# Architectures for Bit-Split String Scanning in Intrusion Detection

STRING MATCHING IS A CRITICAL ELEMENT OF MODERN INTRUSION DETECTION SYSTEMS BECAUSE IT LETS A SYSTEM MAKE DECISIONS BASED NOT JUST ON HEADERS, BUT ACTUAL CONTENT FLOWING THROUGH THE NETWORK. THROUGH CAREFUL CODESIGN AND OPTIMIZATION OF AN ARCHITECTURE WITH A NEW STRING MATCHING ALGORITHM, THE AUTHORS SHOW IT IS POSSIBLE TO BUILD A SYSTEM THAT IS ALMOST 12 TIMES MORE EFFICIENT THAN THE CURRENTLY BEST KNOWN APPROACHES.

**Lin Tan**
University of Illinois, Urbana-Champaign

**Timothy Sherwood**
University of California, Santa Barbara

Whether tethered to an Ethernet cable or connected through wireless technology, computer systems now operate in an environment of near ubiquitous connectivity. The availability of always-on communication has created countless opportunities for Web-based businesses, information sharing, and coordination, but it has also created new opportunities for those who seek to illegally disrupt, subvert, or attack these activities. Every day, additional critical data becomes accessible over the network, and any publicly accessible system on the Internet is subject to more than one break-in attempt per day. Because we are all increasingly at risk, interest in combating these attacks at every level is widespread, from end hosts and network taps to edge and core routers. Intrusion detection and prevention has proven highly effective at finding and blocking known attacks in the network before the end host even encounters them, but making such protection scalable entails significant computational challenges. Intrusion detection systems must scan every byte of every packet to find the signatures of known attacks, and this requires very-high-throughput methods for string matching.

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 this 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 each rule's bits. Our new algorithms are specifically tailored toward implementation in an architecture built up as an array of small memory tiles, and we developed the software and the architecture together. This article summarizes the key findings from a longer article.[1] Our efforts result in a device that maintains tight worst-case bounds on performance, is updatable with



Published by the IEEE Computer Society

new rules without interrupting operation, has configurations generated in seconds instead of hours, and is 10 times more efficient than existing best-known solutions. In particular we describe

- a novel configurable string-matching architecture that can store the entire Snort rule set—about 1,000 strings that are 12 bytes apiece, on average—in only 0.4 Mbytes and can operate at upward of 10 Gbps per instance;
- string-matching algorithms that operate through the conjunction of many small state machines working in unison, reducing the number of required out edges from 256 to as few as two; and
- a rule compiler that takes only seconds to partition and bit split a finite-state machine representation of the strings into a set of small implementable state transition tables used to program our architecture.

## Detecting intrusions

Given the importance of protecting information and services, the security community has put much effort into detecting and thwarting attacks in the network.[2,3] Intrusion detection systems and intrusion prevention systems have emerged as two of the most promising ways to protect the network, and predictions show the market for such systems growing to $918.9 million by the end of 2007.[4]

Network-based intrusion detection systems either attempt to find examples of misuse or anomalies. Both approaches 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 management software to configure, log, and display alarms generated by lower-level processing. These two parts, working in concert, alert administrators to suspicious activities, keep logs to aid in forensics, and assist in the detection of new worms and denial-of-service attacks. The lowest level, where data is actually inspected, is where the computational challenge lies.

To define suspicious activities, most modern network intrusion detection and prevention systems rely on a set of rules applied to matching packets. At minimum, a rule consists of a type of packet to search, a string of content to match, a location at which to search for that string, and an associated action to take if the search meets all of the rule's conditions. 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 take many forms, but frequently their heart consists of strings to be matched anywhere in a packet's payload. The problem is that for accurate detection, we must be able to search every byte of every packet for a potential match from a large set of strings. For example, the Snort rule set has on the order of 1,000 strings with an average length of about 12 bytes (http://www.windowsitpro.com/WindowsSecurity/Article/ArticleID/39360/39360.html). In addition to raw processing speed, a string-matching engine must have bounded performance in the worst case to withstand a performance-based attack.[5] Because rule sets are constantly growing and changing as new threats emerge, a successful design must be quickly and automatically updatable, all while the system maintains continuous operation.

## String matching with state machines

Familiar and efficient algorithms for string matching, such as Boyer-Moore,[6] are designed to find a single string in a long input. Our problem is slightly different: We're searching for one of a set of strings from the input stream. Although simply performing multiple passes of a standard one-string matching algorithm would be functionally correct, it doesn't scale to handle the thousands of strings that modern intrusion detection systems look for. Instead, it is possible to fold the set of strings we're looking for together into a single large state machine. This method, the Aho-Corasick algorithm,[7] functions in the fgrep utility as well as in some of the latest versions of the Snort network intrusion detection system.[2] One of Aho-Corasick's biggest advantages is that it performs well even in the worst case, making it impossible for an adversary to construct a stream of packets that is difficult or impossible to scan. At a high level, our algorithm works by separating the set of strings into groups and building a small state machine for each group. Each state machine's

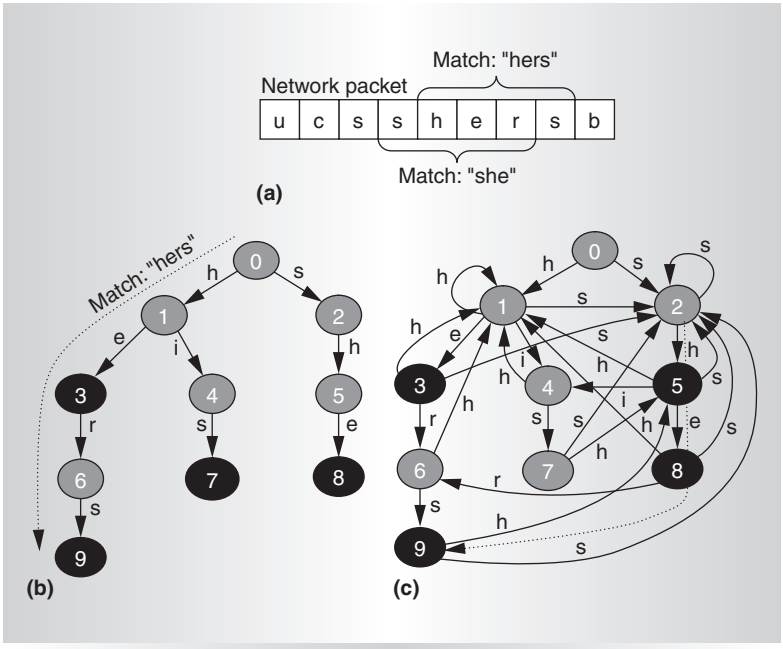


Figure 1. In this multiple string-matching problem, the input is a set of strings or patterns, S, and buffer b. The goal is to find every occurrence of an element of S in b, with the constraint that b is a stream and only one pass is allowed (a). The Aho-Corasick algorithm takes the set of all strings and builds a state machine to find them one character at a time (b) and adds failure edges to avoid backtracking (c). The state machine will recognize the appearance of any of the search strings anywhere in the entire data stream.

charge is to recognize a subset of the strings from the rule set.

This approach results in two major concerns. The first is that building a state machine from any general regular expression can, in the worst case, require an exponential number of states. We circumvent this problem by exploiting the fact that the vast majority of rules aren't general regular expressions but, in fact, are strings. For this proper and well-defined subset of regular languages, the Aho-Corasick algorithm requires only a linear number of states.

Before describing the second concern, we need to explain more about the nature of these state machines.

### Aho-Corasick algorithm

The essence of the Aho-Corasick algorithm involves a preprocessing step that creates a state machine that encodes all of the strings to be searched. This preprocessing step generates the state machine in two stages. The first stage assembles a tree of all the strings that the search must identify in the input stream. The tree's root represents the state in which 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. (This feature can serve to identify a particular 4-byte IP address, for example.) 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.

Figure 1 shows how we construct one of these state machines two steps. First, the preprocessor generates a tree from a set of target strings (this example uses the strings "he", "she", "his", and "hers"). This lets us search for any string as long as the staring point is state 0. The second preprocessing step inserts failure edges into the tree to handle the fact that any string can start to appear at any time. For clarity, we omit failure transitions back to state 0. When a string match isn't found, the suffix of one string might match the prefix of another. Inserting failure edges lets us shortcut from a partial match of one string to a partial match of another.

The advantage of this approach is that when running, it requires only one pass through the data and has excellent worst-case performance: One state transition means one byte of input has been searched. Ending up in state that corresponds to a match (for example, state 9) means the search algorithm has found a string in the list (for example, "hers"). No matter what the starting state is, the design should always end up in state 9 when the transitions for *h*, *e*, *r*, and *s* are followed.

This addresses our first concern, but if we aren't careful we'll have to support 256 possible edges (every possible byte will require either a tree edge or a failure edge) on each and every node in the state machine. This results in a huge data structure that can be neither stored nor traversed efficiently.

### A bit-split state machine

Although it's possible to use the Aho-Corasick state machines to search a stream of data with just a constant amount of time required per character, a real implementation requires large amounts of storage and a dependent memory reference for each character searched. Storing each state as an array of 256 next

pointers is wasteful. Furthermore, the number of next pointers that any given state needs varies widely. Nodes near the root of the tree need more than 200 next pointers, while nodes near the leaves need only one or two. We need a way to break this problem into a set of smaller problems, each with more-regular behavior.

To solve this problem, we split each Aho-Corasick state machine into a new set of eight state machines. Each state machine is then responsible for only one of an input character's eight bits.

This technique has three advantages:

- 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 run independently (assuming we can merge the results).
- The splitting technique will never result in a blow-up of states for string matching, and preprocessing can be completed in time linear with the number of bytes in the rule set (on the order of tens of seconds for the Snort rule set).

From state machine *D*, constructed with the Aho-Corasick algorithm, our algorithm extracts each bit of the 8-bit ASCII code 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 (one per bit). For each bit position *i*, we take the following steps to build binary state machine $B_i$. Beginning with 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 that transition with bit *i* set to 0. These sets become two new states in $B_i$. This process repeats 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*. We've described the details of the algorithm for splitting the state machines apart more fully in another publication;[1] in this article we focus on how to run the bit-split machines and how to efficiently implement them in our architecture.

Figure 2 shows how the binary state machines work together to match any string at any offset. Each state machine recognizes only one input bit at a time. It's still necessary to store the mapping of output states in *D* to all the states in the binary state machines. Because each output state in *D* corresponds to a string in the rule set, the mapping of output states can be used to discover when a binary state machine has matched a string. A resulting state in $B_i$ is an accepting state if it maps back to any of D's accepting states. Figure 2 shows that the third state machine is looking only at the stream of bits 100001000. When it detects 001, it knows that it could be seeing the bit-slice of the string "she." When it sees 0100, this could be the string "hers."

Implementing the state machine from Figure 1 requires that each state have up to 256 possible next states, which means memory will be required for each of 256 pointers for each of the 10,000 or so states. Figure 2 illustrates our proposed alternative.

Of course, simply knowing that 001 has occurred at bit 3 is insufficient to identify the string "she." One-eighth of all possible three-character sequences will have 001 for their third bit. (If there are $256^3$ possible 3-byte character sequences, $128^3$ of them will have 001 for their third bit.) In fact, many of the other search strings could have 001 in bit 3. To be certain of a match, we must consider all the potential matches of all the different state machines. At first glance, this might seem like a complicated operation, but it can be done simply and efficiently.

## Architectural implementation

Our architecture is hierarchical, in recognition of the way the sets of strings are broken down. At the highest level is the full device. Each full device holds the entire set of search strings, and each cycle the device reads in a character from an incoming packet and computes the set of matches. Matches can be reported after every byte or on a per-packet basis after they have accumulated. To multiply the throughput, we can replicate devices and have one packet go to each device in a load-balanced manner, but in this article we concentrate on a single device.

Each device contains a set of rule modules.

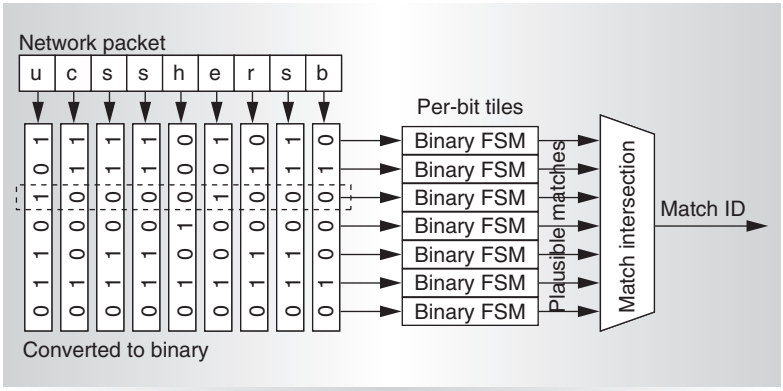


Figure 2. Bit-splitting the Aho-Corasick state machines. We must resolve the problem of having too many possible out edges at each node (256 possible next bytes). Our solution is to build a set of finite-state machines (FSMs) that look at just one or two of the bits of each byte at a time. Each of these FSMs will output a list of potential matches. Only if all state machines agree on the same potential match has an actual match occurred.

The left side of Figure 3 shows how the rule modules interact with one another. Each rule module acts as a large state machine that reads in bytes and outputs string match results. Unlike in a scheme built around the reconfigurable nature of field-programmable gate arrays (FPGAs)[8-14] which compile specialized circuits from a set of rules, our approach can take advantage of SRAM's incredible density to implement the search. The rule modules, configured only through the loading of their tables, are all structurally equivalent, 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. Throughput, not latency, is the primary concern of our design; therefore, the broadcast has limited overhead because it can be deeply pipelined if necessary.

Our preprocessing method partitions the full set of rules among the rule modules. The partitioning method affects the total number of states required in the machine and will therefore affect the total amount of space required for an efficient implementation. Our previous paper more fully discussed how to find an efficient partitioning.[1] When a rule module finds a match, it reports that match to the device's interface so that the intrusion detection system can take the appropriate actions. What happens inside each rule module is what gives our approach both high efficiency and high throughput.

Each rule module consists of a set of tiles. The right side of Figure 3 shows the structure of each and every tile in our design. Tiles, when working together, are responsible for the actual implementation of each binary state machine that actually recognizes a string in the input. As previously mentioned, each new state machine essentially acts as a filter: If any one binary state machine says that it is not a match, then it is not a match. Only if all the binary state machines agree can an actual match be declared.

It turns out that a full set of eight binary state machines, each searching for one bit at a time, is not optimal, and there are several choices for bit splitting. The original Aho-Corasick machines had one machine with 256 possible next edges, while a binary machine would require eight machines with two out edges each. In fact, four machines, each with four out edges, is space optimal for the sizes we target.[1] The idea of bit-splitting to four machines is exactly the same as to eight machines in that each machine looks at only a slice of the input; however, in this case the slice is two bits rather than one.

Each tile is essentially a table with a certain number of entries (Figure 3 shows 256 entries), and each row in the table is a state. Each state has a partial match vector and a certain number of next-state pointers, which encode the state transitions (four possible next states appear in Figure 3, and a different pair of bits from the byte stream indexes each one). The partial match vector is a bit vector that indicates the potential for a match for every rule for which the module is responsible. This match vector stores the set of possible matches for a given state, as described in the previous section. If there are up to $g$ rules mapped to a rule module, then each state of each tile will have a $g$-bit-long partial match vector. By taking the AND of each partial match vector, 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. By encoding the list of possible matches as a bit vector, we can determine the intersection of the sets with a simple AND operation.

At the beginning of each packet, and before

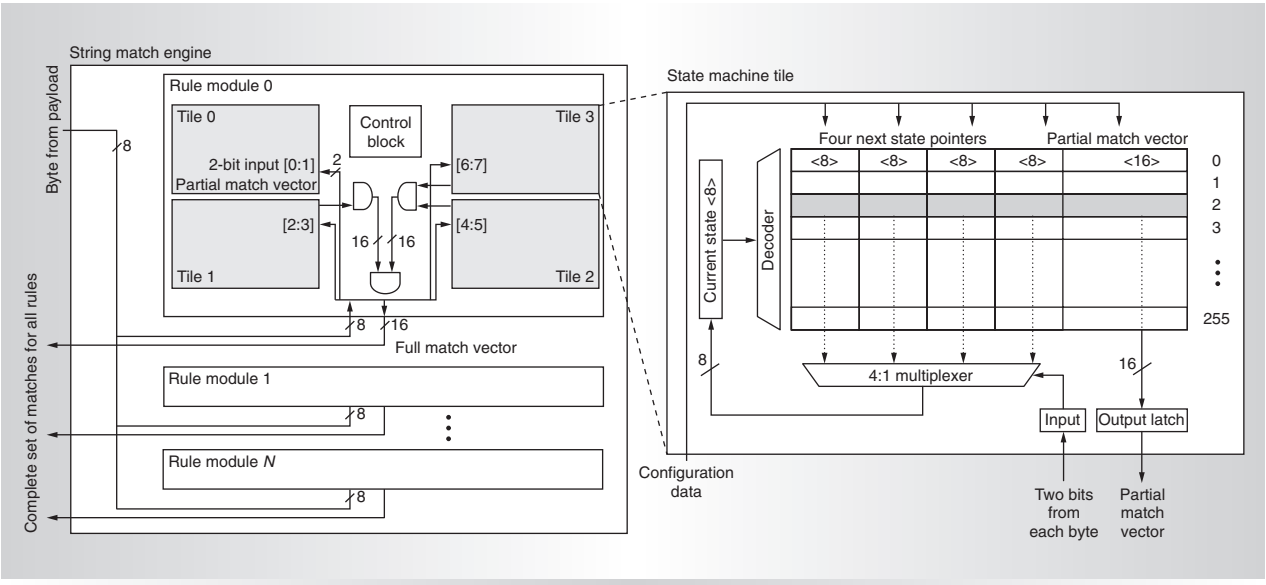


Figure 3. String-matching engine of the high-throughput architecture. The left side is a full device, consisting of a set of rule modules. Each rule module acts as a large state machine and is responsible for a group of rules (g rules). Each rule module consists of a set of tiles (four tiles in this example). The right side shows a tile's structure. Each tile is essentially a table with a certain number of entries (256 entries in this example), and each row in the table is a state. Each state has a certain number of next state pointers (four possible next states appear here) and a partial match vector of length g (g = 16 in this example). A rule module takes one character (8 bits) as input at each cycle and outputs the logical AND operation result of the partial match vectors of each tile.

the device accepts any input characters, all tiles need to be reset to start from state 0. On each cycle, a module divides the input byte into groups of bits (in the example, eight bits can be divided into four groups of two). Each tile then has its own group of bits and uses its own internal state to index a line in the memory tile. The tile reads out the partial-match vector, along with the set of possible state transitions from the memory. It then uses the input bits to select the next state for updating, and the partial match vector goes to an AND unit where it combines with the other partial match vectors. Finally, the full device concatenates all full match vectors from all modules, to indicate which of the strings were matched.

Figure 4 shows an example of the four state machines split from the Aho-Corasick state machine in Figure 1, each of which has then been mapped onto our architecture. Each of these state machines is responsible for two bits of an input byte. Figure 4 assumes the sue of "hxhe" as an example input stream and shows the transitions of all four of the state machines with arrows, starting from state 0. At each cycle each tile produces a partial match vector, and the module then outputs the logical AND of these partial match vectors. In accordance with the different requirements presented by various intrusion detection and prevention systems, our architecture can be configured to output matches only after an entire packet is scanned, instead of after each and every cycle.

## Results and conclusions

Although we don't provide detailed results in this article, we do describe two key findings from our earlier paper.[1] Two main advantages of the approach just described are increased throughput and reduced area. Because such scanning devices are often replicated on chip, we cannot consider the importance of one without considering the other. A full Aho-Corasick machine, even if it uses minimally sized pointers for each of the 256 out edges, will require at least 3.7 Mbytes of on-chip memory. With bit-split machines, we can significantly reduce this requirement to only 0.4 Mbytes. Our device's performance is really limited by the access time to one small tile of memory, but even at 1 byte per cycle,

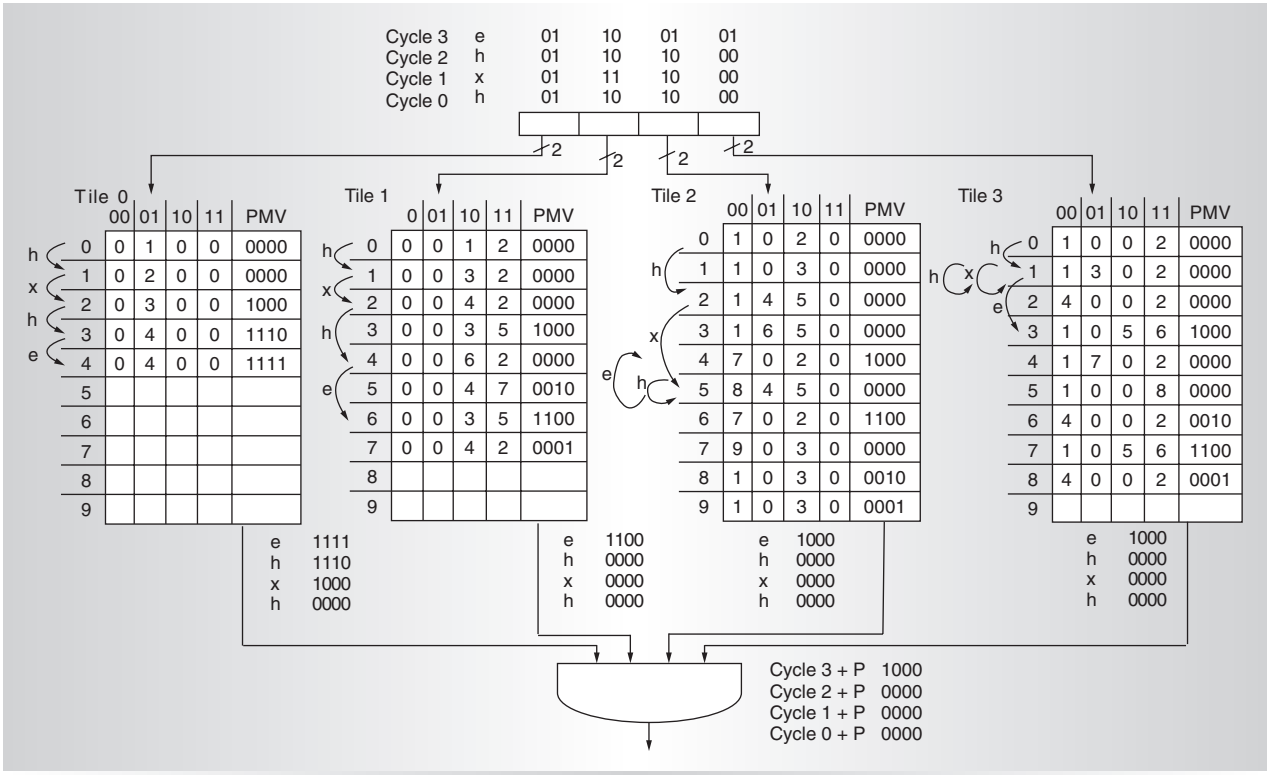


Figure 4. State transitions of input stream "hxhe" on the rule module for strings "he," "she," "his," and "hers." Given that we are only mapping four strings to this rule module in this example, only the first four bits of the 16-bit partial match vector are shown (the other bits are all zeros). Instead of eight state machines, we split the Aho-Corasick state machine into four state machines, each responsible for two bits of an input byte. The full match vector output on cycle 3 + P (where P is the delay of the system in cycles) is 1000, which shows that in this cycle a string that matches "he" has been found.

the throughput is more than 10 Gbps. As for performance density, we estimate that our design will achieve an area efficiency of 321 characters/mm$^2$ in 0.13-micron technology, which is more than four times that of the best FPGA-based designs. Our design's performance density is nearly 12 times that of the best FPGA-based designs.

Although this article explored an application-specific approach, the techniques we developed and described would likely permit the efficient mapping of string matching to other tile-based architectures used in industry. For example Cho and Mangione-Smith describe a technique for implementing state machines on block RAMs in FPGAs.[13] Concurrent with our work, Aldwairi et al. proposed mapping state machines to on-chip SRAM.[14] Another example of where our optimizations would be valuable is applications mapped down to more general-purpose programmable-memory tiles.[15]

Increasing concerns about security will almost certainly require computer systems to change. Although network intrusion detection and prevention systems are certainly not a silver bullet for the complex and dynamic security problems today's system designers face, they do provide a powerful tool. Because network intrusion detection systems require no update or modification to any of the systems they help protect, they have grown rapidly in recent years, both in computational power and rate of adoption. The architecture and algorithm we describe is small enough to be incorporated in existing network chips as a separate accelerator, it is fast and efficient enough to keep up with aggressive network speeds, and it supports always-on capability. 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 state-of-the-art string-matching algorithms. MICRO

## References


1. L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," *Proc. 32nd Ann. Int'l Symp. Computer Architecture* (ISCA 05), IEEE CS Press, 2005, pp. 112-122.
2. M. Roesch, "Snort—Lightweight Intrusion Detection for Networks," *Proc. LISA 99: 13th Systems Administration Conf.*, Usenix Assoc., 1999, pp. 229-238.
3. J. Xu et al., "Architecture Support for Defending Against Buffer Overflow Attacks," 2002; http://www.crhc.uiuc.edu/EASY/Papers02/EASY02-xu.pdf.
4. *Intrusion Detection/Prevention Product Revenue up 9% in 1Q04*, tech. report, Infonetics Market Research, 2004.
5. S.A Crosby and D.S. Wallach, "Denial of Service via Algorithmic Complexity Attacks," *Proc. Usenix Ann. Technical Conf.*, Usenix Assoc., 2003, pp. 29-44.
6. R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm," *Comm. ACM*, vol. 20, no. 10, Oct. 1977, pp. 761-772.
7. A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Comm. ACM*, vol. 18, no. 6, June 1975, pp. 333-340.
8. I. Sourdis and D. Pnevmatikatos, "Predecoded CAMs for Efficient and High-Speed NIDS Pattern Matching," *Proc. 12th Ann. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 04), IEEE CS Press, 2004, pp. 258-267.
9. Z.K. Baker and V.K. Prasanna, "A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 04), IEEE CS Press, 2004, pp. 135-144.
10. Z.K. Baker and V.K. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs," *Proc. 12th Int'l ACM/SIGDA Symp. Field-Programmable Gate Arrays* (FPGA 04), ACM Press, 2004, pp. 223-232.
11. Y.H. Cho, S. Navab, and W.H. Mangione-Smith, "Specialized Hardware for Deep Network Packet Filtering," *Proc. 12th Int'l Conf. Field-Programmable Logic and Applications* (FPL 2002), ACM Press, 2002, pp. 452-461.
12. S. Dharmapurikar, M. Attig, and J. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters," *IEEE Micro*, vol. 24, no. 1, Jan.-Feb. 2004, pp. 52-61.
13. Y. Cho and W. Mangione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," *Proc. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 04), IEEE CS Press, 2004, pp. 125-134.
14. M. Aldwairi, T. Conte, and P. Franzon, "Configurable String Matching Hardware for Speeding up Intrusion Detection," *ACM SIGARCH Computer Architecture News,* ACM Press, vol. 33, no. 1, Mar. 2005, pp. 99-107.
15. K. Mai et al., "Smart Memories: A Modular Reconfigurable Architecture," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), ACM Press, 2000, pp. 161-171.



**Lin Tan** is a PhD student in computer science at the University of Illinois, Urbana-Champaign. Her research interests include software debugging, applying machine learning techniques to improve software reliability, and specialized processors targeted to the domains of networking, communications, and security. Tan has a BS in computer science from Zhejiang University, China. She is a member of ACM.

**Timothy Sherwood** is an assistant professor in computer science at the University of California, Santa Barbara. His research interests include program phase analysis (SimPoint) and specialized processors targeted to the domains of networking, communications, and security. Sherwood has an MS and a PhD in computer science from the University of California, San Diego, and a BS in computer engineering from the University of California, Davis. He is a member of IEEE and ACM

Direct questions and comments about this article to Lin Tan, Department of Computer Science, University of Illinois, Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801-2987; lintan2@cs.uiuc.edu.