly against their approach. Even though every shift-register and logic unit is being used in a highly efficient manner, the density and regularity of SRAM are used to a significant advantage in our approach resulting in silicon level efficiencies of 10 times or more. It should be also noted that most FPGA based approaches are usually truly tied to an FPGA based implementation because they lie on the underlying reconfigurability to adjust to new rule sets. In our approach this is provided simply by updating the SRAM and can be done in a manner that does not require a temporary loss of service.

While in this paper we have explored an application specific approach, it is certainly feasible that the techniques we have developed and presented would allow for the efficient mapping of string matching to other tile based architecture. For example, instead of using a specialized memory tile, if the tiles are programmable in a more general sense [16, 26, 24] the optimizations we present would still be valuable. We will make our rule compiler freely available for academic use once published.

6 Conclusions

While in this paper we examine the use of our technique strictly for intrusion detection with Snort, our methodology is general purpose enough to be useful across a variety of other application domains. String matching plays a crucial part in the execution of many spam detection algorithms (to match strings which are most likely spam) [1]. In general, any state machine problem where there is a high fanout from each of the nodes can be improved dramatically. Run-time model checking, which work by comparing executed state against a known model of accepted computation is an example of one such application which would benefit greatly. Even outside of security we see opportunities for high-speed string matching. For example, in peephole optimization, we want to replace a sequence of instructions with another functionally equivalent but more efficient sequence to achieve higher overall performance of programs [25, 17]. A sequence of instructions to be replaced can be of different lengths and can appear at any location of programs. A faster string matching algorithm could boost optimization speed and enable the creation of simplified run-time optimizers for embedded systems. There may also be opportunities to apply our technique to well studied areas of IP lookups [20], and packet classification [14].

As security becomes an increasingly important concern, computer systems will almost certainly need to change to help address this problem. While Network Intrusion Detection and Prevention Systems are certainly not a silver bullet to the complex and dynamic security problems faced by today's system designers, they do provide a powerful tool. Because network IDSs require no update or modification to any of the systems they help to protect, they have grown rapidly in recent years both in adoption and power. In this paper we present an architecture and algorithm that is small enough to be included on existing network chips as a separate accelerator, that is fast and efficient enough to keep up with aggressive network speeds, and that supports always on capability with tight worst case bounds on performance. To provide this functionality we rely on the combination of a simple yet scalable special purpose architecture working in tandem with a new specialized rule compiler which can extract bit-level parallelism from the state of the art string matching algorithms. In the end, we have shown how the problem of high-speed string matching can be addressed by converting the large database of strings into many tiny state machines, each of which searches for a portion of the rules and a portion of the bits of each rule.

References

[1] Commtouch ® Software Ltd. White paper: Recurrent pattern detection (RPD™) technology. www.commtouch.com/documents/Commtouch RPD White Paper.pdf.

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

[3] Michael Attig, Sarang Dharmapurikar, and John Lockwood. Implementation results of bloom filters for string matching. In *Proceedings of the Field-Programmable Custom Computing Machines, 12th Annual IEEE Symposium on (FCCM'04)*, pages 322–323. IEEE Computer Society, 2004.

[4] Zachary K. Baker and Viktor K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on fpgas. In *Proceedings of the Field-Programmable Custom Computing Machines, 12th Annual IEEE Symposium on (FCCM'04)*, pages 135–144. IEEE Computer Society, 2004.

[5] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on fpgas. In *Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 223–232. ACM Press, 2004.

[6] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Security Symposium*, June 2000.

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

[8] Young H. Cho, Shiva Navab, and William H. Mangione-Smith. Specialized hardware for deep network packet filtering. In *12th International Converence on Field-Programmable Logic and Applications*, 2002.

[9] Christopher R. Clark and David E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *FPL*, 2003.

[10] Scott A Crosby and Dan S Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of USENIX Annual Technical Conference*, June 2003.

[11] Sarang Dharmapurikar, Michael Attig, and John Lockwood. Deep packet inspection using parallel bloom filters. *Micro, IEEE*, 24(1):52–61, January - February 2004.

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

[13] Maya Gokhale, Dave Dubois, Andy Dubois, Mike Boorman, Steve Poole, and Vic Hogsett. Granidt: Towards gigabit rate network intrusion detection technology. In *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 404–413. Springer-Verlag, 2002.

[14] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network Magazine*, March 2001.

[15] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the 10 th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, page 111. IEEE Computer Society, 2002.

[16] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Annual International Symposium on Computer Architecture*, June 2000.

[17] B. J. McKenzie. Fast peephole optimization techniques. *Softw. Pract. Exper.*, 19(12):1151–1162, 1989.

[18] Infonetics Market Research. Intrusion detection/prevention product revenue up 9% in 1q04. Technical report, June 2004.

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

[20] M. Sanchez, E. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Magazine*, 15(2):8–23, 2001.

[21] P. Shivakumar and N.P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report WRL-2001-2, HP Labs Technical Reports, Dec. 2001.

[22] Ioannis Sourdis and Dionisios Pnevmatikatos. Predecoded cams for efficient and high-speed nids pattern matching. In *Proceedings of the Field-Programmable Custom Computing Machines, 12th Annual IEEE Symposium on (FCCM'04)*, pages 258–267. IEEE Computer Society, 2004.

[23] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 7(1):1–40, February 1999.

[24] Steve Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *36th International Symposium on Microarchitecture*, December 2003.

[25] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. Using peephole optimization on intermediate code. *ACM Trans. Program. Lang. Syst.*, 4(1):21–36, 1982.

[26] Michael Bedford Taylora, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry, Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Annual International Symposium on Computer Architecture*, June 2004.

[27] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *the 23rd Conference of the IEEE Communications Society (Infocomm)*, March 2004.

[28] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. In *Workshop on Evaluating and Architecting Systems for Dependability*, June 2002.