Bit-Split String-Matching Engines for Intrusion Detection and Prevention

LIN TAN
University of Illinois, Urbana-Champaign
BRETT BROTHERTON
University of California, Riverside
and
TIMOTHY SHERWOOD
University of California, Santa Barbara

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 thousands of 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 codesign and optimization of 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.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; I.5.5 [**Pattern Recognition**]: Implementation

General Terms: Algorithms, Design, Performance, Security

Additional Key Words and Phrases: String-matching architecture, security, state machine splitting

Authors' addresses: Lin Tan, Department of Computer Science, University of Illinois, Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801; Brett Brotherton, University of California, Riverside, CA; Timothy Sherwood, Department of Computer Science, University of California, Santa Barbara, CA 93106-5110.

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 [Roesch 1999; Xu et al. 2002; Baratloo et al. 2000]. 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 USD by the end of 2007 [IDSmarket 2004]. Network-based intrusion detection systems can be categorized as either misuse 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. However, it is at the lowest level, where data is actually inspected, that the computational challenge lies.

To define suspicious activities, most modern network intrusion detection/prevention systems rely on a set of rules that are applied to matching packets. At minimum, a *rule* consists 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 to search every byte of every packet for a potential match from a large set of strings. For example, the rule set from Snort has on the order of 1000 strings with an average length of around 12 bytes. 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 so that a performance-based attack cannot be mounted against it [Crosby and Wallach 2003]. Because of

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 especially 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 toward 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 than 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[1] 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.
- We propose an extension of our bit-split algorithm that can process multiple bytes per cycle and the highest throughput we can achieve is 36 Gbit/sec by reading 4 bytes per cycle.
- 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 noninterrupting rule update, which can complete in the order of seconds, while FPGA-based methods generally require days or months to recompile rules.
- We build a prototype of our bit-split string-matching engine using FPGA boards to ensure our string-matching architecture is both functionally correct and fully implementable as a hardware module.

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

[1]The default string set supplied with `Snort` includes a set of over 1000 suspicious strings containing more than 12,000 characters.

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. Section 5 discusses the extension of our bit-split algorithm that can process multiple bytes per cycle. Section 6 describes the prototype of our string-matching engine that we built using FPGA boards. In Section 7, a discussion of the related work is presented, and, finally, we conclude with Section 8.

2. ARCHITECTURE

Intrusion Detection/Prevention Systems (IDS or IPS) play an increasingly important role in network protection. At the core of most Network IDSs is a computationally challenging problem, because it requires *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).

1. **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 System should require that its string-matching algorithm has 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.
2. **Noninterrupting Rule Update:** Currently the Snort rule set is updated roughly monthly, but researchers are working on systems that will provide a real-time response to new attacks and worms [Singh et al. 2004]. 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.
3. **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 that 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 groups and building a small state machine for each group. Each state machine is in charge of *recognizing* a subset of the strings from the rule set.

The first concern is that building a state machine from any general regular expression can, in the worst case, require an exponential number of states. We get around this problem by exploiting the fact that we are not matching general regular expression, but rather a proper and well defined subset of them for which we can apply the Aho-Corasick algorithm [Aho and Corasick 1975]. The other problem is that if we are not careful we will need to support 256 possible out-edges (one for each possible byte) on each and every node on the state machine. This results in a huge data structure that can neither be stored nor traversed efficiently. We solve this problem by bit-splitting the state machines into many smaller state machines, which each match only one bit (or a small number of bits) of the input at a time (in parallel). 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 a 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.

Each rule module 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

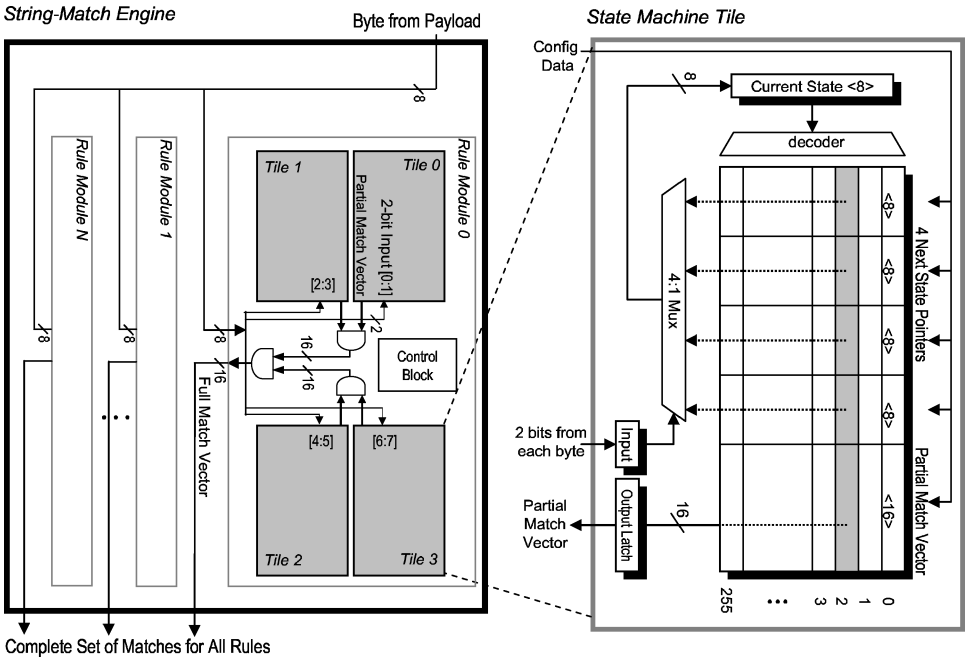

Fig. 1. The string-matching engine of the high-throughput architecture. The left side is a full device, comprised of a set of rule modules. Each rule module acts as a large state machine and is responsible for a group of rules, g rules. Each rule module is composed of a set of tiles (four tiles are shown 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 (four possible next states are shown) and a partial match vector of length g. 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.

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 of 1 Kb. A string of length l requires l states,[2] and then if we multiply that by the total number of rules, we quickly find ourselves with far more data than is feasible to store on-chip. Thus, the trade-off is either to store the state off-chip and lose the bounds on worst-case performance, or to find a way to *compress* the data in 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 [Tuck et al. 2004]. 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* of 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.

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

Each tile is essentially a table with some number of entries (256 entries are shown in Figure 1); 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 (four possible next states are shown and each is indexed by a different two 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 g rules mapped to a rule module, then each state of each tile will have a partial match vector of length g bits (Figure 1 shows a module with $g = 16$). By taking the AND of each of the partial match vectors, we can find a full match vector, which indicates that 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 eight-bits are divided into four groups of two). 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 were matched.

2.3 Support for Noninterrupting Update

A major weakness of many 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 an aggregate rate of more than one rule every day [Tuck et al. 2004].

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 transferred to the device. This problem is only going to grow in importance in the coming years as more attacks are unleashed. Automated systems will be put in place that detect new worms and denial of service attacks to generate useful signatures in 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 μsec, and to finish

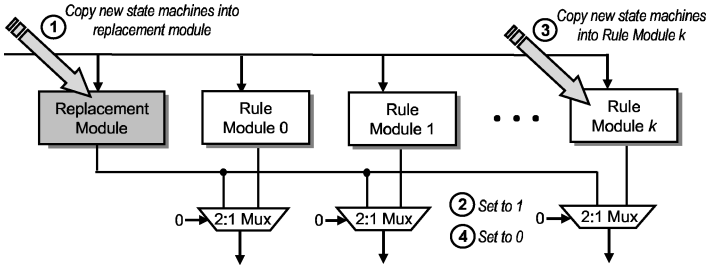

Fig. 2. Noninterrupting update is supported through the addition of an extra rule module, which covers for a module taken off-line so it can be updated.

an entire update would take less than 108 μsec. Compiling a set of rules for a single module takes around 0.4 sec and is fully decoupled from device operation. Therefore, if rules are added one at a time, each rule will take less than a total of 1 sec to add. A full recompile of all the rules takes approximately 20 sec.

When adding rules to modules, there are a several issues that need to be handled. First, care needs to be taken not to overflow a rule module (in terms of either number of rules or number of states allowed). This can be enforced by keeping a list of the resources remaining for each rule module in the management software. As long as the module does not overflow, any new rule can be added to any module with full functional correctness. The only aspect that will be impacted by inserting rules out of lexicographic order is that there will be less prefix sharing and thus more states will be required. Rather than moving many rules just to do an insert, it may make more sense to simply keep a module reserved for adding new rules and then periodically perform a global reshuffle. The exact nature of the rule update policy is left open to IDS system designer.

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 [Boyer and Moore 1977], 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 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 [Aho and Corasick 1975], is what is used in the `fgrep` utility as well as in some of the latest versions of the `Snort` [Roesch 1999] network intrusion detection system. One of the biggest advantages of Aho-Corasick it that it performs well even in the worst-case, which makes it impossible for an adversary to construct a stream of packets that is difficult or

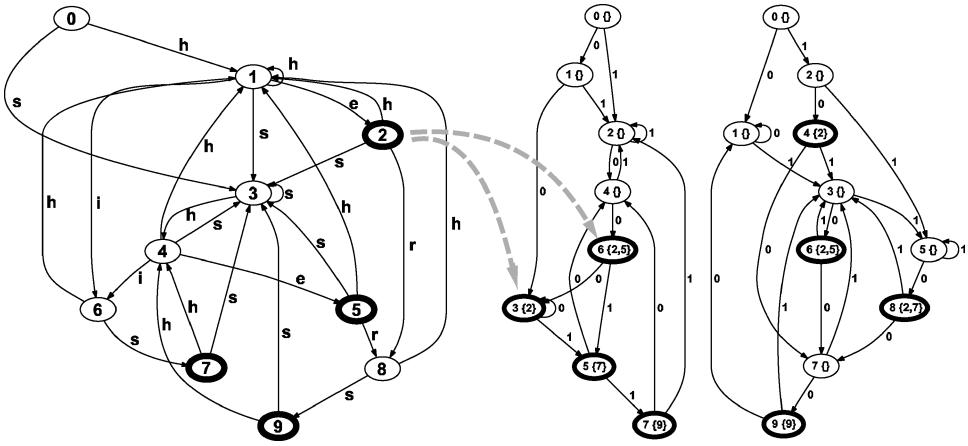

Fig. 3. Extracting bit-level parallelism from the Aho-Corasick algorithm by splitting the state machine into eight parallel state machines. The leftmost state machine is the Aho-Corasick state machine (D) constructed for strings "he," "she," "his," and "hers." Next state pointers pointing back to State 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 in this limited space. 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. The dashed lines shows how one state from the original machine (2) may need to be recognized at multiple bit-split states (3 and 6).

impossible to scan. All of our techniques inherit their worst-case performance behavior from the original Aho-Corasick algorithm.

3.1 The Aho-Corasick Algorithm

The essence of the Aho-Corasick algorithm involves a preprocessing 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.[3] All the strings are enumerated from this root node and any strings that share 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, we 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).

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

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 a 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 twofold. 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 four 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 MB, 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 Tuck et al. [2004] 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 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 dependent memory reference for each character searched. Storing each state as an array of 256 next pointers 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 leaves 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 eight state machines. (Eight 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 eight state machines are loosely coupled, and 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 [Srinivasan and Varghese 1999]). The algorithm, analysis, and results of traversing multiple edges at a time are described in Section 5.

From the state machine D constructed in Aho-Corasick Algorithm, each bit of the eight-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 than 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 no longer needed 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 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. After conversion, 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." B_3-State 3 with state set {0, 1, 2, 6} is then 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. The pseudocode of the construction algorithm is as follows.

Algorithm 1. Construction of a binary state machine from an Aho-Corasick state machine.

Input. An Aho-Corasick state machine *base* and the bit position for which a binary state machine will be constructed *pos*. Bits in a character are numbered 0, 1, 2, 3, 4, 5, 6, 7 from left to right. *base.move*[][] is the transition function and *base.output*[] is the output function [Aho and Corasick 1975].

Output. A binary state machine *b*[*pos*], with *move*[][] as the transition function and $nState$ as the total number of states, that accepts one bit at position *pos*.

Method.

```
{
    state = 0;
    queue = {state};
    state_sets[state] = {0};

    while (queue is not empty) {
        se[0] = empty;  /* se[0]: a set used to record all possible
                           D-states if the input bit is '0' */
        se[1] = empty;  /* se[1]: a set used to record all possible
                           D-states if the input bit is '1' */
        let r be the next state in queue;
        queue = queue - {r};
        isOutputState[0] = false; /* false: 0 state is not an
                                     output state */
        isOutputState[1] = false; /* false: 1 state is not an
                                     output state */

        for (each state st in state_sets[r]) {
            for ( i = 0; i < 256; i++) {
                s = base.move[st][i];
                a = ( ( (unsigned) (i << pos) ) >> (8 - 1) ) & 0x00000001 ;
                                /* assign the desired bit to ''a''
                                   i << pos: shift the desired bit to the
                                       left most
```

```
                                >> (8 - 1) shift it to the right most */
            add s to se[a];
            if (base.output[s] is not empty )
                isOutputState[a] = true;
        }
    }
    for (i = 0; i < 2; i++) {
            if ( se[i] is not empty ) {
                existing = false;
                for (j = 0; j <= state; j++)
                    if ( se[i] == state_sets[j] ) { /* the
                    new state set is generated before */
                        position = j;
                        existing = true;
                        break;
                    }
                if (!existing) { /* a real new state */
                    state++;
                    queue = queue + {state};
                    move[r][i] = state;
                    state_sets[state] = se[i];
                    F[state] = isOutputState[i];
                    /* F[]: record final states */
                }
                else { /* an existing new state */
                    move[r][i] = position;
                }
            }
    }/* end of for */
}

nState = state + 1;   /* nState: total number of states in
this Binary DFA */

/* only store old output states in state_sets[] to save
space, non-output states are eliminated in the following
process */
for (i = 0; i < nState; i++ ) {
    settemp = empty;
    for (each state st in state_sets[i] ) {
        if ( base.output[st] is not empty )
            settemp = settemp + {st};
    }
    state_sets[i] = settemp;
}
} /* end of algorithm */
```

Table I. Binary Encoding of Input Stream "hxhe"

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

The time complexity of this bit-split algorithm is $O(N)$, where N is the total number of states in the original Aho-Corasick state machine. This is because the total number states of each bit-split state machine is rigorously bounded by N. Briefly speaking, bit-splitting creates a set of strings that are all 1's and 0's. In the worst-case, there are no duplicate strings created by the bit splitting. Therefore in the worst-case, a state machine created from those binary strings has the same number of states as the Aho-Corasick state machine. Otherwise, a bit-split state machine has less states. We implemented this algorithm and our experiments show that the bit-split algorithm can compile the Snort rule set we use on the order of seconds.

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 I. 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 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 eight 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 right-most 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. The search algorithm is as follows.

Algorithm 2. Search for strings in binary state machines.
Input. A text string x and 8 binary state machines from $b[0]$ to $b[7]$.
Output. All matches of strings in x.

Method.

```
{
    for (i = 0; i < 8; i++)
        state[i] = 0;   // every binary state machine starts from state 0

    for ( each character in x ) {
        let k be the next character in x;
        mayoutput = true;
        for (i = 0; i < 8; i++) {
            bit = ( ( (unsigned)(k << i) ) >> (8 - 1) ) & 0x00000001 ;
            if (! b[i].search_onechar(bit,state) )
                mayoutput = false;
        }
        if (mayoutput) {
            se = set intersection of b[i].state_sets[state[i]] for all i;
            if (se is not empty) {
                print index; /*index is the location in x
                             where matches are found*/
                print output function of all states in se;
            }
        }
    } /* end of for*/
} /* end of algorithm */

bool search_onechar(int key, int *state){
// return true: output state, false: not output state

    state[pos] = move[state[pos]][key];
    if (F[state[pos]])
        return true;
    else
        return false;
}
```

3.5 Partitioning the Rules

If we put all of the more than 1000 strings into a large state machine and construct the corresponding bit-split state machines, a partial match vector of more than 1000 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. A more detailed discussion 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 fewer states in total. While this is not the optimal solution, it beats the two alternatives, dividing by length and dividing randomly. For the Snort rule set we use, 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 four state machines split from the Aho-Corasick state machine in Figure 3 mapped onto our architecture. Here instead of splitting into eight state machines, we split the Aho-Corasick state machine into four state machines, which is optimal in terms of storage, which we will show in Section 4. Each of these four state machines is responsible for two bits of an input byte. Still taking "hxhe" as an example input stream, the transitions of all of the four 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 AND RESULTS

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 Partitioning

As we mentioned in the prior section, we can divide the Aho-Corasick state machine into eight binary state machines, each of which processes only one bit at a time. While eight binary state machines is the easiest to understand, we could also split the Aho-Corasick algorithms into four state machines, each of

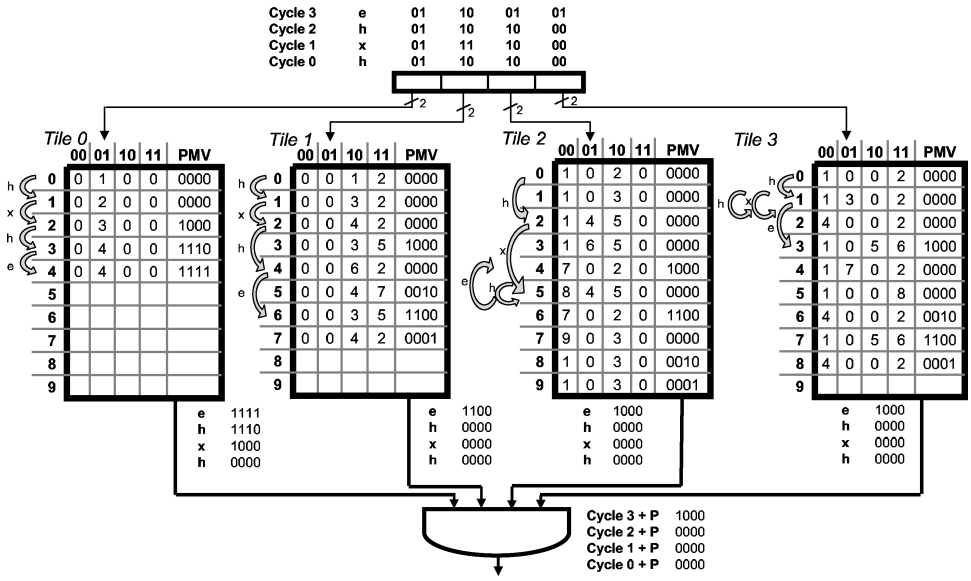

Fig. 4. The state transitions of input stream "hxhe" on the rule module for strings "he," "she," "his," and "hers." Only the first 4 bits of 16-bit partial-match vectors (PMV) are shown in this diagram because the rest of 12 bits are all zeros for only 4 strings are mapped onto the rule module. Here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

which processes two bits at a time, or two state machines that process four 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 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.

From Figure 1, we can see that the total number of bits $T_{n,g}$ is the product of the number of rule modules m, the number of tiles per rule module n, and the number of bits each tile requires. To zoom in, the number of bits each tile requires can be further expressed as the number of entries times the number of bits each entry requires. Among these two factors, the number of bits each entry requires is the number of next state pointers f times the number of bits each pointer needs p plus the number of bits the partial match vector needs g.

Let us list the set of variables we use.

- $T_{n,g}$, Total number of bits our architecture requires
- S, Total number of strings in a rule set
- L, Number of characters per string on average
- n, Module size = number of tiles per rule module = number of state machines into which each Aho-Corasick state machine is split
- g, Group size = number of bits in a partial match vector

Table II. Optimal Module Sizes[a]

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

[a] $\lceil \log_2(gL) \rceil$ is denoted by p for clarity.

- m, Number of rule modules
- p, Number of bits for a next state pointer
- f, Fanout = number of next state pointers for each state

Therefore, the total number of bits each entry requires is $fp+g$, the number of bits each tile requires is $2^p(fp+g)$, and the total number of bits our architecture requires $T_{n,g}$ is $mn2^p(fp+g)$.

The two variables n and g are the actual parameters that we want to tune, and the other temporary variables can be expressed as functions of n and g. m is, at least, the the ceiling of S divided by g. p is related to the number of states of each state machine, which will not exceed the number of characters in each group. p is on average $\lceil \log_2(gL) \rceil$. If an Aho-Corasick state machine is split into n state machines, then each of the n state machines only has to deal with $8/n$ bits, which can represent $2^{8/n}$ next state pointers. Therefore, f is $2^{8/n}$.

We now have obtained the following equations.

- $T_{n,g} = mn2^p(fp+g)$
- $m = \lceil \frac{S}{g} \rceil$
- $p = \lceil \log_2(gL) \rceil$
- $f = 2^{\frac{8}{n}}$

Plugging m, p, and f into $T_{n,g}$, we obtain the approximate total number of bits our architecture requires,

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

S and L are constants for a given rule set.

From this formula, we can see that the smaller the group size g is the smaller $T_{n,g}$ is. (By removing the first two ceiling functions, $T_{n,g}$ becomes $nSL(\lceil \log_2(gL) \rceil 2^{\frac{8}{n}} + g)$).

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 II, where fanout is the number of next state pointers for each state.

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

4.2 Practical Optimal Partitioning

We have obtained the theoretical optimal parameters for our architecture in the previous section. We are now going to confirm some of these results, point

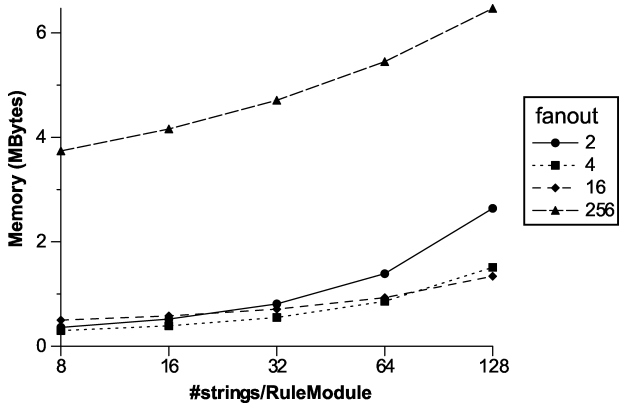

Fig. 5. Practical memory comparison for different fanout and group sizes. X-axis is the number of strings per rule module, also referred to as group sizes, on a logarithmic scale. The four lines correspond to data for four different fanout. Ideally, the trend of decreasing memory size should continue for less than eight strings per rule module. To make the best use of space, we, however, have to decrease the total number states per tile as we decrease the group size. However, there is a lower bound on the total number of states per tile imposed by the length of the longest string in the rule set (further explained in Section 4.2). Therefore, the smallest efficient group size for the Snort rule set we use is eight strings per rule module.

out some of the problems with 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 $T_{n,g}$, $\lceil S/g \rceil$, 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 seven bits are needed. In short, p values that are 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.

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. The four lines in the figure correspond to data for four different levels of state-machine fanout. We can see from the graph that the line for a fanout of 256 is high above the other lines, which indicates that the traditional way to implement state machines (with 256 next state pointers) uses far more storage than our bit-split state machines. Even if the group size is as small as 8, 3.74 MB are needed, which is more than seven 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 with group size 8 and 16. These configurations use only 0.4 MB to store the entire

Snort rule set. We chose a group size of 16, which allows for longer strings, with the concern that string length is growing and that larger tiles sizes cause 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, noninterrupting update and area efficiency. Here we compare these three requirements on our design and a number of other designs. In Section 2.3, we described our architectural support for incremental and noninterrupting 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 III, we can see that our design can achieve worst-case throughput of over 10 Gbit/sec, even if only one byte is read in each cycle, while the best of all FPGA-based methods we examined can only achieve a throughput of just over 3 Gbit/sec with this read-in rate. Even the smallest throughput configuration of our design handles over 8 Gbit/sec, with a great increase in area efficiency and performance per area. In addition to throughput, we compare area efficiency (in char/mm^2) among different designs. We explore tradeoffs in SRAM memory bank sizes using a modified version of CACTI 3.2 [Shivakumar and Jouppi 2001]. 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 an area efficiency of 320.972 characters/mm^2, which is more than four 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 design and FPGA-based designs. The X-axis is the area efficiency, the number of characters per square millimeter the design 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. Thus, 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 configurations 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. [2004] optimized the Aho-Corasick algorithm by looking at bitmap compression and path compression to reduce the amount of memory needed to 2.8 and 1.1 MB, respectively, which are still at least about three times of that of our design, which is only requires 0.4 MB.

5. MULTIPLE BYTES PER CYCLE

The methods described in the previous sections allow for a compact design that can be efficiently replicated if higher bandwidths are required.

Table III. Detailed Comparison of Bit-Split FSM and FPGA-Based Designs[a]

Description	Throughput (Gbps)	Char/Area ($1/mm^2$)	Throughput* Char/Area ($Gbps/mm^2$)	Notes
Bit-Split FSM (12,812 characters 1,053 strings Group Size 16)	10.074	55.219	556.306	Bank size 64 B
	9.759	72.592	708.424	Bank size 128 B
	9.326	156.569	1460.092	Bank size 256 B
	9.042	198.442	1794.316	Bank size 512 B
	8.706	285.676	2487.194	Bank size 1024 B
	8.408	320.972	2699.210	Bank size 2048 B
[Cho and Mangione-Smith 2004]	6.400	33.222	212.618	Spartan3-5000
	3.200	92.846	297.108	Spartan3-2000 Requires additional 90KB Block Mem.
[Aldwairi et al. 2004]	10.100	5.581	56.370	8B/cc, Altera EP20K
	5.000	23.516	117.582	4B/cc, Altera EP20K
[Sourdis and Pnevmatikatos 2004] Pre-decoded CAMs	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, $g = 64$
	2.975	76.035	226.203	Virtex2-3000, $g = 128$
	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. 2002] Regular expressions	0.248	32.496	8.059	1B/cc, Virtex-1000
	0.400	32.496	12.998	1B/cc, Virtex-1000
	0.396	33.353	13.208	1B/cc, Virtex-2000
[Cho et al. 2002] Dis. comparators	2.880	∼7.911	∼22.785	1B/cc, Altera EP20K
[Clark and Schimmel 2003] NFAs-shared decoders	0.800	∼74.733	∼59.787	1B/cc, Virtex-1000

[a]Throughput, density, and efficiency are shown for a variety of different design options. g, group size. 1 B/cc = read in one byte per cycle time.

Replication, while useful, requires that there are multiple distinct streams of data to be processed, for example, independent network packets. However, in some situations there may be only a single stream of data to search. In this case, one way that our method can be extended to provide adequate bandwidth is to process multiple bytes of the input stream at a time.

Processing multiple bytes of data at a time is analogous to a multibit trie, instead of using only a single byte of the input stream to determine the next state, two or more of the bytes of the input stream are used. Applying this idea to the original Aho-Corasick state machines could be problematic, as each state already has 256 next state pointers. To make the state machine read two bytes at a time, each state would have 256^2 next state pointers and doing n bytes at a time will require 256^n out edges per state. This explosion of transitions and states suggests that multibyte transitions may not be feasible for the full Aho-Corasick state machines.

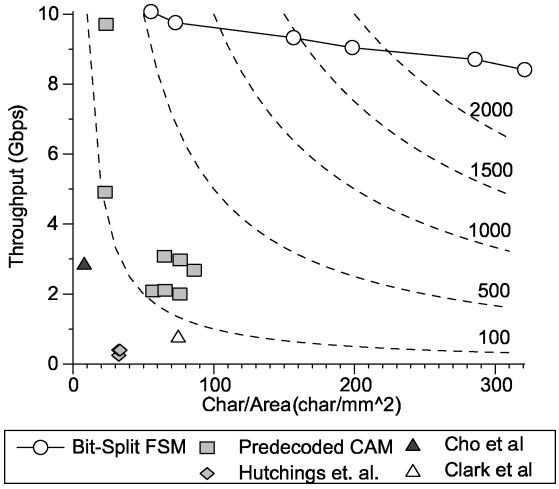

Fig. 6. Efficiency (throughput*char/area) Comparison of string-matching designs. Each dashed line is the aggregation of all points that have the same performance per area value. The points on the upper right part denote more efficient designs. We can clearly see that even the least efficient configuration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

Things become significantly easier with our bit-split algorithm, as it reduces the number of next state pointers to exactly 2 for each state. This property allows us to read multiple bytes per cycle in practice. For example, to read two bytes per cycle, we only need to build binary state machines that each reads two bits per cycle (with four out edges). In this section we describe a multi-byte extension to our bit-split engine to approach higher bandwidths when replication cannot be employed.

5.1 A Multibyte Algorithm

Given eight binary state machines that accept a set of strings one byte at a time (as described in Section 3), eight new state machines that operate together to read two bytes at a time can be constructed as follows. Each new state machine reads *two* bits at a time so that these eight state machines together can deal with two bytes at a time. The construction process is identical to and independent from each of the eight new state machines.

To build a two-bit ahead state machine from a regular bit-split binary state machine, we just collapse two adjacent edges into one edge. Take B_3 (the state machine in the middle) in Figure 3 as an example. From state 0, it directly transitions to state 3 with two state sets {} and {2} by reading in two bits "00." State set {} indicates there is no output for the first input bit and state set {2} is the output set for the second input bit. Thus, now each node has four next nodes, which are for two-bit input "00," "01," "10," and "11," respectively. Two state sets are kept for each node. One state set indicates partial matches for the first bit of the two bits and the other is for the second bit. If the two state sets in two nodes are not exactly the same, these two nodes are considered as two different nodes. For example, the next state for state 0 by reading in two

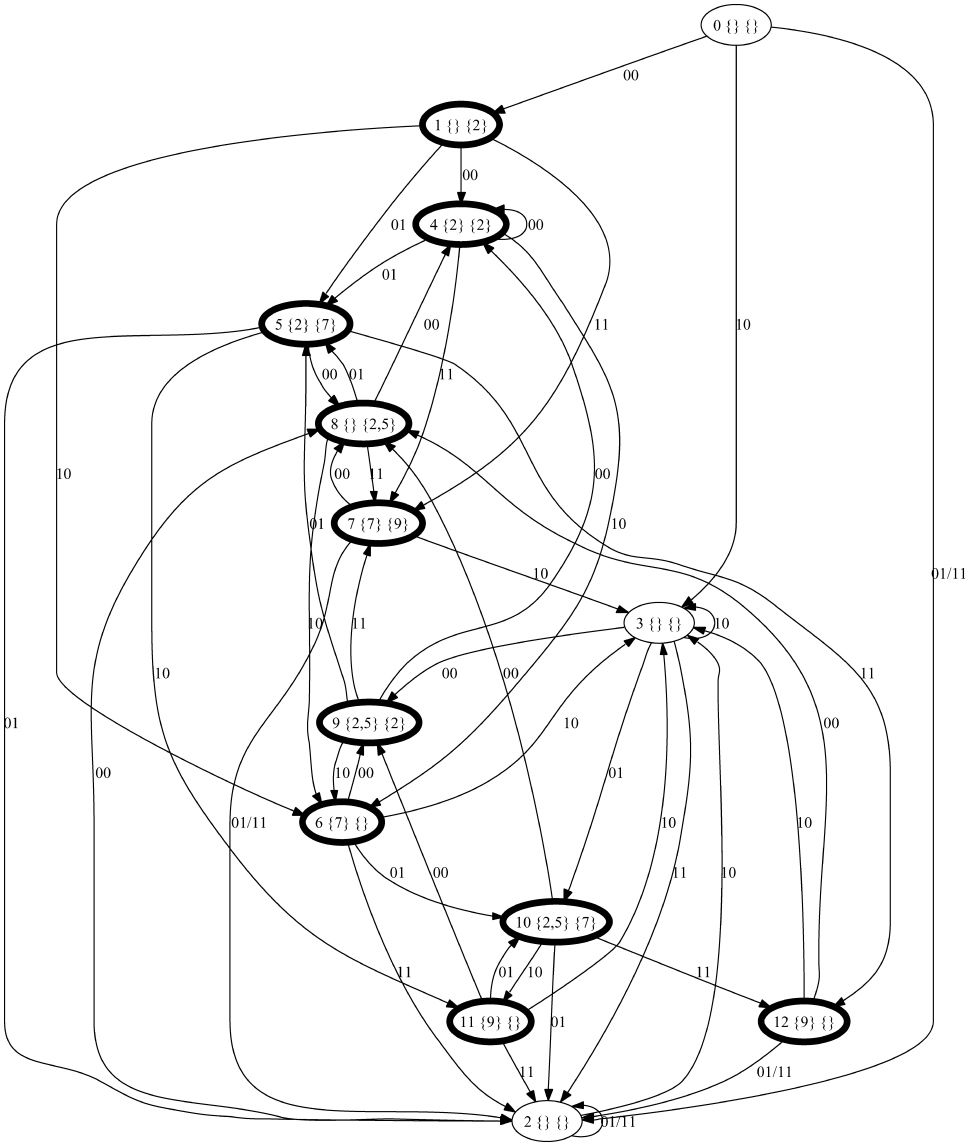

Fig. 7. The 2-bit ahead state machine constructed from B_3 in Figure 3.

input bits "00" is state 3 with state set {} and {2}. Also, the next state for state 3 by reading in two input bits "00" is state 3 with state set {2} and {2}. However, these two next states are two different states because the first state sets of them are different.

The collapsing process is continued until no new node is generated. One example 2-bit ahead state machine is in Figure 7, which is converted from the binary state machine B_3 in Figure 3.

The search on a two-bit ahead state machine is similar to that on a regular bit-split state machine. A two-bit ahead state machine is fed with two bits per

cycle and it can output state sets on each cycle. The only difference is if the last input is only one bit, either of the edges that begin with this one bit can be used for state transition and only the first state set will be output. For example, in State 6, if the next input is only "0," we can use either "00" or "01" to transition and the first state set {2, 5} is considered for output. At each cycle, when we merge all state sets from all two-bit ahead state machines, all of the first state sets are ANDed together to produce the first final state set and, similarly, all of the second state sets are ANDed together to generate the second final state set. The first final state set and the second final state set are then output at the same cycle to indicate matches at these two bytes.

The above algorithm can be further extended to read three bits or four bits at a time. To read three bits at a time, each node has eight next states and three partial-match vectors. Similarly, to read four bits at a time, each node has 16 next states and four partial-match vectors. By reading multiple bits per cycle, we are trading area for throughput. The simulation results are reported in the results section.

5.2 Algorithm Analysis

The architecture to support a two-bit ahead state machines is very similar to that for Figure 3. Each rule module still consists of eight tiles. Each tile stores one two-bit ahead state machine. An entry in each tile has four next state pointers and two partial match vectors.

From B_3 in Figure 3 and Figure 7, we can see that the total number of states increases from 8 to 13. We, therefore, are interested in how many states a bit-split state machine can grow to by two-bit ahead conversion.

If the original bit-split state machine has n states, the two-bit ahead version of it will have at most $2n$ states. This is because the only scenario that a state in an original bit-split state machine, called B-state, will become multiple states, called A-states, in the two-bit ahead version is that this B-state has multiple incoming edges and the source states of these edges have different state sets. Then, when the B-state is reached, different first state sets will require different A states to be created. If a B-state has x incoming edges, it will be converted into at most x A states in the corresponding two-bit ahead state machine. Therefore, because an original bit-split state machine with n states has $2n$ incoming edges, its corresponding two-bit ahead state machine has at most $2n$ states. If many source states of incoming edges of one B-state have the same state set, then the number of states in the two-bit ahead state machine will be less than $2n$.

In general, the upper bound of the number of states for a k-bit ahead state machine is $n2^{k-1}$. For example, the upper bound for a three-bit ahead state machine is $4n$ and the upper bound for a four-bit ahead state machine is $8n$. The brief idea is as follows. For each B-state B_j, $j = 1, 2, \ldots n$, a new A-state may be created for each path of length k – 1 from a B-state B_i to B_j. This is the only case that a new A-state can be created. Therefore, the total number of A-states is, at most, the total number of paths of length k – 1 with B_j as the destination for $j = 1, 2, \ldots n$. Therefore, we now only need to show that the total number of paths of length k – 1 with B_j as the destination for $j = 1, 2, \ldots n$

Table IV. Throughput and Area Results of Multiple Bits per Cycle[a]

No. of Bytes Per Cycle	Throughput (Gbps)	Throughput* Char/Area ($Gbps/mm^2$)	Bank Size
4	36.089	93.607	64 B
	29.732	180.104	8192 B
3	28.464	161.718	64 B
	25.304	568.997	2048 B
2	20.149	269.972	64 B
	16.819	1309.911	2048 B
1	10.074	556.306	64 B
	8.408	2699.210	2048 B

[a]The group size is 8.

is at most $n2^{k-1}$. This number can be calculated from a different angle, which means the total number of paths of length k – 1 with B_j as the destination for $j = 1, 2, \ldots n$ is the same as the total number of paths of length k – 1 with B_j as the source for $j = 1, 2, \ldots n$. Since each B-state has exactly two out edges, the number of paths of length k – 1 from B_j is 2^{k-1}. Combining this fact with that there are n states in total in the original bit-split state machine, we can draw the conclusion that the total number of paths of length k – 1 from state B_j is, at most, $n2^{k-1}$.

5.3 Results

To estimate the throughput and area of using multiple bytes per cycle, we use the same methods as presented in Section 4. Since we need to store more than one partial-match vector for each state for multiple byte implementation, we choose the smallest feasible size (as discussed in Section 4), which is eight strings per group.

The highest throughput we achieve is 36 Gbps by reading four bytes per cycle, but this comes at a high price in terms of area. We can also see from Table IV, that the smaller the bank size, the higher the throughput, but again the downside is larger area. The results in Table IV assume the worst possible case for state explosion, and a more clever reduction scheme that takes into consideration the nature of the actual state machine could likely reduce this significantly. That being said, the results for a four-bytes per cycle state machine in the worst-case is prohibitively expensive for the entire Snort data set (with an estimated area of just under 50 cm^2). However, if a string-matching application is required that has fewer strings, or some of the rules from Snort can be safely removed, this could easily become a feasible approach. While research is now ongoing into more efficient implementation methods for large numbers of bytes at once, the case where the bytes-per-cycle is set to 2, the scaling is more reasonable. Doing two-bytes per cycle instead of one results in a drop in Gbps/mm^2 of just over a factor of 2.

6. FUNCTIONAL PROTOTYPING

To ensure that our string-matching architecture is both functionally correct and fully implementable as a hardware module, we have implemented a prototype system. Using this system, built primarily from a single network enabled FPGA

board, we have flushed out any remaining bugs and demonstrated a working interface. The two major implementation issues that arise in constructing a real-system-based on our architecture are: how our memory-based design could be mapped to a modern FPGA device and how it could be interfaced with higher level software layers.

6.1 Mapping the String-Match Engine

To prototype our string-match system, we chose to build upon the Altera Stratix family of reconfigurable devices. In our initial tests we used the smaller EP1S10 FPGA, which has a total of 10,570 Logic Elements (LEs), a single 512-Kbit M-RAM block, 60 4-Kbit M4k RAM blocks, and 94 smaller 512-bit M512 RAM blocks. The prototyping board we used also features a built in 10/100 Ethernet interface, 8 MB of Flash memory, a two line LCD display, and a Mictor Trace/Debug Connector. The software layers of our system are built on top of the Altera configurable softcore processor, the NIOS II. The soft-core processor serves as the interface between the Ethernet and the string-matching engine and controls the writing of rules into the rule module memory. The soft-core processor is configured using a GUI builder, which comes with the Quartus tool chain, while the design of the string-match engine is implemented completely in Verilog.

Each rule module takes up approximately 150 LEs and 48 Kbits (6 KB) of memory. The 150 LEs are used for the logic that controls the memory, the string-match engine itself, and to implement the register file. Because the ASIC design described in Section 2 is dominated by memory devices, the limiting factor in prototyping our system is the amount of embedded memory available on the FPGA. The memory is used to store the state machine data. Each state requires 48 bits of data, 32 bits for the next state data (four possible next states each requiring eight bits), and 16 bits for the output of that state (the partial-match vector). Each tile, therefore, uses 3 M4k blocks of the Stratix's on-chip memory. This means that for each rule module, a total of 12 blocks of M4k memory are used. Having the memory for the different tiles on different blocks is essential for the function of this design. In order to maintain maximum throughput all of the tiles need to be operated in parallel.

Since each tile needs to be operating in parallel, each tile must have its own separate memory block, and thus each rule module must have four separate memory blocks, one for each tile. The small EP1S10 board only has enough separate memory blocks to allow six rule modules (using all the M4k and M512 blocks). However, our more aggressive Stratix devices have far more memory and will allow for enough rule modules to fit over 500 rules. We are currently in the process of changing over our design to use these larger chips, as well as high-speed optical network connections.

Our first implementation can sustain an aggregate throughput of one byte of data per clock cycle running at over 150 MHz, which gives us a data rate of over 1 GB/sec. While our prototype requires further circuit level optimization, this network speed is sufficient to allow deployment in a real network.

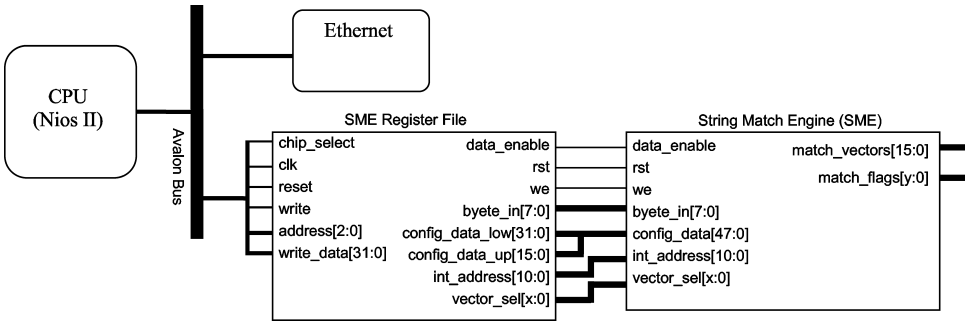

Fig. 8. The interface between the string-match engine, the NIOS II and Ethernet. The width of `match_flags` and `vector_sel` varies with the number of rule modules in the string-match engine. Processor communicates with SME and Ethernet through the data bus. All control of SME is done by writing to the register file. Although not shown in this diagram, outputs of string-match engine go back to the processor.

Unfortunately our current implementation has a bottleneck unrelated to the string matching engine.

While the string-matching core is currently capable of 1 GB/sec, we currently cannot currently deliver that rate of data to our device because of the interaction with the processor. The NIOS II Processor can only run at a speed of 50 MHz, which, because of its tight integration with our initial prototype, effectively limits the capabilities of the system. While the dependence on the NIOS II in our design limits our bandwidth, it does provide us with a much quicker development cycle for testing our prototype. A further improvement to our system would be to use DMA to directly pipe data between the Ethernet device and the string-match engine. The exact interface we have used is described in Section 6.2.

6.2 Coordinating with IDS Systems

A significant advantage of our memory-based string match engine is that it has a very simple and easy to use interface (shown in Figure 8). Control is coordinated through a register file in the string-match engine that is memory mapped into the processor's address space. The entire interface is accessible over an internal bus. The register file has eight registers, which control the functionality of the SME. The first register, `byte_in`, accepts in a byte of data stream to be searched as input. The next register is `data enable`, which enables the string-match engine. The `config_data_low` register takes in the lower 32 bits of data to be written to a tile in one of the rule modules, while the `config_data_up` register takes in the upper 16 bits. The `int_address` register takes the address of memory to write to. The address indicates which rule module, tile, and state in that tile to write to. Each write copies over an entire state worth of data, including the next-state transitions and the partial-match vector. The `rst` register, resets the state machines and the outputs of the string-match engine. The `we` register is a write enable for writing to a tile's memory. Last, the `vector_sel` register selects which rule module has its match-vector output.

The string-match engine has two outputs, a 16-bit match vector, match_vectors, and a second output with one bit for each rule module, match_flags. The match_flags output is flags, which indicate what rule modules found a match. For example, if rule module number 3 had a match for any one of its 16 rules then bit 3 of the second output would be 1, indicating a match in that rule module. The match_vectors output outputs the match vector of whatever rule module is currently selected by the vector_sel register. Both of these outputs are sent back to the NIOS II processor. The NIOS II reads in both of the outputs through a parallel I/O module, then with this information the processor can determine which rule modules and what rules within the modules were set off and then notify the user all the matches for a packet of data.

There are several functions which are used to control the string match engine. All software and drivers for the SME are written in C and compiled and tested using the NIOS IDE. Our custom driver has several functions for controlling the SME. The first function:

```
sme_write_tile(unsigned int base,       unsigned int module_num,
               unsigned int tile_num,   unsigned int address,
               unsigned int data_upper, unsigned int data_lower)
```

writes one line of memory (48 bits wide) to a given tile inside a given rule module. This is used for initialization/updating of rules. The function first generates the correct address using the tile number, address, and rule module number. It then writes the corresponding values to the int_address, config_data_low, config_data_up registers. Last, it sets the we register so the data is written to memory. The second function

```
sme_send_byte(unsigned int base, unsigned int byte_from_packet)
```

sends a byte of data to the string match engine. This function will likely be replaced with a DMA setup-tear down routine in the next iteration of our design. In addition to these functions, there is also a function for initializing the SME and one to enable and disable it. These low level routines are used to create higher level functions in the software. The main higher level function sends a packet of data to the SME, informs the user of all matches in the packet and resets the SME so it is ready for the next packet. The rule compiler can also be run on the NIOS-II to create a fully self-contained embedded packet scan architecture. Updating of rules could be done over the network by sending specially encoded packets to the string match engine, which provide it with the necessary data to update the rule modules.

7. 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.

7.1 Software-Based

Most software-based techniques concentrate on the reducing the common case performance. Boyer–Moore [Boyer and Moore 1977] is a prime example of such technique, as it lets its user search for strings in sublinear 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 [Fisk and Varghese] 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. [2004] takes a different approach to optimizing Aho-Corasick by instead looking at bitmap and path compression to reduce the amount of memory needed.

7.2 FPGA-Based

The area that has seen the most amount of string-matching research is in the reconfigurable computing community [Clark and Schimmel 2003; Hutchings et al. 2002; Sourdis and Pnevmatikatos 2004; Gokhale et al. 2002; Baker and Prasanna 2004a, 2004b; Cho et al. 2002; Dharmapurikar et al. 2004; Cho and Mangione-Smith 2004]. 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 over time but only relatively slowly. Because FPGAs are inherently reconfigurable, the majority of prior work in this area focuses on efficient ways to map 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 [2004] describes an approach that is specifically tuned to the hardware resource available to devices available from Xilinx to provide near optimal resource utilization and performance. Because they demonstrate that 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 ten 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 architectures. For example Cho and Mangione-Smith presented a technique for implementing state machines on block-RAMs in FPGAs [Cho and Mangione-Smith 2004] and concurrent to our work. Aldwairi et al. [2004] proposed mapping

state machines to on-chip SRAM. In their work, the packet header is used for preclassification, so that packets are only matched against a subset of rules and some measure of parallelism is exploited. If there are system level reasons to divide the packets in this way, then the two approaches are complimentary and could be combined. Another area where our optimizations would be valuable is when the application is mapped down to more general-purpose programmable memory tiles [Mai et al. 2000; Taylor et al. 2004; Swanson et al. 2003].

8. CONCLUSIONS

While in this paper we examine the use of our technique strictly for intrusion detection with Snort, our methodology is general 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) [SpamDetection]. 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 [Tanenbaum et al. 1982; McKenzie 1989]. 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 [Sanchez et al. 2001] and packet classification [Gupta and McKeown 2001].

In addition to improve string-matching performance, our bit-split FSM scheme can be detached from string matching to be applied to general search problems on general state machines. Any such state machine problem, where there is a high fanout from each of the nodes, may be improved dramatically.

To test both the correctness and efficiency of our approach, we have successfully created a functional prototype, which shows that this design is feasible in hardware. Our design functions correctly, finding all string matches in incoming packets. By mapping our state machine tiles down to the embedded block RAM available on modern FPGAs, we have been able to build a usable prototype that can be deployed in real networks.

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 that 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.

ACKNOWLEDGMENTS

The authors would like to thank Fred Chong, Ryan Dixon, Ömer Egecioglu, Andrew Petersen, George Varghese, and the anonymous reviewers for thier suggestions relating to this paper. This work was funded in part by NSF Career Grant CCF-0448654.

REFERENCES

AHO, A. V. AND CORASICK, M. J. 1975. Efficient string matching: An aid to bibliographic search. *Communications of the ACM 18*, 6, 333–340.

ALDWAIRI, M., CONTE, T., AND FRANZON, P. 2004. Configurable string matching hardware for speedup up intrusion detection. In *Workshop on Architectural Support for Security and Anti-virus (WASSA) Held in Cooperation with ASPLOS XI*.

BAKER, Z. K. AND PRASANNA, V. K. 2004a. A methodology for synthesis of efficient intrusion detection systems on FPGAs. In *Proceedings of the Field-Programmable Custom Computing Machines*. 135–144.

BAKER, Z. K. AND PRASANNA, V. K. 2004b. Time and area efficient pattern matching on FPGAs. In *Proceeding of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*. 223–232.

BARATLOO, A., SINGH, N., AND TSAI, T. 2000. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Security Symposium*.

BOYER, R. S. AND MOORE, J. S. 1977. A fast string searching algorithm. *Communications of the ACM 20*, 10, 761–772.

CHO, Y. AND MANGIONE-SMITH, W. 2004. Deep packet filter with dedicated logic and read only memories. In *IEEE Symposium on Field-Programmable Custom Computing Machines*.

CHO, Y. H., NAVAB, S., AND MANGIONE-SMITH, W. H. 2002. Specialized hardware for deep network packet filtering. In *12th International Converence on Field-Programmable Logic and Applications*.

CLARK, C. R. AND SCHIMMEL, D. E. 2003. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Proceedings of the 13th International Conference on Field Programmable Logic and Applications*.

CROSBY, S. A. AND WALLACH, D. S. 2003. Denial of service via algorithmic complexity attacks. In *Proceedings of USENIX Annual Technical Conference*.

DHARMAPURIKAR, S., ATTIG, M., AND LOCKWOOD, J. 2004. Deep packet inspection using parallel bloom filters. *Micro, IEEE 24*, 1, 52–61.

FISK, M. AND VARGHESE, G. 2001. Applying fast string matching to intrusion detection. Tech. Rep. In preparation, successor to UCSD TR CS2001-0670, University of California, San Diego.

GOKHALE, M., DUBOIS, D., DUBOIS, A., BOORMAN, M., POOLE, S., AND HOGSETT, V. 2002. Granidt: Towards gigabit rate network intrusion detection technology. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications*. 404–413.

GUPTA, P. AND MCKEOWN, N. 2001. Algorithms for packet classification. *IEEE Network Magazine*.

HUTCHINGS, B. L., FRANKLIN, R., AND CARVER, D. 2002. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 111.

IDSMARKET. 2004. Intrusion detection/prevention product revenue up 9% in 1Q04. Infonetics Market Research. Tech. rep. June.

MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W., AND HOROWITZ, M. 2000. Smart memories: A modular reconfigurable architecture. In *Annual International Symposium on Computer Architecture*.

MCKENZIE, B. J. 1989. Fast peephole optimization techniques. *Softw. Pract. Exper. 19*, 12, 1151–1162.

ROESCH, M. 1999. Snort—lightweight intrusion detection for networks. In *Proceedings of LISA'99: 13th Systems Administration Conference*. 229–238.

SANCHEZ, M., BIERSACK, E., AND DABBOUS, W. 2001. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Magazine 15*, 2, 8–23.

SHIVAKUMAR, P. AND JOUPPI, N. 2001. CACTI 3.0: An integrated cache timing, power, and area model. Tech. Rep. WRL-2001-2, HP Labs Technical Reports. Dec.

SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. 2004. Automated worm fingerprinting. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*.

SOURDIS, I. AND PNEVMATIKATOS, D. 2004. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *Proceedings of the Field-Programmable Custom Computing Machines*. 258–267.

SPAMDETECTION. Commtouch® software ltd. White paper: Recurrent pattern detection (RPD™) technology. http:www.commtouch.com/documents/
Commtouch RPD White Paper.pdf.

SRINIVASAN, V. AND VARGHESE, G. 1999. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems 7*, 1 (Feb.), 1–40.

SWANSON, S., MICHELSON, K., SCHWERIN, A., AND OSKIN, M. 2003. Wavescalar. In *36th International Symposium on Microarchitecture*.

TANENBAUM, A. S., VAN STAVEREN, H., AND STEVENSON, J. W. 1982. Using peephole optimization on intermediate code. *ACM Trans. Program. Lang. Syst. 4*, 1, 21–36.

TAYLOR, M. B., LEE, W., MILLER, J., WENTZLAFF, D., BRATT, I., GREENWALD, B., HENRY, HOFFMANN, JOHNSON, P., KIM, J., PSOTA, J., SARAF, A., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2004. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Annual International Symposium on Computer Architecture*.

TUCK, N., SHERWOOD, T., CALDER, B., AND VARGHESE, G. 2004. Deterministic memory-efficient string matching algorithms for intrusion detection. In *the 23rd Conference of the IEEE Communications Society (Infocomm)*.

XU, J., KALBARCZYK, Z., PATEL, S., AND IYER, R. K. 2002. Architecture support for defending against buffer overflow attacks. In *Workshop on Evaluating and Architecting Systems for Dependability*.

Received August 2005; revised November 2005; accepted January 2006