

REGULAR EXPRESSION SOFTWARE DECELERATION FOR INTRUSION DETECTION SYSTEMS¹

Zachary K. Baker, Hong-Jip Jung, and Viktor K. Prasanna

University of Southern California
Los Angeles, CA, USA
zbaker, hongjung,prasanna@usc.edu

ABSTRACT

The use of reconfigurable hardware for network security applications has recently made great strides as FPGA devices have provided larger and faster resources. Regular expressions have become a necessary and basic capability of Intrusion Detection Systems, but their implementation tends to be expensive in terms of memory cost and time performance. This work provides an architecture that reduces the exponential NFA to DFA conversion cost to a linear growth for many expressions.

By handling the timing and integration of the regular expression-based rules in a custom microcontroller, the memory costs are reduced and the capabilities are increased over a DFA-only solution. Both the microcontroller and its associated DFA are implemented on the FPGA. The patterns and software are stored using run-time programmable memory tables. This allows on-the-fly modification to the regular expressions.

This paper presents the design details of the regular expression microcontroller and its integration to the DFA state machines. The types of expressions that the system can handle efficiently are discussed as well as the outstanding problems that continue to challenge the community.

1. INTRODUCTION

The continued discovery of programming errors in network-attached software has driven the introduction of increasingly powerful and devastating attacks [1, 2]. Attacks can cause destruction of data, clogging of network links, and future breaches in security. One part of a comprehensive network security strategy is a Intrusion Detection System (IDS) at a network choke-point such as a company's connection to a trunk line (Figure 1). The IDS goes beyond the header, actually searching the packet contents for various patterns that imply an attack is taking place, or that some disallowed content is being transferred across the network. In general, an

IDS searches for a match from a set of rules that have been designed by a system administrator. These rules include information about the IP and TCP header required, and, often, a regular expression that must be located in the stream.

Early IDS rules used only string literals. Recently, rule designers have found regular expressions to be more powerful and flexible in describing patterns. The Snort database is increasingly composed of regular expressions and correlated-content expressions. Unfortunately, the processing of regular expression, or, equivalently, the simulation of a Non-deterministic Finite Automata (NFA), is complex. The memory requirements can grow exponentially when many expressions are evaluated simultaneously [3]. Software implementations had limited throughput when IDS was only the simpler string matching problem, and supporting regular expressions only increases the difficulty to maintaining a high throughput. Even with the most sophisticated algorithms, sequential microprocessor-based implementations can struggle to provide the level of service available in a customized hardware device [4].

However, support for regular expressions has only been provided in hardware [5, 6], not in a form that can be changed at runtime. The ability to quickly change the database patterns is vital, as new attacks must be imported into the system as quickly as possible. FPGA recompilations require hours. Thus, memory-based approaches only requiring a software change, such as this design, are a necessity.

This paper presents a "software-decelerated" approach for regular expression processing. In [7], the notion of moving control-intensive operations to an on-FPGA microcontroller or microprocessor was presented. By "decelerating" the complex control state machines, the FPGA hardware is freed for the true strength of the FPGA – handling a large amount of data and computation. In much the same way, our work utilizes a small microcontroller that is linked to each regular expression-matching state machine to provide more complex pattern matching functionality. The Xilinx Virtex 4 fx100 device can support about 50 of these microcontroller/DFA modules.

This paper is structured as follows: We begin with a brief

¹Supported by the United States National Science Foundation/ITR under award No. ACI-0325409 and in part by an equipment grant from the Xilinx and HP Corporations.

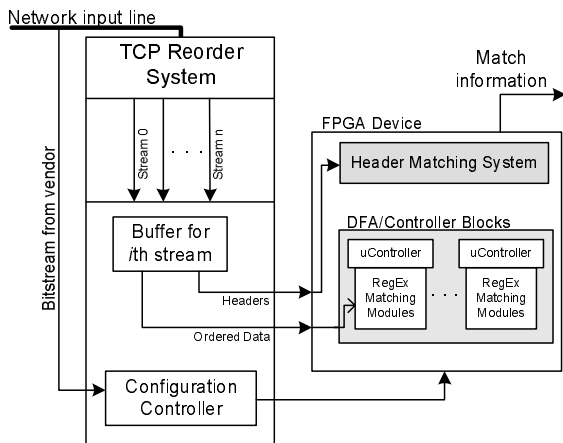


Fig. 1. IDS System Architecture

discussion of the large body of prior work in string matching for intrusion detection, and the basic principles of the Aho-Corasick and Bit-split algorithms. We then present our work on an embedded microcontroller that is integrated within the DFA state machine mechanism to reduce memory costs and provide regular expression functionality to the system. We will discuss what the architecture is, and is not, capable of processing, and the NFA to DFA conversion problems that continue to challenge the community. Finally, we will present some results from our experiments, showing the reduction of state memory usage from exponential growth to linear growth with the number of patterns.

2. RELATED WORK IN HARDWARE IDS

Snort [8] is a popular option for implementing intrusion detection in software. It is an open-source, free tool that promiscuously tap the network and observe all packets. After TCP stream reassembly, the packets are sorted according to various characteristics and, if necessary, are string-matched against rule patterns.

In our previous work in regular expression matching [9, 5], we presented a method for matching regular expressions using a Non-deterministic Finite Automaton, implemented on a FPGA.

The NFA concept is updated with predecoded inputs in [10]. The paper addresses the problem of poor frequency performance for a large number of patterns, a weakness of earlier work. By adding predecoded wide parallel inputs to a standard NFA implementations, excellent area and throughput performance is achieved.

These hardware approaches allow high speed and area efficiency, and can be easily extended to provide most regular expression functionality (as in an evolution of the original work in [5]). However, they will never provide the fast reconfiguration that is possible in a memory-based system,

short of an in-situ bitstream reconfiguration approach [11, 12].

In [13, 14], a novel hashing mechanism utilizing Bloom filter is discussed. Their implementation of a hashing-table lookup using a Bloom filter is an effective method to search thousands of strings for matches in a single pass. An interesting aspect of this work is that it can be extended somewhat using the software deceleration approach developed in this paper. The Bloom filter, instead of including just “present” bits at hash locations, can include an address for the microcontroller. By linking string literals together with software, some regular expressions can be supported. However, linking strings is still somewhat less flexible than a more general DFA-based processor.

3. SOFTWARE DECELERATION FOR REGULAR EXPRESSIONS

A regular expression [15](abbreviated as regexp from now on) is a string that matches a set of strings. Regular expressions are used for the concise notation of a string set. Instead of enumerating all possible variations of an attack, a regexp is designed using a variety of wildcarding and generalizing operators to match exactly what is required. The processing of a regexp is more difficult than simple string literals because each regexp has meta-information that conveys meaning.

For example, a regexp “(a|b)*cd” can be used to find matches for strings “cd”, “acd”, “bcd”, and “aabcd”, etc. The vertical bar ‘|’ is used for the alternation if we want to specify that either of left or right whole subexpressions occur. The parentheses ‘(’ and ‘)’ match whatever regexp is between them and indicates the start and end of a group. The Kleene closure operator ‘*’ means that the preceding “(a|b)” can appear zero or more times. Another form is “(ab){x,y}” meaning that “ab” must appear at least x times but less than y times.

3.1. Problem Definition

The problem addressed in this paper is in the efficient processing of regular expressions. In particular, we wish to reduce the memory cost of a set of Snort regular expressions when represented as state machines in hardware. Our contribution comes in two parts. We have developed an architecture that can reduce the cost of two particularly expensive regular expression operators (wildcards and constrained repetition). As well, we have developed software for converting classes of regular expressions into a form suitable for hardware IDS that integrates with the architecture.

First, we will discuss the reasons for, and the complexities of, the conversion process. Interpreting regular expressions at runtime is inefficient even in software. Thus some

intermediate form is required. Conveniently, a regular expression is closely related to the nondeterministic finite automata (NFA). However, simulating an NFA is also difficult. In this form many states can be active at once, and the transitions from each state must be computed individually before the next time step can occur. Because this reduces the system throughput, in general NFAs are converted into a *Deterministic* Finite Automata. In the DFA form, only one state is active at any time, thus there is only one next state to compute in each time step. This accelerates the processing, but complicates the pattern compilation process.

In the NFA to DFA conversion process, a single state is made to represent several active states in the NFA form. In order for the states to be distinguishable, a DFA state may be required for all possible NFA states. Clearly this can cause the total memory required to explode exponentially. This is particularly caused by wildcards, e.g. (.*) and especially by constrained repetition, e.g. (ab(x,y)). The focus of this paper is in eliminating the state explosion problems of those two types of regular expression components.

3.2. Key Ideas

We have developed a microcontroller-based architecture that links a DFA simulation module with a small, highly customized microcontroller. The microcontroller allows the memory-efficient implementation of the large class of regular expressions that include .* and ([\n]){x,y} without state explosion. Each regular expression is broken into independent segments at wildcard separators and reassembled post-match in the microcontroller. Each DFA is responsible for up to 28 pattern segments (the pieces of patterns separated by wildcards). The microcontroller keeps track of the flags and counters required to implement the wildcards and reassembles the pattern segments. This approach uses orders of magnitude less memory than implementing the extended regular expression functionality in the DFA.

Reducing the state explosion is only part of the problem when moving to hardware implementation. Like other implementations of state machines that require one transition in each cycle, a huge amount of storage is required. This problem comes from the large number of edges, maximum 256, pointing to the potential next states. In an earlier work([16]), we developed an efficient FPGA implementation of ideas from [17] that provides a much higher density of useful edges. The technique allows state machines to be represented using significantly less state memory that would be required in a naïve implementation by splitting each set of input bits across several smaller DFA blocks.

Via the use of this “bit-splitting,” a single state machine is split into multiple machines that handle some fraction of the input bits. The best approach seems to be 4 smaller machines, each handling 2 bits of the input byte. Thus, instead of requiring 2^8 memory locations for each of the possible

input combinations, only 2^2 locations are required per unit, for a total of $4*2^2=16$ locations over the four machines.

The “bit-split” architecture was introduced in [17] to implement the Aho-Corasick string matching algorithm with increased memory efficiency. The bit-split approach achieves this objective by reducing outgoing edges from each state by splitting one Aho-Corasick state machine into a set of several state machines. By converting only part of the regular expression to DFA, efficiency is greatly increased. Our approach replaces the removed parts (namely wildcards and constrained repetition) with an integrated microcontroller.

This approach allows the evaluation of a variety of rules without state explosion. Example rules from the database that are supported are as follows:

```
1 NLST\s [^\n] {100}
2 rcpt\s+to\:\s* [|\x3b]
3 Content-Type\s+audio\/(x-wav|x-midi)
4 File[0-9]+=http\x3a\x2f\x2f [^\n] {150}
```

4. ARCHITECTURE IMPLEMENTATION DETAILS

In this section we present the details of the microcontroller architecture and its associated software. The regular expression database is partitioned into sets and then processed into programs that work with a highly customized hardware architecture. A description of the overall methodology, DFA creation flow, and the microcontroller architecture follow.

4.1. Microcontroller Methodology

In general, the sort of rules that the microcontroller reduces the memory costs for compared to the basic DFA architecture are similar to the following:

```
PRIVMSG\s+nickserv\s+IDENTIFY\s [^\n] {100}
```

In this rule, the stream must begin with *PRIVMSG*, followed by at least one character of whitespace, followed by *nickserv*, again followed by at least one character of whitespace, followed by a single whitespace character, followed by exactly 100 bytes of characters that are not a new line character. This sort of pattern is quite common, and in a DFA, causes a state explosion. Each wildcard causes a replication of part of the tree, and the 100 character counter causes a 100 fold replication of the tree. This exponential behavior is the main weakness of the NFA→DFA conversion methodology. The microcontroller provides an efficient solution to this problem by providing an external method to record the wildcard states without using a large amount of memory. This approach was first detailed in [18] but was not implemented in detail. We contribute both a customized architecture as well as the integration with the more-efficient DFA simulation.

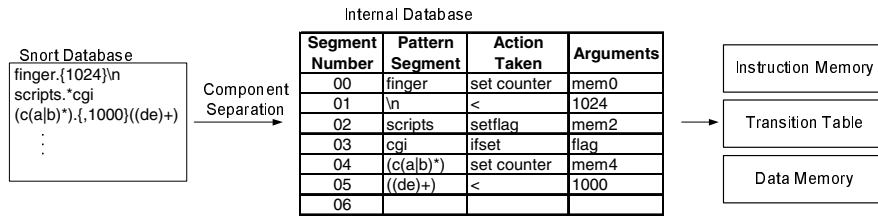


Fig. 2. Breaking patterns apart at wild-card boundaries provides smaller state tables after NFA→DFA conversion

A basic sketch of the approach is illustrated in Figure 2. The first step is to separate the rules into their components. This is done by parsing the regular expressions and extracting the substrings between wildcards and wildcard constraints. Each pattern segment and its matching wildcard function is saved to an internal table. This internal representation is translated by a template-based assembler to the binary instruction codes and starting data memory configurations. After the instruction code is created, the code is “linked” to generate the translation tables.

The code is executed within the constraints of the system architecture in Figure 3.

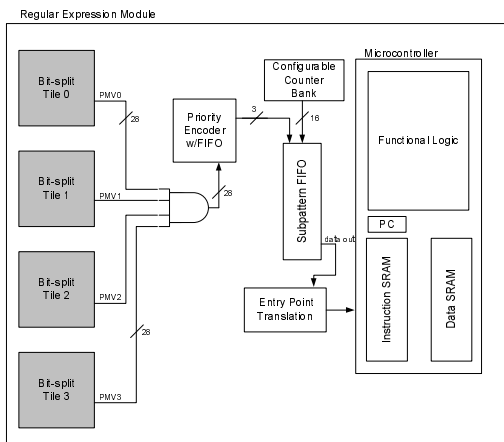


Fig. 3. System architecture of the regular expression microcontroller and bit-split state machines

4.2. Conversion from Regexp/NFA to Annotated DFA

The conversion from a set of regular expression segments to a DFA for IDS requires some additional features different from off-the-shelf software for regexp matching. This is because we are searching for a set of segments. Each segment can have some regular expression operators. These are all processed into one large DFA block. When this DFA is simulated, all of the patterns are matched against the input stream simultaneously. By collecting the patterns together in one module, higher memory and computational efficiency is achieved compared to processing them individually.

4.2.1. Finding Overlapped Matches

Because the patterns are combined together, there is a possibility that one pattern can overlap with another in such a way that a match might be overlooked. For instance, suppose there are two attacks “telephone” and “phonebook”. These two can be combined into one regexp “(telephone)|(phonebook)”. If “telephonebook” is the input string, we must be able to detect both “telephone” and “phonebook” in the input stream. This is important for IDS. An attacker can execute both attacks using only 4+5+4 characters. Thus, we need to know that both “telephone” and “phonebook” were detected. Therefore, we should modify the DFA to handle this problem. This is done by simply adding an edge from ‘e’ at the end of “telephone” to ‘b’ of “phonebook” in the DFA. By adding an edge, we actually merge two DFAs into one. However, this method cannot be applied to general regexp. For example, let us consider the regexp ((ad?b)+bcb)d(bb)?). If the input string is “dbbcb”, we should find three overlap matches “d”, “dbb”, and “adbcb”. This task cannot be done by merging two DFAs with only some new edges as in the case of (telephone)|(phonebook). There should be more states which are not included in the original two DFAs, but this process is largely outside the scope of this paper. Finding such a generalized algorithm for these additional states is a future work.

4.2.2. Annotating the Patterns

Another feature required to report attacks is the ability to distinguish one pattern from another by giving them unique numbers. Let us get back to the previous example. If we merge “telephone” and “phonebook” by using ‘|’, off-the-shelf regexp matching software [15] will regard these two patterns as one because there is only one DFA. However, for the IDS, we should assign “telephone” to Match#1 and “phonebook” to Match#2 to know what attack is performed.

For this example, let us use the third rule in our example database from Figure 2, the regexp “(c(a|b)*).{,1000}((de)+)”. This regexp provides a more interesting example than the other rules as finger, scripts, and cgi just produce a graph similar to an Aho-Corasick tree. The rule breaks into two segments: (c(a|b)*) and ((de)+). The functionality of the constrained distance operator “. {,1000}” will be replaced in

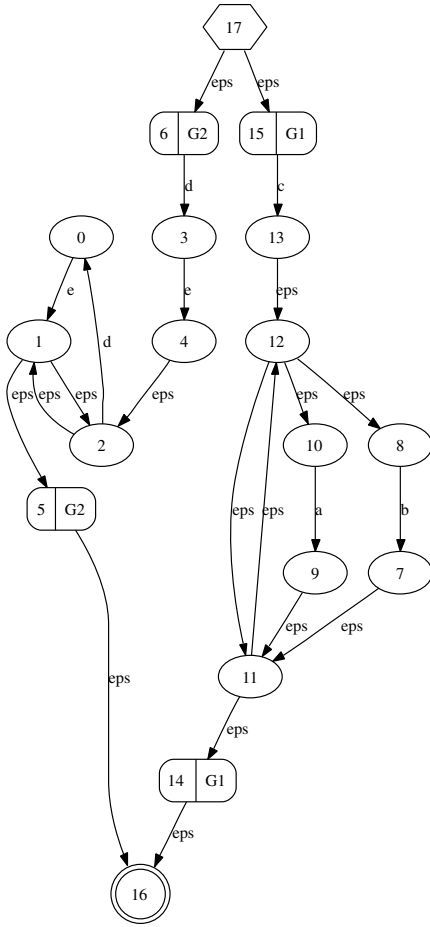


Fig. 4. The NFA of “(c(a|b)*)((de)+)”

software by the microcontroller at runtime.

The NFA constructed from the two segments (c(a|b)* and ((de)+) is shown in Figure 4. In the figure, the nodes are each annotated with a group number (G1, G2...). Every node annotated with same group number, such as G1 and G2, are in the same pattern. That is, if we regard “(c(a|b)*)” as Match#1 and “((de)+)” as Match#2, pattern1 includes the state 15, 13, 12, 10, 8, 9, 7, 11 and 14. Match#2 includes the state 6, 3, 4, 2, 0, 1 and 5. We can easily find each group while we are constructing NFA from regexp by the property of *Thompson’s construction* [3].

Putting this pattern number into the accepting state of DFA is straightforward. While we are doing the *subset construction* [3] from Figure 4 to make DFA, all the transitions from state 15 to 14 has group number G1. Thus, the accepting states of the DFA made by these paths correspond to Match#1. In the same way, the accepting states of DFA made by the paths from state 6 to 5 correspond to Match#2. Figure 5 shows all the information needed. The rounded rectangle indicates an accepting state and the hexadecimal number is for the pattern match number. If there is an input

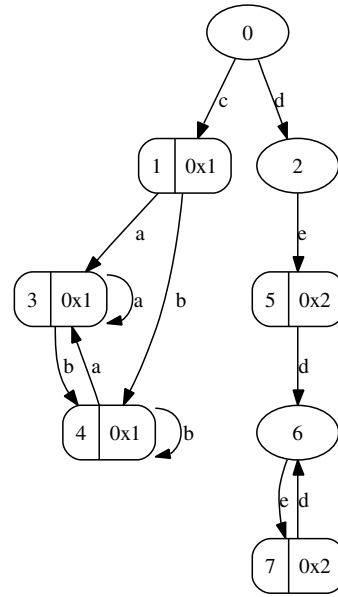


Fig. 5. The DFA of “(c(a|b)*)((de)+)” with pattern numbers

string “cadede”, our DFA can find each of the possible six matchings. The sequence of state transition and the matched pattern number is as follows.

1 (0x1) ->3 (0x1) ->2->5 (0x2) ->6->7 (0x2)

Bitwise-ORing is used in one accepting state to indicate the matched patterns because one state can have many matched patterns in DFA. Although we do not have such a case in this example, if one accepting state is matched with both Match#1 and Match#2, it should have a pattern number (0x1)|(0x2) = (0x3).

Conversion from modified DFA to bit-split proceeds as detailed in our earlier paper [16] and original UCSB paper [17].

4.3. Microcontroller Software

Table 1 lists the available opcodes implemented by the microcontroller. The instructions provide the necessary capabilities to support all required functions for the regular expression matching that we target. The only expressions that can not be handled by the system are restricted because of the NFA→DFA conversion process, and not the microcontroller.

A custom microcontroller was implemented for this application because off-the-shelf microcontrollers have many disadvantages. One, they require more logic area for the operations achieved. While the PicoBlaze [19] requires only 78 slices, it is an 8 bit controller and thus cannot provide the wider data operations needed for controlling the counters needed in the IDS application. The MicroBlaze [20] supports the appropriate widths, but is far larger (up to 1000

slices), compared to the 114 slices required for the custom controller. Any generic controller provides abilities that are not required. This increases the area costs, and decreases the total number of rules that can be supported.

Each type of wildcard block is built from a set of predefined opcode templates. These templates can be stacked on each other to extend functionality to larger and more complex regular expressions.

Unbounded wildcards are simple to implement in software. After the wildcard is reached, it remains active until the end of the stream. In the example patterns implemented in Figures 2 and 6, the unconstrained wildcard .* is reached after matching the initial string *scripts*. The translation table provides an entry point at program address 10, which sets a flag in the data memory at location 2. If the second pattern segment *flag* is found, instruction 12 is the beginning of the execution segment. The controller loads memory location 2 again and checks to see if it is set. If it is set, then the output register is set and the *match* signal is raised, signaling the match for the entire regular expression to the host. Execution then jumps unconditionally back to a wait loop at address zero.

In the case of the constrained wildcard expression *finger*.{1024} \n, the detection of the pattern segment *finger* causes the counter to be saved to memory location 0. The detection of the newline character \n reloads the old counter, subtracts the old counter from current counter, then loads and subtracts the bound (1024). If the bound is satisfied, the output flags are set and the regular expression match is successful.

In both of these situations, the component parts are easily expanded to support larger and more complex regular expressions. Both minimum and maximum bounds can be constructed from combinations of the subtraction instructions and conditional jumps.

Additionally, pattern segments that are attached to multiple patterns can be handled in the same fashion. Because \n is such a common byte code in IDS pattern matching, its detection will require the inspection of many counters. In this situation, the microcontroller loads and compares the bounds of every potentially relevant pattern. The cost of handling a large number of newline handlers can be reduced by limiting the maximum number of newline-incorporating rules in a given module. Because there are a large number of modules on the FPGA (~50), this partitioning is a reasonable approach to spread out highly common sequences.

4.4. Microcontroller Hardware

We will briefly discuss some of the important elements of the microcontroller. The system includes 16 FIFO slots, 32 16-bit data words for storing counters and flags, and 128 instruction words. The architecture includes a set of four bit-split state machine tiles forming a module as originally

	0	000	0	jump to here - dead loop
finger	1	010	0	set counter in mem0
	2	000	0	jump to 0
	3	100	0	load old counter to r1 from mem0
	4	101		subtract current counter - r1
\n	5	100	1	load bound from mem1 to r1
	6	100		subtract (result - bound)
	7	000	0	jump to start if positive
scripts	8	011	0	set output for pattern 0
	9	000	0	jump to start
	10	001	2	set wildcard flag in mem2
	11	000	0	jump to start
	12	111	2	load flag from mem2
cgi	13	000	15	jump over next if wildcard is active
	14	011	1	set output for pattern 1
	15	000	0	jump to start
	:			(other patterns continue using the same templates)

Fig. 6. Program listing for detecting the patterns listed in Figure 2

discussed in [17] and adapted for FPGA by our team in [16]. The output of the bit-split module (consisting of an encoded index for each pattern segment) is connected to the input of the FIFO event handler.

Figure 2 illustrates the table for conversion of segment match addresses into entry points into the instruction memory.

Pattern Segment #	Instruction Pointer
0	1
1	3
2	10
3	12
4	16
5	18

Table 2. The transition table converts the encoded pattern segment identifier into an entry point for the microcontroller

The IDS matching problem can yield multiple matches simultaneously or in quick succession – in particular given the increased number of individual pattern segments after component separation that can share characters.

In order to handle simultaneous matches, a specialized priority encoder was developed that provides the encoded address of all of the matches in successive cycles. Because of this potential delay with reporting results, and because the microcode can require multiple cycles to execute for each string encountered, a FIFO buffer is provided to handle the delay between when the string is detected and it is processed by the microcontroller. Of course, because the microcontroller is largely handling the timings between segment detections, it is important that the state of the system when the detection occurred to be preserved as well. This is actually handled by recording the relevant counter index for each pattern segment in the FIFO as the segment match index comes in from the DFA. The segment stores the appro-

	Instruction	Input	RAM Output	Shadow	Neg Flag
000	jump if > 0	inst addr, flags			0
001	set flag	mem addr, value	value		
010	set counter	mem	bytes_passed	ram_out	
011	set output reg	value			
100	sub	regs		(shadow-ram_out)	if < 0
101	sub	regs		(time-ram_out)	if < 0
110	nop				
111	load	mem	ram_out	time_counter	if(ram_out<1)

Table 1. Opcodes for the regular expressions microcontroller

appropriate counter’s value at the time of the detection in memory until it can be processed. Several counters are provided for 1) any byte, 2) non-\n bytes, 3) white space (\s), and 4) a programmable match counter. The microcontroller reads the current counter value and stores it in memory as in Figure 6 – the counters are not linked to any particular pattern. The specialized counters exist to provide more capability.

5. RESULTS

The synthesis tool for the VHDL designs within Carte version 2.1 is Synplify Pro version 8.1, and ISE tools version 7.1.2 is Synplicity Synplify Pro 8.1 and the place-and-route tool is Xilinx ISE 7.1.2. The target device is the Virtex4 fx100 with speed grade -12. The FX series device provides a much better RAM/logic ratio compared to the other devices in the Virtex IV series. Because the architecture is constrained only by the amount of block RAM and not the logic, it is desirable to find the device with the largest amount of block RAM. The results are based on the placed-and-routed design. The Virtex 4 fx100 device supports 376 block RAMs, and allowing 47 rule modules. The 47 rule modules each have a regular expressions microcontroller. Each module requires 456 slices total. The controller itself occupies 218 of those slices. Of the 218, most of the slices are used for distributed RAM, as there is a large amount of memory dedicated to the instruction, data, translation, and event buffer memories:

Memory	Width x Depth	Total Bits
FIFO	21x16	336
Translation	7x32	224
IMem	16x128	2048
Dmem	16x32	512
total: 3120 bits		
98 slices used for memory		

With 47 rule modules at 28 patterns per module, a single device can handle 1,316 patterns. At 412 MHz and one byte per cycle, the system has a throughput of 3.3 Gbps. The comparisons against other implementations are shown in Table 3. We only consider architectures providing on-the-fly memory-based pattern recognition capabilities, excluding hardwired reconfiguration-based architectures. Note that

the USC RegExp Controller entry (this work) is the only architecture that provides both on-the-fly pattern reconfiguration and support for a larger class of regular expressions. The overall throughput is only slightly less than the UCLA design. However, the RAM usage is significantly higher, but, again, the meta-information conveyed in our design is far greater than a simple string matcher. The block RAM usage of the string-centric “USC Bitsplit” row is quoted as identical to the “RegExp Controller” because it uses the same RAM modules. However, the earlier work did not utilize almost half 1/3 of the available memory. The RegExp controller makes use of it for providing the encoded entry points.

The system can support many types of regular expressions. However, the microcontroller solution is not optimal for many types of Snort regular expressions. For instance, expressions containing very short pattern segments will cause the microcontroller to a higher duty cycle than is sustainable. In the current design, there are only 16 slots in the FIFO event buffer. Regardless of the size of the buffer, it can always be overflowed by a sufficiently determined attacker with some knowledge of the system design if particularly short pattern segments are repeatedly transmitted. More precisely, the program sequence for an unconstrained wildcard (for instance, *scripts.*cgi*) requires 5 cycles every time the *cgi* pattern segment is detected (4 program cycles and then one idle cycle before the next pattern event can be processed). As it requires only 3 cycles for “*cgi*” to enter the system from the network stream, it would only require 40 repeated “*cgi*” sequences before the buffer is overrun, given 16 FIFO slots. Thus, the minimum *safe* segment size is 5 bytes.

6. CONCLUSION

With the growing importance of Intrusion Detection Systems, the flexibility, speed, and memory-efficiency of the regular expression processor is essential.

In this paper we show that bit-split architecture modules combined with a small microcontroller are flexible enough that regular expressions can be evaluated while maintaining competitive data rates in an FPGA implementation. The key contribution of this work is an architecture that can provide regular expression matching at memory costs similar to

Design	Supports	Device	BW	bRAM Mem	Logic Cells/char	Characters Total
USC RegExp Controller	RegExp	Virtex 4 fx100	1.4 Gbps	46 bytes/char	2.56	16715
USC Bitsplit [16]	strings	Virtex 4 fx100	1.6 Gbps	46 bytes/char	1.4	16715
USC KMP Arch [21]	strings	Virtex II Pro	1.8 Gbps	4 bytes/char	3.2	3200
UCLA ROM Filter [22]	strings	Virtex 4	2.2 Gbps	5.72 bytes/char	0.209	32168

Table 3. Results comparisons against various other memory-based on-the-fly reconfigurable implementations in reconfigurable hardware

string matching (compared to a direct DFA conversion). As well, the architecture provides on-the-fly pattern reprogramming without FPGA reconfiguration.

7. REFERENCES

- [1] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the Slammer Worm," *IEEE Security & Privacy Magazine*, vol. 1, no. 4, July-Aug 2003.
- [2] D. Moore, C. Shannon, G. Voelker, and S. Savage, "Internet Quarantine: Requirements for Containing Self-propagating Code," *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2003)*, April 2003.
- [3] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers - Principles, Techniques, and Tools*. Reading, MA, USA: Addison-Wesley, 1986.
- [4] Y. Cho and W. H. Mangione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," in *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, 2004.
- [5] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in *Proceedings of the Ninth Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, 2001.
- [6] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *Proceedings of the Tenth Annual Field-Programmable Custom Computing Machines (FCCM '02)*, 2002.
- [7] P. James-Roxby, G. Brebner, and D. Bemmann, "Time-Critical Software Deceleration in an FCCM," in *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, 2004.
- [8] Sourcefire, "Snort: The Open Source Network Intrusion Detection System," 2003, <http://www.snort.org>.
- [9] R. Sidhu, A. Mei, and V. K. Prasanna, "String Matching on Multicontext FPGAs using Self-Reconfiguration," in *Proceedings of the Seventh Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '99)*, 1999.
- [10] C. R. Clark and D. E. Schimmel, "Scalable Parallel Pattern Matching on High Speed Networks," in *Proceedings of the Twelfth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2004 (FCCM '04)*, 2004.
- [11] B. Blodget, S. McMillan, and P. Lysaght, "A Lightweight Approach for Embedded Reconfiguration of FPGAs," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, 2003.
- [12] A. Poetter, J. Hunter, C. Patterson, P. Athanas, B. Nelson, and N. Steiner, "JHDLBits: The Merging of Two Worlds," in *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL 2004)*, 2004.
- [13] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Implementation of a Deep Packet Inspection Circuit using Parallel Bloom Filters in Reconfigurable Hardware," in *Proceedings of the Eleventh Annual IEEE Symposium on High Performance Interconnects (HOTI03)*, 2003.
- [14] S. Dharmapurikar and J. Lockwood, "Fast and Scalable Pattern Matching for Content Filtering," in *Proceedings of Symposium on Architectures for Networking and Communications Systems (ANCS 05)*, 2005.
- [15] "PCRE - Perl Compatible Regular Expressions," 2006, <http://www.pcre.org/>.
- [16] H. J. Jung, Z. K. Baker, and V. K. Prasanna, "Performance of FPGA Implementation of Bit-split Architecture for Intrusion Detection Systems," in *Proceedings of the Reconfigurable Architectures Workshop at IPDPS (RAW '06)*, 2006.
- [17] L. Tan and T. Sherwood, "A High Throughput String Matching Architecture for Intrusion Detection and Prevention," in *Proceedings of the 32nd Annual Intl. Symposium on Computer Architecture (ISCA 2005)*, 2005.
- [18] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz, "Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-76, May 22 2006. [Online]. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-76.html>
- [19] "PicoBlaze User Resources," http://www.xilinx.com/ipcenter/processor_central/picoblaze/picoblaze_user_resources.htm.
- [20] "MicroBlaze Performance," http://www.xilinx.com/ipcenter/processor_central/micoblaze/performance.htm.
- [21] Z. K. Baker and V. K. Prasanna, "Time and Area Efficient Pattern Matching on FPGAs," in *The Twelfth Annual ACM International Symposium on Field-Programmable Gate Arrays (FPGA '04)*, 2004.
- [22] Y. Cho and W. H. Mangione-Smith, "Fast Reconfiguring Deep Packet Filter for 1+ Gigabit Network," in *Proceedings of the Thirteenth Annual IEEE Symposium on Field Programmable Custom Computing Machines 2005 (FCCM '05)*, 2004.