Efficient Regular Expression Evaluation: Theory to Practice

Michela Becchi
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-4306
mbecchi@cse.wustl.edu

Patrick Crowley
Washington University
Computer Science and Engineering
St. Louis, MO 63130-4899
+1-314-935-9186
pcrowley@wustl.edu

ABSTRACT
Several algorithms and techniques have been proposed recently to accelerate regular expression matching and enable deep packet inspection at line rate. This work aims to provide a comprehensive practical evaluation of existing techniques, extending them and analyzing their compatibility. The study focuses on two hardware architectures: memory-based ASICs and FPGAs.

1. INTRODUCTION
Efficient regular expression evaluation is a critical mechanism in modern network security. While deep packet inspection will never be a comprehensive security solution, it is the standard technique for detecting malicious patterns in network traffic. As a result of its importance, researchers have been active in proposing improved algorithms, data structures, and architectures for high-speed implementations. Unfortunately, much of the work has developed in a disjointed fashion, resulting in a disconnection between the high-level goals and characteristics of the research proposals and the practical realities of rule-sets and implementation technologies. To reduce these ideas to practice in real systems, it is necessary to understand how the basic techniques for building efficient regular expressions relate to one another, and what implications they have on a given implementation technology. In this paper, we describe the basic techniques required for efficient regular expression evaluation and explore their joint application on the two primary implementation technologies—namely, field-programmable gate arrays (FPGAs) and memory-based ASIC solutions.

Architectures for high-speed regular expression evaluation are based on either deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs) [2]. While DFAs have been more popular, the merits of NFA-based approaches are increasingly appreciated [11][14]. Beyond the choice of automata, there are three basic algorithmic techniques used to create feasible automata for high-speed regular expression evaluation: *(i) edge-compression*, *(ii) alphabet-reduction*, and *(iii) increased stride*.

Edge-compression reduces the size of a state-minimized DFA by exploiting the redundancy present in the transitions between states. Such is the case with D^2FA [6][7], where default paths are constructed to eliminate redundant edges. Alphabet-reduction is a basic technique for mapping the set of symbols found in an alphabet to a smaller set by grouping characters that label the same transitions everywhere in the automaton [7][8]. A reduced alphabet size can dramatically diminish the amount of storage needed to represent transitions within an automaton. Multi-stride DFAs were proposed in [8] as a way to increase processing throughout. Specifically, a stride-*k* DFA consumes *k* characters per state transition rather than just one, thus yielding a *k*-fold performance increase.

Each of these algorithmic techniques has associated costs and benefits. When applied together, it is not straightforward how these costs and benefits interact. In some cases, ad-hoc algorithms are needed to make the techniques practical on reasonable hardware. Moreover, most of these techniques—such as alphabet-reduction and increased stride—have been evaluated only in the context of DFAs.

The contributions of this paper can be summarized as follows: We describe the first *practical* algorithm for creating state-minimized, edge-compressed, reduced-alphabet, multi-stride DFAs. To the best of our knowledge, we are the first to describe concrete algorithms for creating efficient, reduced-alphabet, multi-stride NFAs. Additionally, we describe and evaluate how these efficient automata can be reduced to practice on FPGAs and memory-based ASICs. As a side contribution, we propose a logic-minimization scheme, which can be used to both efficiently perform alphabet-translation on FPGAs and efficiently handle DFA states with poor compression properties in ASICs.

The remainder of this paper is organized as follows. In Section 2 we provide additional background. In Section 3 we propose an algorithm to build compact NFAs, with support for NFA-based alphabet-translation and stride doubling. In Section 4 we show how to practically combine the different techniques listed above on medium and large DFAs. The characteristics of the resulting automata are analyzed in Section 5. In Section 6 we carry out an evaluation on FPGAs and memory-based ASIC architectures. We conclude the discussion in Section 7.

2. BACKGROUND
The prior work in the area of regular expression matching at line rate can be categorized by distinct implementation targets: FPGA-based designs [14][15][16][17][18] [19][20] and schemes suitable for deployment on general-purpose processors or ASIC hardware [6][7][8][9][10][11][12][13].

ANCS'08, November 6–7, 2008, San Jose, CA, USA.

The two main advantages of FPGAs reside in their intrinsic reconfiguration capability and parallelism. An interesting NFA implementation for FPGAs is described by Sidhu and Prasanna [14]. The main idea is to encode each state in a flip-flop to allow each character to be processed in constant time. In this work, we adopt Sidhu and Prasanna's scheme. A problem with FPGA implementations is due to the limitation in the number of regular expressions deployable on a single chip. We aim to maximize this number by minimizing the implemented NFA and using other logic-optimization techniques. We achieve densities that substantially exceed those of recent proposals [20].

The main advantages of algorithmic approaches suitable for implementation on memory-centric systems are generality, versatility, and availability of higher clock frequencies. In this context, memory storage and bandwidth requirements are the main issues.

A substantial body of research work has focused on compression techniques aimed at reducing the amount of memory needed to represent DFAs. In particular, Kumar et al. [6] proposed an algorithm to compress a DFA through the introduction of default transitions, a generalization of the failure pointer concept presented in the classical Aho-Corasick algorithm for string matching [1]. The basic idea is to trade memory storage requirement for processing time. A more general and less complex algorithm to achieve the same goal was recently proposed by Becchi et al. [7]. By restricting default transitions to be backward directed, they can achieve a better worst-case bound on the processing time, while offering the same compression degree of [6]. For reasons we will clarify later, we will use the scheme proposed in [7].

A considerable amount of literature has addressed the problem of state explosions occurring when compiling complex regular expressions into a single DFA [10][11][12][13]. This problem is orthogonal to the techniques analyzed in this paper. However, our proposal in the context of DFAs can be applied to multiple-DFA proposed by Yu et al. in [10] and to the hybrid-FA proposed by Becchi and Crowley in [11]. In fact, both schemes have raw DFAs as building blocks.

3. TECHNIQUES FOR NFAs

In this section, we present different techniques to efficiently represent non-deterministic finite automata.

3.1 Building *Good* NFAs

Given a regular expression or a set of regular expressions, it is possible to construct different accepting NFAs, which are functionally equivalent but have different sizes and exhibit different traversal properties. Unlike the case with DFAs, there exists no defined or commonly accepted notion of a *minimal* NFA. However, we can define an intuitive preferential ordering on NFAs. Given two functionally equivalent NFAs—say NFA_1 and NFA_2—NFA_1 is preferable to NFA_2 if *(i)* it is smaller (both in number of states and number of transitions), and *(ii)* processing the same input text triggers fewer state traversals in NFA_1 than in NFA_2. The first property holds independent of the implementation, whereas the second is crucial on memory-centric systems but is not significant on FPGAs.

In this section we provide general mechanisms for building convenient NFAs. In particular, the goal is two-fold: minimizing the number of states and minimizing the number of transitions.

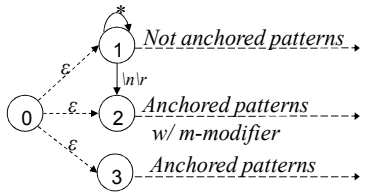

Figure 1: NFA starting point, to avoid state replication for anchored patterns with *m-modifier*.

Achieving these two goals automatically takes care of the NFA traversal properties. Note that none of these goals would be achieved by simply applying either Thompson's [4] or McNaughton-Yamada's [5] NFA construction algorithm.

Given a set of regular expressions, our starting point is the corresponding NFA with epsilon transitions, which can be easily built following standard procedures [2]. We notice that practical data-sets contain three kinds of patterns: *(i)* unanchored patterns (which may appear at any position of the input text), *(ii)* anchored patterns (which must appear only at the beginning of the text), and *(iii)* anchored patterns with an *m-modifier (*which can occur either at the beginning of the text or after any new-line). If not properly represented, the latter may cause state replication in the NFA. To avoid this effect, we start building the NFA from the skeleton represented in Figure 1.

We make the following observations: First, epsilon transitions should be removed in that they increase both the number of states and average active-set size during traversal (if a state *s* is active, so are all states connected to *s* through an epsilon transition). Second, the number of outgoing transitions on the same character from any given state must be kept as low as possible. Both goals can be achieved by a reduction operation similar to subset construction (i.e., the NFA-to-DFA transformation). The core of the reduction operation is represented in the pseudo-code on the next page, where an NFA is represented through its transition function δ associating to each pair *(state, character)* a set of target states.

Like subset construction, NFA-reduction performs a breadth-first traversal of the NFA. For each state and for each character, the union of the epsilon closures of the set of target states is computed (line 10). Each of these sets of states is called a *subset*. The procedure keeps a mapping (stored in the *subsets* variable) between the subsets and the final NFA states.

The main differences between subset construction and NFA-reduction are the following: First, in the DFA case, each state must present one outgoing transition for each character of the alphabet. Conversely, in the NFA, case it is sufficient to represent only those transitions corresponding to progress in the matching operation; when an input character does not match any transition from an active state, then that activation ends. Second, when converting an NFA to DFA, processing self-transitions has two effects: it increases the number of DFA states and causes the NFA state with self-transitions to appear in different subsets. This, in turn, makes the number of overall states and transitions increase. These effects are particularly significant when the self-transition is labeled with wildcards or large character ranges. In fact, if an NFA state with outgoing transitions on *m* different characters appears in *n* different subsets, the corresponding *n* states in the transformed automaton will have at least *m* outgoing transitions each.

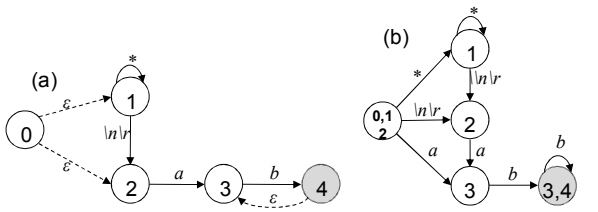

Figure 2: Effect of NFA-reduction on NFA representing ^ab^+ with *m-modifier*: (a) original NFA and (b) reduced NFA.

NFA-reduction isolates self-transitions from transitions to different target states (lines 11-12) and processes them separately (lines 13-17, 18-22). This prevents NFA states with many self-transitions from being duplicated in many subsets and thus limits the overall number of states and transitions.

```
procedure reduce_NFA (modifies NFA nfa=(δ(states, ∑∪{ε}));
(1)   set target, self_target, other_target;
(2)   map subsets : states→ set
(3)   list queue;
(4)   state ns=0 ;
(5)   subsets(0)= ε-closure(0);
(6)   queue.push(0);
(7)   while (!queue.empty()) do
(8)     state s= queue.pop();
(9)     for char c ∈ ∑ do
(10)      target = ∪ { ε-closure(t): t∈δ(s',c) & s'∈ subsets(s)};
(11)      self_target = target ∩ subset(s);
(12)      other_target= target \ self_target;
(13)      if (!self_target.empty()) then
(14)        if (!self_target ∈ subsets) then
(15)          subsets(++ns)=self_target;
(16)          queue.push(ns);
(17)        δ(s,c)= δ(s,c) ∪ subset -1(self_target);
(18)      if (!other_target.empty()) then
(19)        if (!other_target ∈ subsets) then
(20)          subsets(++ns)=other_target;
(21)          queue.push(ns);
(22)        δ(s,c)= δ(s,c) ∪ subset -1(other_target);
end;
```

Accepting states, omitted in the pseudo-code for the sake of readability, are dealt with as they are in the NFA-to-DFA transformation. Specifically, the final NFA states are associated with the set of rules accepted by the original NFA states in the corresponding subset.

Figure 2 and Figure 3 show examples of the application of the described reduction procedure on NFAs accepting anchored and non-anchored regular expressions. As can be observed in Figure 3, the procedure causes common prefixes to collapse. The more different regular expressions present common prefixes, the largest the overall reduction. In both examples, state 1 is a self-target and is processed separately.

Finally, notice that the algorithm, although designed also to remove epsilon transitions, can be effectively applied to an NFA without epsilon transitions for state- and transition-reduction purposes.

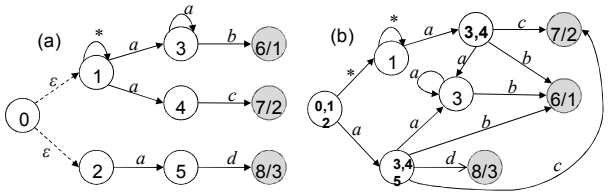

Figure 3: Effect of NFA-reduction NFA representing (1) a^+b, (2) *ac*, and (3) ^*ad*: (a) original NFA and (b) reduced NFA.

3.2 Generating Multi-Stride NFAs

As mentioned above, an NFA with *stride k,* or *k*-NFA, is meant to process *k* input characters at a time. Given an NFA defined on alphabet Σ, the corresponding *k*-NFA is defined on alphabet Σ^k. We present an algorithm to double the stride of an NFA. The proposed algorithm can be repetitively invoked to generate *k*-NFAs with *k* being a power of two, and can be easily generalized for any *k*.

Intuitively, the stride of an NFA can be doubled by tracing, from each state, all possible two-character combinations and the corresponding target states. Combinations not leading to a target state have no role in the matching operation, and, hence, do not produce state transitions. The proposed algorithm is represented in the pseudo-code below. To understand this algorithm, we must discuss how multiple targets and accepting states are handled.

```
procedure double_stride (NFA nfa=(δ(states, ∑),rules(states))
          modifies NFA nfa'= (δ'(states', ∑²), rules'(states')));
(1)   map accept: rule→ state;
(2)   set target1, target2, processed, rules1, rules2;
(3)   list queue;
(4)   queue.push(0); processed={0};
(5)   while (!queue.empty()) do
(6)     state s= queue.pop();
(7)     for char c1 ∈ ∑ do
(8)       target1 = {t: t ∈ δ(s,c1) }; target2 = Ø
(9)       if (!target1.empty()) then
(10)        for char c2 ∈ ∑ do
(11)          target2 = ∪ {t: t ∈ δ(s',c2) & s'∈ target1 };
(12)          rules1 = {r: r ∈ rules(s1) & s1 ∈ target1};
(13)          rules2 = {r: r ∈ rules(s2) & s2 ∈ target2};
(14)          target2 = target2 ∪ {t:accept(r)=t & r∈rules1-rules2};
(15)          for state t∈ target2 do
(16)            δ'(s,c1c2)= δ'(s,c1c2) ∪ {t};
(17)            if (!t∈ processed) then
(18)              queue.push(t); processed= processed ∪ {t};
end;
```

First, when a state *s* transitions to multiple targets $t_1,\ldots,t_h$ on the same input pair c_xc_y, *h* different transitions (s,t_i) are created (lines 15-16).

This leads to an increased automaton size. The NFA-reduction algorithm presented above can be used to reduce the obtained *k*-NFA; alternatively, the two algorithms can be merged to concurrently double the stride and reduce the NFA.

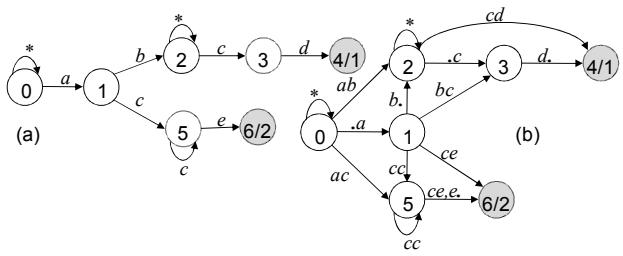

Figure 4: Effect of stride doubling: (a) NFA accepting (1) *ab.*cd* and (2) *ac*$^{+}$*e*; (b) corresponding *2*-NFA

Second, to be certain that increasing the stride does not alter the set of accepted strings, the *k*-NFA should monitor matches happening on the first character of any input pair (line 12). To this end, the algorithm keeps a mapping (line 1) between each rule, and a (possibly extra) accepting state without any outgoing transitions. Lines 12-14 ensure that functional equivalence with the original NFA is preserved.

Figure 4 shows an example of stride doubling. A few points are worth observing. First, the number of states in the original and in the stride-two NFA is the same (in general, they are comparable—some extra accepting states may be created). Second, nodes connected to dot-star states in the original NFA have incoming transitions with a *do-not-care* on the first character in the stride-two counterpart. Third, accepting states have incoming transitions with a *do-not-care* on the second input character. The significance of the last two observations will be clear in Section 3.3.

3.3 Reducing the NFA Alphabet Size

Alphabet-reduction, described in [7] in the DFA case, is based on the observation that in most DFAs, only a subset of characters in the alphabet define distinct transitions in the automaton. If, for any state s, $\delta(s,c_x)= \delta(s,c_y)$, then characters c_x and c_y can be associated with the same symbol in a reduced alphabet Σ'. The resulting automaton based on Σ', *DFA'*, will have a reduced number of transitions. This comes at the cost of having to translate all input characters to the reduced alphabet (from Σ to Σ') prior to matching.

The same concept applies also to NFAs, to which the alphabet-reduction algorithm presented in [7] can be adapted. Specifically, per-state character classes are still computed and compared to determine global character classes. When processing a given state, the target sets on each input character are in this case equated.

As can be verified, alphabet-reduction will produce Σ'*={a,b,c,d,e,all_remaining_chars}* for the NFA in Figure 4 (a), and Σ'_2*={ab,ac,ba,bc,cc,cd,ce,da,dc,ea,ec,b[^ac], d[^ac],e[^ac], [^bde]a,[^bde]c, all_remaining_char_pairs}* for the *2*-NFA in Figure 4 (b). Assuming that Σ is the ASCII alphabet (256 elements), Σ_2 would have 256^2, that is 65536 elements. Therefore, alphabet-reduction not only can be beneficial for basic NFAs, but also becomes fundamental when increasing stride. When applying alphabet-reduction after stride doubling, the resulting alphabet is not only much smaller than Σ^2, but also smaller than the square of the size of the reduced stride-one alphabet.

4. TECHNIQUES FOR DFAs

In the context of DFAs there exists a well-established state-minimization procedure [3], and several compression techniques have been devised to reduce the DFA memory storage requirement [6][7][9][8]. Moreover, stride doubling has been proposed in [8] in the context of 1) small DFAs consisting of 40-100 states and 2) a compressed alphabet of limited size. The proposal targets ASIC architectures where it is conceivable to have multiple DFA engines running in parallel. Unfortunately, the combination of techniques proposed in [8] is not feasible in contexts where bigger DFAs and/or larger compressed alphabets are involved.

In this work, we aim to study the feasibility of multi-stride DFAs in contexts where the underlying patterns are more complex and numerous.

4.1 Addressing Multi-Stride DFAs

The basic problem in building a DFA of stride *k,* or *k*-DFA, lies in the memory requirement. In fact, given a DFA defined on an alphabet Σ, the non-compressed *k*-DFA will have $|\Sigma|^k$ outgoing transitions per state. Let us assume, for example, to have a 1K-state DFA defined on the ASCII alphabet. The full *2*-DFA version will consist of about 65 million transitions, whereas the full *4*-DFA will present about 4.3 trillion transitions. When compiling several regular expressions, DFAs from practical data-sets easily reach 100K states.

To make a *k*-DFA feasible, we take advantage of two techniques presented in [7]: (i) *alphabet-reduction* and (ii) *default transition compression*. The use of alphabet-reduction (also exploited in [8]) can be motivated as in the NFA case: when the stride increases, the automaton in effect uses only a small subset of the entire alphabet. The benefit of the second technique can be explained as follows: Default transition compression acts by removing transition redundancy present in a DFA. In other words, it takes advantage of the fact that different states transition to the same target on the same input character. If the stride doubles, then the number of transitions in the DFA increases quadratically, but the number of states does not. Intuitively, the fraction of distinct transitions decreases. Therefore, transition redundancy tends to increase with the stride.

These two mechanisms are used not only to compress the *k*-DFA, but also to make building a *k*-DFA feasible with reasonable hardware resources. In particular, we find it necessary to concurrently implement stride doubling and default transition compression to avoid troublesome memory size requirements

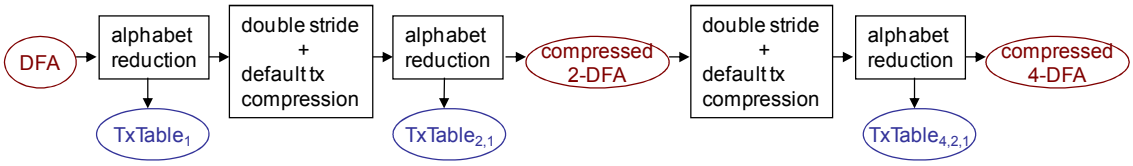

Figure 5: Steps involved in building a *k*-DFA. Each alphabet-translation table (*TxTable*) depends on the previous one.

when constructing the automaton. Moreover, alphabet-reduction is performed after each stride doubling, as shown in Figure 5. Note that the different alphabet-translation tables ($TxTable_i$) can be combined to obtain the global alphabet-translation $(\Sigma_1)^k \rightarrow \Sigma'_k$.

It must be noted that combining stride doubling with default transition compression is possible because the algorithm proposed in [7] is based on a breadth-first traversal of the DFA (and does not require additional data structures). This would not be possible using the default transition generation algorithm proposed in [6], which requires a fully generated DFA and an additional data structure with size quadratic in the number of DFA states. We observed that performing default transition compression *after* stride doubling on medium-size DFAs is infeasible on reasonably equipped machines; we have used a Linux machine with a 4GB main memory. When processing a 2K-state DFA, we ran out of memory during creation of the stride-4 version (with a reduced alphabet) whereas only 10% of the memory was used when default transition compression was combined with stride doubling.

One last problem to be addressed when generating a *k*-DFA is the preservation of correct accepting states. As for NFAs, at each state traversal we want to know which regular expressions, if any, are matched. This information can be associated either with the states or with the transitions. Unlike Brodie et al. [8], we opt for the first alternative, because the second would make the number of possible transitions explode; on each character, we can have a non-matching transition or a transition accepting any possible combination of rules. Let us assume that, when doubling the stride, we transition from state s_x to state s_y moving though state s_w, which accepts rule j. Because s_y was originally non-accepting, we must create a new state $s_y^{\{j\}}$ accepting j, and associate the transition to it. The *k*-DFA will have more states than the original one. However, s_y and $s_y^{\{j\}}$ will have the same outgoing transitions. Therefore, $s_y^{\{j\}}$ will account for a single default transition connecting it to s_y (it can be shown that s_y always has depth smaller than $s_y^{\{j\}}$). If the rule-set contains r rules and the original DFA has n_{NA} non-accepting states, then in the worst case $n_{NA}\sum_{k=1}^{r} C_r^k = n_{NA}\sum_{k=1}^{r}\binom{r}{k}$ new accepting states consisting of a single default transition can be created.

4.2 Multi-Stride DFA Generation Algorithm

The resulting DFA stride doubling algorithm is illustrated in the following pseudo-code. The original DFA *dfa* is fully represented through its transition function $\delta(state,\sum)$, whereas the computed *dfa'* is compressed and represented through its set of default and labeled transitions (*default_tx* and *labeled_tx,* correspondingly). A mapping between states and accepted rules is used in both cases (*rules* and *rules'*).

The algorithm traverses the original DFA in a breadth-first fashion (lines 5-25). For each state, in lines 8-20, the outgoing transitions are computed, whereas in lines 21-25, default transition compression is performed.

The *mapping* variable is used to determine when a new accepting state has to be created (lines 12-15). Note that new accepting states are assigned a default transition upon creation (line 15) and are not included in the traversal (second condition in line 18).

```
procedure double_stride (DFA dfa=(δ(states, Σ),rules(states))
    modifies DFA dfa'= (default_tx, labeled_tx, rules'(states')));
(1)   list queue, all; set depth, tx; state ns=size(states);
(2)   map mapping : (states, set) → states';
(3)   for state s ∈ states do mapping(s,rules(s))=s;
(4)   for state s ∈ states do rules'(s)=rules(s);
(5)   queue.push(0); all.push(0); depth[0]=0;
(6)   while (!queue.empty()) do
(7)      state s= queue.pop();
(8)      for char c1 ∈ Σ do
(9)         state t1= δ(s,c1);
(10)        for char c2 ∈ Σ do
(11)           state t2= δ(t1,c2);
(12)           if (mapping(t2, rules(t1) ∪ rules(t2))=λ) then
(13)              mapping(t2, rules(t1) ∪ rules(t2))=ns++;
(14)              rules'(ns)= rules(t1) ∪ rules(t2);
(15)              default_tx[ns]=t2; labeled_tx[ns]=Ø;
(16)           t2 = mapping(t2, rules(t1) ∪ rules(t2));
(17)           tx(s,c1c2)=t2;
(18)           if (!t2∈all & t2<size(states)) then
(19)              queue.push(t2); all.push(t2);
(20)              depth[t2]=depth[s]+1;
(21)     int reduction=1;
(22)     for state t∈ all & depth[t]<depth[s]do
(23)        if (common_tx(s,t) > reduction) then
(24)           default_tx[s]=t; reduction=common_tx(s,t);
(25)     labeled_tx[s]=uncommon_tx(s,default_tx[s]);
end;
```

5. ANALYSIS ON REAL DATA-SETS

We now analyze the characteristics of *k*-NFAs and *k*-DFAs obtained by applying the presented techniques on real-world rule-sets.

5.1 Effect of Data-Set Size on *k*-NFA/*k*-DFAs

We first study the effect of increasing the data-set size. To this end, our first set of experiments is conducted by drawing an increasing number of regular expressions from Snort [22]. In particular, we consider 22 Perl-Compatible Regular Expressions from rules matching the headers (`tcp, $HOME_NET, any, $EXTERNAL_NET, $HTTP_PORTS`). This data-set contains unanchored as well as anchored patterns with the *m*-modifier. All considered regular expressions contain character ranges and eight of them present dot-star sub-patterns (however six of those share a common prefix up to the dot-star term).

The characteristics of the *k*-NFAs and *k*-DFAs obtained with this data-set as well as the corresponding alphabet size are detailed in Table 1. As can be seen, strides of one, two and four have been tested. The following observations can be made:

First, while the number of states in the *k*-DFAs increases with the stride (due to the creation of extra accepting states), that of *k*-NFAs is basically unchanged. For larger data-sets, some NFA states are lost; in fact, some NFA states corresponding to anchored patterns can become unreachable when increasing the stride.

Second, for almost all strides and numbers of rules, the alphabet size in the *k*-NFA case is the same as that of the *k*-DFA case (recall that DFAs are minimized). This proves that the NFA-reduction algorithm works efficiently. The *k*-NFA alphabet is

Table 1: Alphabet size, number of states, and average number of transitions per state on *k*-NFAs and *k*-DFAs of different size. The number of considered regular expressions is progressively increased from 1 to 22.

# RE	Stride = 1						Stride = 2						Stride = 4					
	NFA			DFA			*2*-NFA			*2*-DFA			*4*-NFA			*4*-DFA		
	|Σ|	# state	tx/ state	|Σ|	# st.	tx/ state	|Σ|	# st.	tx/ state	|Σ|	# st.	tx/ state	|Σ|	# st.	tx/ state	|Σ|	# st.	tx/ state
1	25	45	2.6	25	48	1.8	53	45	4.6	53	50	3	93	45	8	93	54	4.4
4	34	176	2.3	34	253	1.6	145	176	5.7	145	268	3	396	176	13.2	395	311	7.9
7	39	244	2	39	351	1.5	213	244	5.9	213	387	3	669	244	16.2	669	519	9.1
10	41	302	2	41	638	1.5	267	302	6.8	267	707	3.3	1,088	302	23.2	1,088	977	13.2
13	42	360	1.9	42	762	1.5	325	358	7.3	325	879	3.5	1,647	357	31.1	1,649	1,334	17.8
16	44	441	1.9	44	1,184	1.5	412	439	8.4	412	1,401	3.9	2,315	438	40.6	2,317	2,210	27.3
19	44	547	1.7	44	1,396	1.4	441	545	7.4	441	1,661	3.6	2,387	544	34.5	2,389	2,656	24.1
22	44	636	1.7	44	1,940	1.4	470	634	7.9	470	2,351	5.3	2,920	632	41.3	2,962	3,876	33.1

slightly smaller in a few cases where stride doubling removes some unreachable states.

Third, the alphabet size grows faster on multi-stride automata than on basic automata. However, doubling the stride does not cause the alphabet size to increase quadratically, even with respect to the compressed base alphabet.

As could be foreseen, *k*-DFAs have more states than their *k*-NFA counterparts, but have far fewer (labeled) outgoing transitions per state. Interestingly enough, for stride two, the average number of outgoing transitions per state stays below 4 up to 1.5K states, and is limited to 5.3 for the 22-regular expression data-set. The same number gets to 33.1 in case of stride four. Although these numbers could be compared with 65,000 and 4.29 billion, respectively, as described in Section 4.1, and could therefore appear very promising, the additional analysis below is necessary to understand whether multi-stride automata are effective in practice.

Figure 6 and Figure 7 illustrate the distribution of outgoing transitions per state for the 22-regular expression set considered above. For stride two, about 80% of the states have one or fewer outgoing transitions in both the DFA and the NFA; between 16% and 17% of the states present more than 1 but fewer than 16 outgoing transitions, and a handful of states have a number of transitions comparable to the alphabet size. The distribution is similar in the case of stride four; with the exception that the percentage of zero-transition states in the *4*-DFA increases sensibly and the fraction of states with a large number of outgoing transitions increases slightly, too.

These distributions suggest that, with a proper encoding, multi-stride automata can be effective with medium-size DFAs and NFAs. Moreover, it may be beneficial to distinguish the 80%-90% of the states having a limited number of outgoing transitions from the other ones, and to adopt two different state representations.

5.2 Large Data-Sets

Table 2 reports the outcome of performing stride doubling on a larger data-set. In particular, we considered all Snort rules matching the headers (`tcp, $HOME_NET, any, $EXTERNAL_NET, any`) and extracted the corresponding Perl Compatible Regular Expressions. Compiling these patterns into a single DFA is infeasible [10][11]; therefore, we partitioned the patterns into the minimum number of sets yielding feasible DFAs (i.e., 3). Even so, it can be observed that the number of states in

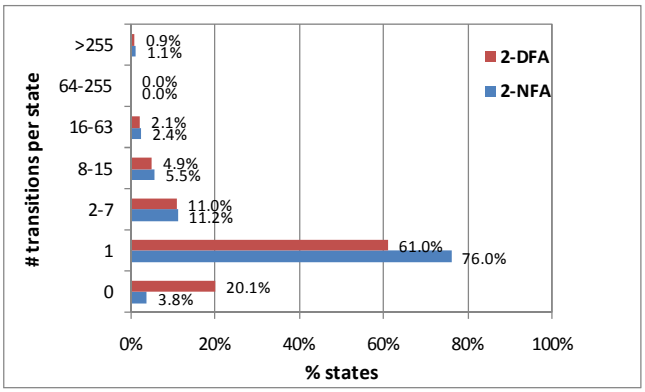

Figure 6: Distribution of the number of outgoing (labeled) transitions per state for stride-2 automata. The last data-set in Table 1 is used.

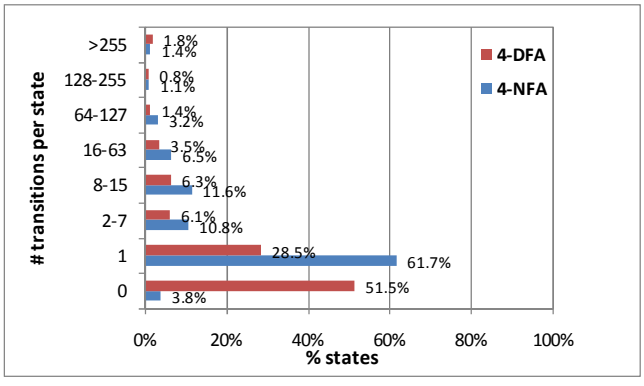

Figure 7: Distribution of the number of outgoing (labeled) transitions per state for stride-4 automata. The last data-set in Table 1 is used.

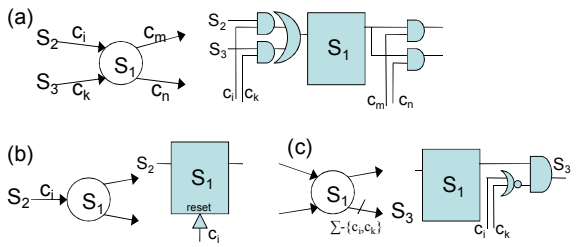

Figure 8: State encoding in FPGA design: (a) basic encoding by Sidhu and Prasanna, (b) single input optimization, (c) multiple outputs optimization.

the DFA far exceeds that in the corresponding NFAs.

It can be observed that the size of the stride-2 alphabet, Σ_2', is larger than the medium-size case but is still far smaller than 65K. Building the *4*-DFA on our 4GB desktop machine for data-sets of this size is impractical: not only do we start with a larger number of states, but stride doubling must also process $|\Sigma_2'|^2$ possible outgoing transitions from each state.

As far as outgoing transition distribution is concerned, more than 50% of the states have one or fewer outgoing transitions; there are 2-7 outgoing transitions for about 20% of the states and 8-15 for another 20%. Only a small percentage have a larger number of outgoing transitions.

6. IMPLEMENTATION

We now consider implementing *k*-NFAs on FPGAs, and *k*-DFAs on memory-centric architectures.

6.1 FPGA

The main motivation for implementing NFAs on an FPGA is the following. During operation, processing one input character can trigger several parallel NFA state traversals. In a memory-based system, this implies multiple memory accesses per character processed. On the other hand, the parallelism available in an FPGA allows processing one input character per clock cycle independent of the number of active NFA states. In particular, this is possible by using a *one-hot encoding* to represent states and alphabet symbols, as proposed by Sidhu and Prasanna in [14].

Figure 8 (a) represents the one-hot encoding scheme.

Table 2: Characteristics of *k*-NFAs and *k*-DFAs accepting large pattern-sets. To allow a feasible DFA representation data-set *any*, consisting of 99 regular expressions, is split into any_1, any_2 and any_3.

	Rule-set	Stride = 1			Stride = 2		
		|Σ|	#states	#tx/state	|Σ|	#states	#tx/state
k-NFA	*any*	78	2,086	4.5	1969	2,091	56.8
	any_1	59	655	4.9	850	659	39.3
	any_2	45	761	2.9	579	761	17.1
	any_3	61	689	3.5	633	690	17.5
k-DFA	any_1	59	23,846	2.4	850	28,223	11.4
	any_2	45	86,977	3.4	579	102,940	10.5
	any_3	60	14,084	2.1	627	19,344	11.1

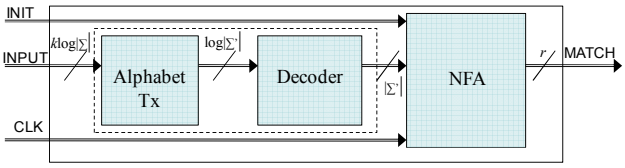

Figure 9: Block diagram of FPGA implementation

Basically, each NFA state is represented by a flip-flop and each symbol by a bit that is set when the input character matches the symbol. The output of the flip-flop representing state *s* is *and*-ed with the symbols on its outgoing transitions, and the resulting signals are routed toward the flip-flops encoding the target states.

To reduce the number of LUTs used in the design, we adopted the following optimizations:

- *Single input optimization*—Figure 8 (b): each state having a single incoming transition is represented through a flip-flop with reset signal. The flip-flop is reset by the negation of the input symbol and is directly connected to the source state.
- *Multiple outputs optimization*—Figure 8 (c): each state transiting to a target on a set of symbols *S* whose size exceeds a given threshold has the corresponding transition represented in terms of Σ-*S*.

Given the encoding above, we wrote a logic generator that automatically translates an NFA description into an equivalent representation in the hardware description language VHDL, which can be processed by FPGA synthesis tools to produce a layout for an FPGA.

Notice that the NFA-reduction scheme described in Section 3.1 is useful for two reasons. First, by reducing the number of states, it decreases the number of required flip-flops. Second, by reducing the transitions, it diminishes the need for LUTs and facilitates the place and route of signals. This, in turn, can lead to higher operating frequencies. The latter effect also is fostered by the use of alphabet-reduction.

6.1.1 Overall design

The overall design schematic is shown in Figure 9. Besides the clock and an INIT signal, which is set at the beginning of every input stream, the module receives *k* characters at every clock cycle, *k* being the stride of the deployed NFA. The output to the module is a set of signals representing the match of the corresponding regular expressions.

Because alphabet-reduction is performed, the input character must first go through an alphabet-translation block. The output of this block must be decoded to produce a one-hot encoding of the processed character, which is input to the NFA block. The NFA block is implemented as described above.

As far as alphabet-translation is concerned, we tested two implementations: *(i)* storing the content of the alphabet-translation tables in block memory and implementing the decoder in combinatorial logic; and *(ii)* combining alphabet-translation and one-hot encoding in combinatorial logic. We reiterate that the underlying alphabet-translation data are NFA-dependent and produced through the algorithm summarized in Section 3.3.

In the first implementation, for stride greater than one, the translation tables were accessed hierarchically as done in [8]. Even though the block memory accesses were pipelined, better performance was achieved using the second approach, especially for large designs. The need for a decoder did not really lead to

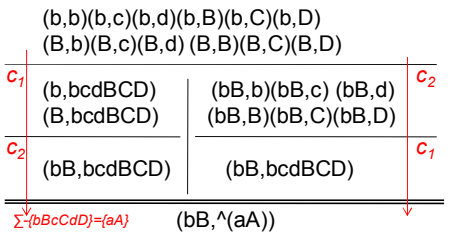

Figure 10: Term-minimization and reduction on Σ={a,b,c,d,A,B,C,D}, k=2. The starting point is the enumeration of initial character pairs. Two possible reductions lead to the same result: (c_1=b OR c_1=B) AND NOT (c_2=a OR c_2=A).

significant savings in LUT usage when the first approach was adopted. We will therefore report the results of the second implementation: combining alphabet-translation and one-hot encoding in combinatorial logic.

6.1.2 Alphabet-translation for k-NFA

Given an NFA and a stride k, the techniques described in Section 3 provide the translation from every possible permutation of k characters of the initial alphabet Σ into a single character of the final alphabet Σ'. For strides greater than one, we want to represent this translation efficiently in combinatorial logic.

Let us first define the problem. For each character c', its translation is a set of tuples $T_i=(S_{i1},.., S_{ik})$, where S_{ij} represents the set of characters of Σ that can be found at position j in a particular combination of inputs. In logic, $c'=OR_i(AND_j(c_j \in S_{ij}))$, where $j \in \{1..k\}$. Initially, we are given potentially many T_i, and all S_{ij} consist of a single element. The goal is to reduce the number of T_i while keeping the size of S_{ij} as small as possible. The first goal is achieved through *term-minimization*, and the second through *term-reduction*.

Term-minimization is similar in spirit to the Quine-McCluskey algorithm. At each step, two terms T_m and T_n are combined if, and only if, they differ at a single position j. The combined term T_{mn} will have $S_{mj} \cup S_{nj}$ at position j, and be otherwise identical to T_m and T_n. Note that a term-minimization can trigger new ones. When no more term-minimizations are possible, term-reduction will look for large S_{ij}, and replace them with $^(\Sigma - S_{ij})$. Note that if S_{ij} is equal to Σ, then the condition on the character at position j in term T_i is thereby eliminated. Figure 10 illustrates this procedure on a simple example.

6.1.3 Results

To evaluate the design, we synthesized three pattern sets of different sizes on Xilinx Virtex 5 – device XC5VLX50 [24]. We used Xilinx ISE design suite, v. 10.1. The data-sets are drawn from a snapshot of Snort rules as of January 2008 [22]. In particular, we extracted all Perl-compatible regular expressions from rules matching three header groups differing in the destination port (`tcp, $HOME_NET, any, $EXTERNAL_NET, any/25/$HTTP_PORTS`), and named the corresponding pattern-sets *any*, *25* and *http*. As can be verified, all corresponding regular expressions contain character ranges, and about half of them contain dot-star terms.

We considered three NFA deployments: stride one with full alphabet and stride one and two with alphabet-reduction. The characteristics of the corresponding NFAs and alphabets are shown in Table 3, columns 3-6. In column 7 we show how many times the optimizations represented in Figure 8 (b) and (c) are invoked. Note that the multiple outputs optimization reduces the effective alphabet size in the design: hence, we report the two different numbers in column 6. The results of synthesis, mapping, and place & route are shown in Table 3, columns 8-13. The *par* tool was configured to pack slices as much as possible. From the analysis of the data the following conclusions can be drawn:

First, alphabet-reduction has the combined effect of reducing the number of transitions and allowing more optimizations. This translates into a lower LUT utilization, which for larger designs (*http* data-set) affects the global slice utilization. More important, the lower number of LUTs and wires implied by alphabet-reduction makes the place & route operation, leading to a higher operating frequency.

Second, the increased complexity of a stride-2 NFA leads to higher LUT and overall logic utilization. However, the penalty on the operating frequency is small compared with the factor 2 improvement implied by the stride. This leads to higher throughput compared with the stride-1 design. If, for a given data-set, enough logic is available on chip, stride-2 NFA can provide twice the processing throughput. In particular, the larger the data-sets, the smaller the utilization penalty compared with a stride-1 solution and the larger the throughput improvement.

For the sake of comparison we synthesized the same designs on a Virtex 4 – device XC4VLX25 [24]. We obtained comparable results, except for a higher slice utilization (ranging from 10% for the smallest design to 60% for the largest one).

Finally, these results suggest that, using the set of techniques detailed in this paper, about 1,200 complex regular expressions could be deployed on a Virtex-5 FPGA through a stride-1 NFA, and about 800-900 through a stride-2 NFA. Note that optimizations proposed in the literature to further reduce the slice utilization and improve performance [19] are also applicable to our design.

6.2 Memory-Centric Architectures

We now assume to have an ASIC architecture consisting of small processing engines coupled with memory banks, and we compute the memory requirement and the throughput assuming single packet processing. The overall throughput can be increased by allowing multiple packet flows to be processed in parallel, as done in [6].

To allow one memory access per state traversal, we encode the states using the variant of content addressing described in [7]. In this scheme, the symbols on which labeled transitions are defined are encoded in the state identifiers, and a hashing mechanism is used to map state identifiers to memory addresses.

Let us assume to deploy the DFAs in Table 2. Due to the size of the alphabets, the bits needed to encode each symbol are 6 for stride one and 10 for stride two. Assuming 64-bit wide memory accesses, the states that can be encoded with content addressing are the ones presenting no more than 10 outgoing transitions for stride one, and no more than 6 outgoing transitions for stride two. To ensure one memory access per state traversal, the remaining states must be fully represented, requiring $|\Sigma'|$ memory entries each.

For the considered DFAs, the fraction of the states that can be encoded with content addressing is in excess of 98% for stride one, and about 82% for stride two. The memory requirement of

Table 3: Results obtained by synthesizing three Snort pattern-sets on a Xilinx Virtex 5 FPGA. Three deployments are considered: stride one with and without alphabet-reduction (*1/red* and *1/full*), and stride two with alphabet-reduction (*2/red*). The number of single input (*si*) and multiple output (*mo*) optimizations performed in each case is reported. The LUT are broken down in the one used for alphabet-translations (ΣT), and the ones used within the NFA block (*NFA*).

Rule-set	# RE	k-NFA characteristics				VHDL	FPGA design					
		type *k*/ Σ	# states	# tx	|Σ| tot/eff.	optimiz. (*si/mo*)	LUT (ΣT/NFA)	FF	slices	period (ns)	slice util.	Perf. (Gbps)
any	99	***1/ full***	2,086	19,975	256/106	248/45	127/1080	2,084	604	2.40	8%	3.33
		1/red	2,086	9,329	78/77	1638/45	66/340	2,084	589	1.96	8%	4.07
		2/ red	2,091	118,818	1969/1968	1249/41	2093/2622	2,091	1,370	2.53	19%	6.32
25	79	***1/ full***	2,147	23,210	256/97	136/66	60/1601	2,147	630	2.01	8%	3.99
		1/red	2,147	8,428	68/67	1815/66	45/270	2,147	604	1.97	8%	4.05
		2/ red	2,142	129,889	1640/1639	1424/65	1583/1858	2,141	1,266	2.29	17%	7.00
http	406	***1/ full***	7,684	80,239	256/92	828/247	182/3672	7,684	2,119	3.08	29%	2.60
		1/red	7,684	25,351	64/63	7157/247	152/506	7,684	2,014	2.37	27%	3.38
		2/ red	7,683	581,879	2206/2205	6215/232	2303/4127	7,681	3,371	2.61	46%	6.13

the encoded states for data-sets $any_1/any_2/any_3$ are 505KB/2.9MB/299KB in the case of stride one, and 356KB/1.27MB/244KB for stride two. If fully represented on the reduced alphabet, the memory requirement of the remaining states is 200KB/55KB/48KB for stride one and 32MB/81MB/16MB for stride two. In the multi-stride case this representation is clearly inefficient.

One possible alternative consists of moving the states that cannot be encoded through content addressing in logic rather than storing them in memory. Specifically, each state *s* having outgoing transitions to a set of targets *T* is represented through a decoder, which translates tuples of input characters (character pairs in case of stride two) into state identifiers belonging to *T*. The logic implementing each decoder can be minimized through the algorithm proposed in Section 6.1.2 for alphabet-translation. Note that the size of each decoder is *independent of the alphabet size*, and depends only on the number of distinct targets a state has. If state *s* transitions to *t* on several character pairs, term-minimization and reduction will group them together, decreasing the required logic gates. After default transition compression, the majority of the states are left with fewer than 10 targets, leading to very small decoders.

Given the limited memory storage requirement of the design, we can assume coupling the processing engines with SRAM banks clocked at 500 MHz. The minimum achievable throughput is 2 Gbps with stride one and 4 Gbps for stride two. This number takes into account a deceleration factor of two due to default transitions [7]. Because the states not encoded through content addressing are effectively fully represented, the effect of default transitions is reduced and the throughput tends to approach 4 Gbps for stride one and 8 Gbps for stride two.

7. CONCLUSION

In this paper we explored the joint application of edge-minimization, alphabet-reduction, and stride-increasing on both NFAs and DFAs, and evaluated the resulting automata on FPGAs and memory-based ASICs. Our results show that more than 800 complex regular expressions can be deployed on an FPGA sustaining a throughput of 6 Gbps. An ASIC solution using embedded SRAM clocked at 500 MHz can guarantee a minimum throughput of 4 Gbps on a single flow. In the latter case, parallel flow processing would allow higher performance.

8. ACKNOWLEDGMENTS

The authors wish to thank Ron Cytron for his useful insights and David Richard for his help with the FPGA implementation. This work has been supported by National Science Foundation grants CCF-0430012 and CCF-0427794 and by Intel's gifts.

REFERENCES

[1] A. Aho and M. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," Communication of ACM, 1975.

[2] J. Hopcroft and J Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison Wesley, 1979.

[3] J. Hopcroft, "An nlogn algorithm for minimizing states in a finite automaton," in Theory of Machines and Computation, J. Kohavi, Ed. New York: Academic, 1971, pp. 189-196.

[4] K. Thompson, "Regular expression search algorithm," in Communications of the ACM, vol. 11, Issue. 6, June 1968, pp. 419-422.

[5] R. McNaughton and H. Yamada, "Regular Expressions and State Graphs for Automata," in IEEE Transactions on Electronic Computers, EC-9(1), pp. 39–47, 1960.

[6] S. Kumar et al., "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in ACM SIGCOMM, Sept 2006.

[7] M. Becchi and P. Crowley, "An Improved Algorithm to Accelerate Regular Expression Evaluation," in ANCS 2007.

[8] B. Brodie et al., "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching," in ISCA 2006.

[9] S. Kumar et al., "Advanced Algorithms for Fast and Scalable Deep Packet Inspection," in ANCS 2006.

[10] F. Yu et al., “Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection,” in ANCS 2006.

[11] M. Becchi and P. Crowley, “A Hybrid Finite Automaton for Practical Deep Packet Inspection,” in CoNEXT 2007.

[12] S. Kumar et al., "Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia," in ANCS 2007.

[13] R. Smith et al.,"XFA: Faster Signature Matching with Extended Automata," in 2008 IEEE Symposium on Security and Privacy.

[14] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in FCCM 2001.

[15] C. R. Clark and D. E. Schimmel, “Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns,” in FPL 2003.

[16] J. Moscola et al., “Implementation of a content-scanning module for an internet firewall,” in FCCM 2003.

[17] R. Franklin et al., “Assisting Network Intrusion Detection with Reconfigurable Hardware,” in FCCM 2002.

[18] C. R. Clark et al., “Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns,” in FLP 2003.

[19] I. Sourdis et al., "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," in FCCM 2004.

[20] A. Mitra et al., “Compiling PCRE to FPGA for Accelerating SNORT IDS,” in ANCS 2007.

[21] M. Roesch, “SNORT: Lightweight Intrusion Detection for Networks,” in 13th System Administration Conf., Nov 1999.

[22] SNORT: http://www.snort.org

[23] R. Sommer and V. Paxson, “Enhancing byte-level network intrusion detection signatures with context,” in CCS 2003.

[24] Xilinx: http://www.xilinx.com