LaFA: Lookahead Finite Automata for Scalable Regular Expression Detection

Masanori Bando
Department of Electrical and Computer Engineering
Polytechnic Institute of NYU
Brooklyn, NY
mbando01@students.poly.edu

N. Sertac Artan
Department of Electrical and Computer Engineering
Polytechnic Institute of NYU
Brooklyn, NY
sartan@poly.edu

H. Jonathan Chao
Department of Electrical and Computer Engineering
Polytechnic Institute of NYU
Brooklyn, NY
chao@poly.edu

ABSTRACT

Although Regular Expressions (RegExes) have been widely used in network security applications, their inherent complexity often limits the total number of RegExes that can be detected using a single chip for a reasonable throughput. This limit on the number of RegExes impairs the scalability of today's RegEx detection systems. The scalability of existing schemes is generally limited by the traditional per character state processing and state transition detection paradigm. The main focus of existing schemes is in optimizing the number of states and the required transitions, but not the suboptimal character-based detection method. Furthermore, the potential benefits of reduced number of operations and states using out-of-sequence detection methods have not been explored. In this paper, we propose Lookahead Finite Automata (LaFA) to perform scalable RegEx detection using very small amount of memory. LaFA's memory requirement is very small due to the following three areas of effort described in this paper: (1) Different parts of a RegEx, namely RegEx components, are detected using different detectors, each of which is specialized and optimized for the detection of a certain RegEx component. (2) We systematically reorder the RegEx component detection sequence, which provides us with new possibilities for memory optimization. (3) Many redundant states in classical finite automata are identified and eliminated in LaFA. Our simulations show that LaFA requires an order of magnitude less memory compared to today's state-of-the-art RegEx detection systems. A single commodity Field Programmable Gate Array (FPGA) chip can accommodate up to twenty-five thousand (25k) RegExes. Based on the throughput of our LaFA prototype on FPGA, we estimated that a 34-Gbps throughput can be achieved.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: General, Security and protection (e.g., firewalls)

General Terms

Algorithms, Design, Security

Keywords

Regular Expressions, LaFA, Finite Automation, Deep Packet Inspection, Network Intrusion Detection System, FPGA

1. INTRODUCTION

Regular Expressions (RegExes) are used to flexibly represent complex string patterns in many applications ranging from Network Intrusion Detection and Prevention Systems (NIDPS) [1, 2] to compilers [3] and DNA multiple sequence alignment [4, 14]. For instance, RegExes are used for representing programming language tokens in compilers or constraint formulation for DNA multiple sequence alignment. In particular, NIDPSs Bro [1] and Snort [2] and Linux Application Level Packet Classifier (L7 filter) [20] use RegExes to represent attack signatures or packet classifiers.

Finite Automata (FA) are the de-facto tools to address the RegEx detection problem. For RegEx detection on an input, the FA starts at an initial state. Then, for each character in the input, the FA makes a transition to the next state, which is determined by the previous state and the current input character. If the resulting state is unique, the FA is called a *Deterministic Finite Automaton* (DFA); otherwise, it is called a *Non-Deterministic Finite Automaton* (NFA) [25]. For the trade-off between performance and resource usage, NFA and DFA represent two extreme cases. DFA has constant time complexity since it guarantees only one state transition per character by definition. NFA allows multiple simultaneous state transitions leading to a higher time complexity. The constant time complexity per character allows DFA to achieve high-speed RegEx detection, which makes DFA the preferred approach. This is especially so for software solutions, since DFA can be serially executed fast on commodity CPUs. NFA, on the other hand, requires massive parallelism, making it harder to implement and update. The price paid for the high speed of DFA is its prohibitively large memory requirement.

Recent applications, most notably Deep Packet Inspection for NIDPS, emerge that require RegEx detection scalable to high quantities of complex RegExes. This need for scalability makes the DFA impractical for such applications, due to its large memory requirement. Since NFA is not high-speed, RegEx detection scalable to high quantities of complex RegExes at high speeds is an open issue.

We make three observations in this paper that are common to most of today's state-of-the-art RegEx detection systems and, we believe, that limit the scalability of these systems. First of all, RegExes consist of a variety of different *components* such as character classes or repetitions. Due to this variety, it is hard to identify a method that is efficient for concurrently detecting all of these different components of a RegEx. However, in most cases today, these heterogeneous components are detected by a state machine that consists of homogeneous states. This leads to inefficiency and, as a result, the scalability of such systems is poor. Secondly, the order of components in a RegEx is preserved in the state machine detecting this RegEx. However, it may be beneficial to change the order of the detection of components in a RegEx. For instance, it is easy to detect simple strings as they are expected to appear less frequently, whereas others are harder to detect (for instance, character classes) and may appear more frequently. By reordering the detection of the components, the trade-off between detection complexity and appearance frequency can be exploited for better scalability. Finally, most RegExes share similar components. In the traditional FA approaches, a small state machine is used to detect a component in a RegEx. This state machine is duplicated since the similar component may appear multiple times in different RegExes. Furthermore, most of the time, these RegExes sharing this component cannot appear at the same time in the input. As a result, the repetition of the same state machine for different RegExes introduces redundancy and limits the scalability of the RegEx detection system.

The rest of this paper is organized as follows: A description of our novel detection approach is given in Section 2. Section 3 describes the LaFA data structure and architecture. Section 4 is a discussion section. Section 5 evaluates the performance of our architecture and presents simulation results. Section 6 discusses related work. We conclude the paper in Section 7.

2. LOOKAHEAD FINITE AUTOMATA (LAFA)

LaFA is a finite automata that we optimize for scalable RegEx detection. LaFA is used for representing a set of n RegExes, $\mathbf{R} = \{r_1, r_2, \dots r_n\}$, which can also be called a *RegEx Database*. For instance, this set of RegExes can be a set of Snort signatures, or Bro or Linux Layer 7 rules. An associated LaFA RegEx detection system can be queried with an input P, such as a network packet. The system will return *a match* along with a list of matching RegExes if input P matches one or more of the RegExes in $\mathbf{R}$. Otherwise, a *no match* is returned.

In this section, we illustrate the important features of LaFA with the help of a warm-up example.

2.1 A Warm-up Example

Let us consider the RegEx set, $\mathbf{R}$ with three RegExes, r_1, r_2, and r_3 as shown in Figure 1 (a). The NFA representation of $\mathbf{R}$ is shown in Figure 1 (b).

2.1.1 Separation of Simple String Matching

A *simple string* is a fixed sequence of characters such as "*abc*" and "*op*" in r_1. In practice, it can be a word from the dictionary or an excerpt from an executable file. Consider the state diagram of NFA in Figure 1 (b). A separate state is defined for each character in *simple strings.* Although a per

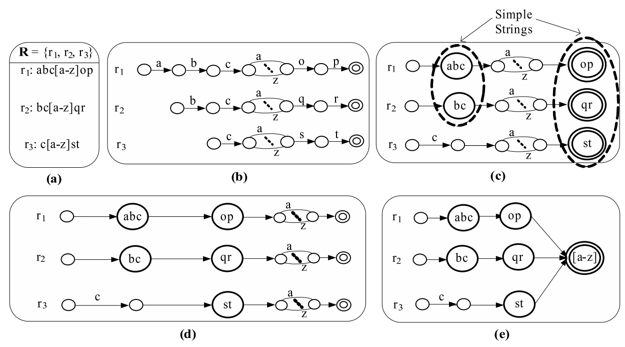


Figure 1: An example illustrating the transformation from a RegEx set R into the corresponding LaFA: (a) The RegEx set R, (b) The NFA corresponding to R, (c) Separation of simple strings, (d) Reordering of the detection sequence, and (e) The final step, the sharing of complex detection modules which leads to the LaFA corresponding to R.

character state transition looks simple, it may require many state transitions per character when multiple RegExes are to be matched against the RegEx database. Consider the input characters "*abc*". The "*a*" state of r_1 is first matched followed by the "*b*" states of both r_1 and r_2. Then, as the input character "*c*" arrives, the "*c*" states of all FAs are matched. All three FAs concurrently transition to their [*a-z*] states. Imagine the resulting large number of concurrent active states for a database of hundreds RegExes. A large number of concurrent states often translates to high resource requirements and low throughput.

Note that the detection of simple strings, namely the *exact string matching* is a well-studied problem with many optimized solutions [5,6,8,13,15,24,27]. In particular, there are high-throughput hardware solutions with very small memory footprints [5,8,24]. In this paper, we argue that neither NFA nor DFA is the most efficient method to detect these simple strings. We propose to merge the states of each simple string portion of the automata ("*abc*", "*bc*", "*op*", "*qr*", and "*st*") into a single super state. These simple strings can then be detected using high-speed exact string matching methods. The resulting FA, as shown in Figure 1 (c), has fewer states than NFA. By generating one detection event per simple string, rather than making many state transitions for each input character as in an NFA or DFA, LaFA significantly reduces the number of operations as Section 5 shows.

2.1.2 Reordering The Detection Sequence

By definition, NFA and DFA are constructed based on the order of components in the RegEx. Therefore, the reordering of the sequence of detection is difficult. In Figure 1 (c), each FA consists of three components : (1) a simple string, (2) a character class [*a-z*] , and (3) another simple string. Consider matching the example input string "*abcxop*" against the states in Figure 1(c). For this particular input, after matching earlier states for "*abc*", "*bc*", and "*c*", all three FAs go to their respective super states of [*a-z*] for matching "*x*", concurrently. As the input continues to "*op*", however, the input matches only to the first RegEx r_1. Note that, in fact, there is no possible input that can match to more than one RegEx for this example RegEx set.We observe that the FAs for r_2 and r_3 made the unnecessary [*a-z*] detection just to abandon the search at the end when neither "*qr*" or "*st*" is found in the input.

In LaFA, we propose to reorder the detection sequence as shown in Figure 1 (d). Here, the third component in each RegEx, the simple string, is swapped with the second component, the character class [*a-z*]. So, for instance for r_1, only if both "*abc*" and "*op*" are detected in order (not necessarily consecutively), the [*a-z*] detection is performed. This way, for our example input of "*abcxop*", exactly one [*a-z*] detection is performed rather than three concurrent [*a-z*] detections as in the case before reordering. In general, we can classify components in RegExes based on the information revealed about a RegEx, if a certain component in that RegEx is found. Consider r_1 as an example again (r_1 : *abc*[*a-z*]*op*). Detecting the string *abc* in the input reveals more information than detecting a range of characters from a to z ([*a-z*]). Intuitively, one expects that more restrictive matching components like *abc* can occur much less frequently in inputs, compared to less restrictive components like [*a-z*]. In this paper, we call components that reveal more information like *abc deep components* and components that do not reveal much information like [*a-z*] *shallow components*.

There are deep components that are more complex to detect than simple strings. For example, a string repetition element $\backslash n\{x, y\}$ is *deep* but requires more complex operations than an exact string matching. One would prefer to detect simple components before the more complex ones so that the probability of requiring evaluation of the latter ones is reduced. Therefore, we further classify components according to how complex it is to detect that component. Figure 2 shows a matrix of the depth and the detection complexity of components with examples. LaFA detects simple and deep components ahead of complex and/or shallow ones. We call this core notion of LaFA **"Lookahead"** in the rest of this paper. Details of the Lookahead operations are presented in Section 3.3.

In Figure 1, the states of complex components are shown with larger circles. The thickness of the state circle reflects the depth of the corresponding component.

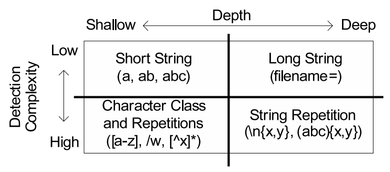

Shallow
Depth
Deep
Low
High
Detection Complexity
Short String (a, ab, abc)
Long String (filename=)
Character Class and Repetitions ([a-z], /w, [^x]*)
String Repetition (\n{x,y}, (abc){x,y})

Figure 2: Matrix of RegEx components based on their detection complexity and depth (information revealed about the RegEx).

2.1.3 Event-Based Approach

A typical detection operation involving NFA or DFA requires at least one state transition for each input character. As a result, the handling of state transitions takes up a prominent portion of the operation costs. By using our proposed super states, we can reduce the number of state transitions, on average, to less than one per input character using a highly optimized detector explained in Section 2.1.1. Every time a component such as a simple string is detected, a *detection event* is generated. These events are then correlated to see if they correspond to one or more RegExes in the RegEx set. The details of this important operation are covered in Section 3.

2.1.4 Variable String Detection and Sharing

As can be seen in the previous steps, detection of some components (in our example, the [*a-z*] component) only needs to be done once at any given time. Instead of duplicating those components in multiple RegExes, we can save resources and memory by sharing those rarely used components. The resulting LaFA representation is shown in Figure 1 (e). Furthermore, we also propose special modules that are specifically designed to detect certain types of variable stringsto replace these states. Details of such modules are given in Section 3.3.

2.2 Summary

In this section, we have illustrated the core ideas behind LaFA (**L**ook**a**head **F**inite **A**utomata) that facilitate the reduction of memory requirements and detection complexity using simple examples. RegExes are first represented in NFA format. Then, the states of simple strings are identified and merged into super states. The components are reordered according to their depth and complexity. Finally, the states of components that can potentially be shared among RegExes are merged. In the next section, we present the details of the LaFA data structure and the associated scalable RegEx detection architecture.

3. LAFA DATA STRUCTURE AND ARCHITECTURE

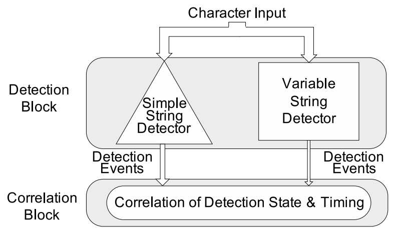


Figure 3: The LaFA Architecture.

Figure 3 is a simple block diagram of the LaFA architecture. The LaFA architecture consists of two blocks, the *Detection Block* and the *Correlation Block*. The *Detection Block* is responsible for detecting the components in the input string. The *Simple-String Detector* is a highly optimized exact string matching system used to detect simple strings and the *Variable String Detector* consists of highly optimized variable string detection modules proposed in this paper. Recall that in Section 2, we mentioned a component that tracks states. The *Correlation block* is responsible for this task. The correlation block inspects the sequence of components in the input string (*i.e.,* order and timing) to determine whether the input matches any of the RegExes in its RegEx database.

The detectors in the detection block communicate their findings to the correlation block by sending *detection events*. A detector in the detection block generates a detection event when it detects a component in the input. Each detection event consists of a unique ID for the detected component, and its location in the input packet. Extra information, such as the length of the detected string, may accompany the event if necessary. There can be two sources of an event. (1) A *string event* is generated when a simple string is detected. (2) The *in-line event* is generated by in-line lookup modules. The in-line event is described in detail in Section 3.3. The generated events prompt a detection sequence in a correlation block to proceed to the detection state.

In the rest of this section, the details of individual blocks in the LaFA architecture and their interaction are presented. LaFA is implemented in a hardware and performance is shown in Section 5.

3.1 Construction of the LaFA Data Structure

This section presents the formal construction of the LaFA data structure. Given a set of RegExes, **R**, the LaFA data structure is constructed in three steps: (1) RegEx partitioning, (2) Component reordering, and (3) Node structure construction.

The first step of LaFA data structure construction is to partition each RegEx into simple strings and variable strings. This partitioning facilitates the detection of a simple string and variable string. In this paper, each simple string is represented with the letter S and each variable string is represented with the letter V for simple representations of components. The variable strings are further classified as Section 3.3 explains.

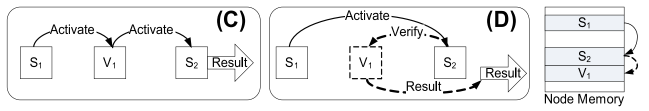


Figure 4: Component Re-ordering for r_1. ($S_1: abc, V_1: [a\text{-}z], S_2: op$)

The next step in data structure construction is component re-ordering. Component re-ordering is introduced in Section 2.1.2. We discuss it in further detail here as we introduce a new term *node* shown as square boxes in Figure 4. In the traditional sequence shown in (C), components are activated one by one following the sequence of RegEx. In contrast, the LaFA sequence shown in (D) jumps (lookahead) variable string V_1 and activates S_2 first. LaFA verifies variable strings (*e.g., V_1*) using special modules for which specific information related to individual components is also needed. The location to store this component verification information (square boxes in the figure) is called a *node*. Nodes are stored in memory, which is also illustrated in the figure. Verification of S_2 and V_1 is performed successively. The nodes of S_2 and V_1 are stored at consecutive locations in the node memory. As long as S_2 and V_1 are allocated to consecutive memory locations, S_1 and S_2 can be stored anywhere in the memory. This helps in balancing the nodes in the node memory and the locations of simple strings are resolved by pointers. We discuss the node structure next.

The final step in the LaFA data structure construction is to identify the contents of the node. As in any other state machine, the LaFA state machine also has basic information in every state. As discussed, LaFA states are associated with nodes and these nodes contain the verification information specific to each component. The node also contains the current state, which indicates whether the current state is active or inactive. Conventional FAs, such as NFA and DFA, change states constantly on every character received; hence, tracking timing is not an issue. In the LaFA state machine, however, the input (event) is generated by a simple string detector. Thus, detection timing needs to be verified. Because of this characteristic, we store timing information in the node. A component can have variable length as well and for this reason, a node keeps two timing information pieces, minimum time and maximum time. A simple string must be detected within this time range. Summarizing node information, each node has the following: 1. A pointer to the next state, 2. the status of the current state, 3,4. minimum and maximum time values. In the following sections, we show why this information is needed and how it is used.

The variable strings does not receive an event, but they are triggered by a simple string. Once a simple string is detected, variable strings are verified consecutively as Section 2.1.2 describes. To verify consecutive variable strings, we create a node for each variable strings and store the necessary information, which is listed below: 1. module ID (define which special module is used to detect), 2. pattern ID (for example [*a-z*] and [0-9] are assigned different IDs), 3,4. minimum and maximum repetition value for those components that have repetition notations, 5,6 minimum and maximum timing values.

This section presents the proposed LaFA architecture for which a prototype has been implemented on a Virtex-4 FPGA and the LaFA scheme has been verified. As mentioned before, there are two main blocks in LaFA, the detection block and the correlation block. We explain them in the following subsections.

3.2 Correlation Block

Practical RegEx sets include hundreds of RegExes that need to be programmed in the correlation block. Figure 5 illustrates how multiple RegExes are organized in a correlation block along with node activation and timing verification operations. The correlation block is organized into columns of nodes where each node is similar to a state in a traditional FA, yet the node includes more information than a state. A node in LaFA is also likely to correspond to multiple states in a traditional FA. Each column corresponds to one component and each column has one node for each RegEx that this component appears in. Once, the correlation block receives a detection event for a particular component all the nodes in the corresponding column are checked to see whether the detection of this RegEx is in progress and which component is expected next. The proposed Lookahead detection flow must satisfy certain timing constraints. Only when the detection of contents with the proper timing is confirmed, is the next node activated. To determine this, each node has a static pointer to its next node.

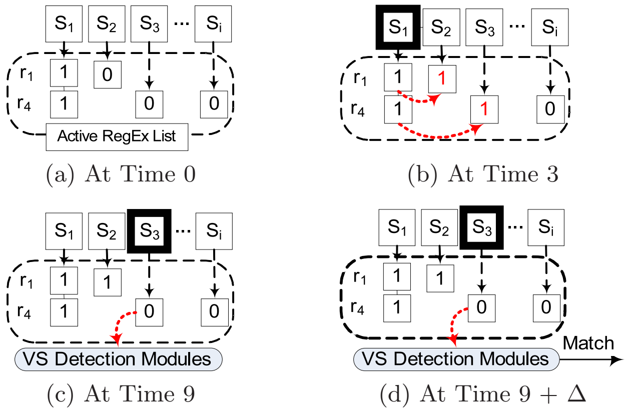
In the example case RegEx, r_1 and r_4 contain the same component, simple string S_1. Two nodes are examined once S_1 is detected. When S_1 is detected, as indicated by a set bit in the node, a node transition occurs to the next node corresponding to the next component in the given RegEx.

Correlation Detection Operation and Timing

The sequence of input character and arrival time is shown below along with two example RegExes, RegEx1(r_1) and RegEx4(r_4). We add another RegEx r_4 to our original example for the more general case. The unit of *time* is defined as the interval required to receive one character.

$$Input : a\ b\ c\ e\ f\ g\ x\ y\ z$$
$$Time : 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9$$

RegEx1:	$S_1\ V_1\ S_2$	$S_1 : abc$	$V_1 : [a\text{-}z]$	$S_2 : op$
RegEx4:	$S_1\ V_2\ S_3$	$S_1 : abc$	$V_2 : [\hat{}x]\{3\}$	$S_3 : xyz$



(a) At Time 0 (b) At Time 3 (c) At Time 9 (d) At Time 9 + Δ

Figure 5: Example of simple string-based detection operations and timing verifications. The figures are sorted in chronological order. (VS=Variable String)

Initially, the Active RegEx List is as shown in Figure 5(a). In the figure, nodes are represented as boxes with "0" or "1". "0" symbolizes that the node is inactive while "1" is the symbol of an active node. The simple string S_1 : "*abc*" is the first component in both r_1 and r_4; therefore, they are active. Upon detecting a simple string, S_1 : "*abc*" at time 3, the simple-string detector generates an event and passes it to LaFA. When the event is received, based on RegEx stored in the node, the next simple string, *i.e.,* S_2 : "*op*" of r_1 and S_3 : "xyz" of r_4 is activated by setting the bit corresponding to RegEx r_1 and r_4 to active "1" as shown in Figure 5(b). Next, at time 9, S_3 is detected and the detection time of S_3 is verified based on the distance between S_3 and S_1. If the timing verification is successful then, as Figure 5(c) shows, a query is sent to the variable string detection module to verify the RegEx element V_2. The processing time of the detection module is denoted by Δ. Hence, after finishing the processing and successful verification by the detection module, a RegEx match is generated as Figure 5(d) shows.

The RegEx detection is performed on the LaFA data structure that was constructed using the procedure described in the previous section. The algorithm for the detection procedure is shown in Algorithm 1.

Algorithm 1 Detection Procedure

```
1: LaFA (Input Event)
2: if (The node is active) then
3:     [Verify Detection Timing]
4:     for (Number of Lookup Nodes) do
5:         if (Buffered Lookup) then
6:             [Access one of special modules and verify]
7:         end if
8:         if (In-Line Lookup) then
9:             [Set RegEx detection pattern in an in-line module]
10:        end if
11:    end for
12:    if (All Lookup Matched) then
13:        if (Following Simple-String Node Exist) then
14:            [Activate next simple-string node]
15:        else
16:            [Generate match signal]
17:        end if
18:    end if
19: end if
```

As we see in the example, there are two simultaneous operations for S_2 and S_3. This slows down the detection speed if those operations are performed sequentially. It is important to note that LaFA avoids this drawback by duplicating the detection block and correlation block, and performing the detection operation in parallel. We call this block, consisting of one set of detection and correlation blocks, a *track*. For instance, we deploy two tracks in the example case, so detection operations of r_1 and r_4 are performed in separate tracks. As a result, LaFA can keep one operation (in parallel) per detection events. This lets us determine the required number of tracks per RegEx sets, which are discussed in Section 5.

3.3 Detection Block

The detection block consists of two main types of modules based on the detection approach in the LaFA architecture: *buffered lookup modules* and *in-line lookup modules*. This section describes each module in detail. The details of the architectures and design considerations for these modules are not covered in this paper due to space limitations.

As mentioned previously, RegEx set is partitioned into simple strings and variable strings. Simple strings are detected using simple-string detectors. Because of the wide variety of variable strings, we further classify them into eight sub-types as shown in Table 1. Each sub-type has a qualifier symbol next to the letter V as shown in the first column of Table 1. An example of each type is provided in the last column of the table. Each component is also assigned a unique ID, whereas components that are exactly the same are assigned the same ID even if they belong to different RegExes. Next, we discuss how these variable string sub-types can be detected.

Table 1: Sub-types of variable strings.

Symbol	component type	Example
VA	Character class (CC)	[a-z], \w
VB	Negated CC	[ˆa-z], [\W]
VC	Negated character	[ˆa], [ˆ\n]
VD	Character repetition	[\n]{3,10}
VE	Simple string repetition	abcd{3,10}
VF	CC repetition	[a-z]{3,10}, [\w]{3,10}
VG	Negated CC repetition	[ˆa-z]{3,10}, [\W]{3,10}
VH	Negated character repetition	[ˆa]{3,10}

3.3.1 *Buffered Lookup Modules*

The buffered lookup modules are core detection modules in the LaFA architecture. The lookahead operation can be realized by these modules. These modules use history stored in buffers to verify past activity. Next, we explain various detection modules based on the buffered lookup approach.

•Timestamps Lookup Module (TLM)

TLM stores the incoming character with respect to its time of arrival. This module can detect non-repetition types of variable strings such as (VA), (VB), and (VC). Let us use r_1 as an example to demonstrate the detection process. When detecting RegEx "*abc*[*a-z*]*op*", TLM is used to verify components[*a-z*]. Detection of a simple string "*op*" is a trigger to access TLM buffer memory and verify the character fetched for a particular time (one character before character "*o*" in this example) against the one received at the input. Thus, it is evident that TLM can detect a wide range of character classes such as [*a-z*] or [ˆ*x*] efficiently.

In a formal notation, let $T_{\mathrm{TLM}}=(T_1, T_2, T_3, \ldots T_l)$ be an ordered set of the TLM while l is the length of the character buffer. In the example, variable string[*a-z*] is located at time 4. Receiving character in time 4 is verified. The query is in a statement in the form of "$[a, z] \cap T_4 \neq \emptyset$" to check for its validity.

•Character Lookup Module (CLM)

CLM is responsible for the detection of (VH) types of variable strings. This is one type of variable string that drags traditional FAs down to impractical architectures. CLM stores the timestamps of each character in the buffer memory. CLM can remember more than one timestamp as per rule requirements with the provision of additional memory in the buffer. Let us pick lowercase "*x*" as an example and follow the example sequence of Figure 5. The lowercase "*x*" was detected at time *7*, and the timestamps were stored. This table is updated whenever a listed character is received. This history also tells us that no "*x*" was found between the time from 1 to 6 and after 8, which is the range of time. This characteristic is useful when detecting long repetitions of negative rule searches.

In a formal notation of the ASCII values, let C_{CLM}=\{C_0, C_1, $\ldots C_i$\} (i can be 0 to 255) be an unordered set. Some components can be empty ($C_x = \emptyset$). Each element C_x stores timestamps (*e.g.,* $C_{120} = \{t_6\}$; Components C_{120} correspond to lowercase "*x*"). The query is in a statement in the form of "$[t_4, t_6] \cap C_{120} = \emptyset$" to check for its validity.

•Short Simple-String Detector

As discussed in Section 2.1.2, short strings are classified as shallow components. Simple strings that are shorter than five characters are defined as short strings. Although exact string matching methods can be used to detect them, the benefits of doing so compared to using simpler detection methods diminish with the length of the string. Therefore, this type of component is detected the same way as TLM except that the short simple-string detector can query up to four characters.

3.3.2 In-Line Lookup Modules

•Repetition Detection Module (RDM)

RDM is responsible for detecting repetitions that are not detected by CLM (variable strings of type VD, VE, VF, and VG). More formally, RDM detects components in the form *base*{x,y} by accepting consecutive repetitions of the *base*, x to y times in the input. Here, base can be a single character, a character class, or a simple string. The {x,y} shows a range of repetition, where x is the minimum and y is the maximum repetitions.

The RDM module consists of sub-modules, each of which is capable of detecting the four types of repetitions. These RDM sub-modules operate *on demand.* Let a RegEx *r* detection be in progress that expects a repetition next. This expected repetition is programmed into an available RDM sub-module, PatternID, with the minimum and maximum repetition values. The sub-module inspects the input packet for the expected base and counts *consecutive* repetitions of this base.

When the input satisfies a repetition programmed on a RDM sub-module, RDM generates an in-line event, which causes the activation of the next node. If the sequence is broken, the detection process is immediately terminated and the sub-module becomes available for another request.

For fixed-range repetition (*e.g.,* [*a*-*z*]{5,10}), after the minimum consecutive detections (*e.g.,* 5), the next simple-string node is activated. However, when the consecutive count reaches the maximum count (*e.g.,* 10), the next simple-string node is deactivated and the RDM resources are released for use by other RegEx components. For at-least range repetition detection (*e.g.,* [*a*-*z*]{5,}), RDM uses the same operation as between-range repetition matching, except that node deactivation is not necessary.

•Frequently Used Repetition Detector

Observing that in practice, there are frequently used repetition bases (frequent bases), such as [*a*-*z*]. This creates an opportunity for sharing the detection effort for these frequent bases, further reducing resource usage. We propose to detect this kind of repetition using a similar method as CLM. A frequently used repetition detector stores the occurrence timestamp history of a given frequent base, the same way a CLM does. Given a range of timestamps $[t_i, t_j]$ and a PatternID, the detector can then validate whether such a repetition existed. Just as with CLM, this allows the detector to be shared among multiple RegExes by performing a buffered lookup operation instead of an in-line lookup operation.

4. DISCUSSIONS

4.1 On-line RegEx Update

An important property of LaFA is that it provides fast updates. When a new RegEx is available, the RegEx Detection System should be taken offline and updated with this new RegEx. This requires either deploying another system for backup or leaving the network open to attacks during the updates. The former solution is costly, whereas the latter is not secure. Thus, fast online updates are crucial.

LaFA can be updated very fast for two reasons: (1) The entire LaFA data structure is stored in the memory, thus any updates to the RegEx set can simply be done with memory updates and without hardware reconfiguration. (2) The detection of different RegExes is loosely coupled in the LaFA data structure, so the updates for one RegEx can be completed by updating only a fraction of the LaFA data structure. In contrast, updates to both DFA- and NFA-based methods are slow. In DFA-based methods, the data structures are very strongly coupled, so that an update in one RegEx may affect most of the RegExes in the set. Thus, any update has a potential to change most of the data structure, taking longer to complete. In NFA-based approaches, the data structures are hard-coded on the logic gates, thus any update requires a hardware reconfiguration, which also takes long. The LaFA architecture treats each RegEx independently. Thus, updating, deleting, or adding RegExes does not affect the other memory contents or the detection process of any other RegExes. The only places that need to be updated are the simple-string detector and a node memory that is related to the RegEx. In addition, offline node construction is performed locally. Thus, the update time of LaFA is significantly less than traditional FA schemes.

4.2 Detection Dependency

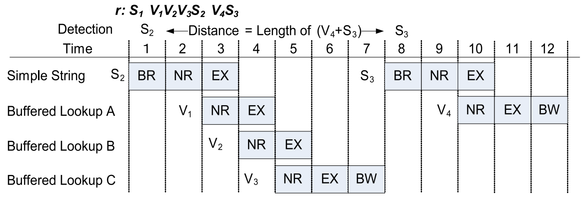


Figure 6: Timing Diagram of LaFA state machine and modules.

In our implementation, the state machine and the modules are pipelined. Figure 6 shows the Timing Diagram for the LaFA state machine and lookup modules. The state machine has four pipeline stages, namely (1) BR: Bitmap read, (2)NR: Node read, (3)EX: Execute (Timing Check), and (4)BW:Bitmap Write. The BR and NR stages fetch information from the active rule bitmap and the node information memories. Based on the information, the node verification is executed in the EX stage. Finally, in the BW stage, any updates to the bitmap are applied.

Depending on the RegEx that is being detected, the time dependencies of the correlation processing pipelines and module processing pipelines can change. For instance, consider the example RegEx $r : S_1V_1V_2V_3S_2V_4S_3$ in Figure 6. The detections of simple string ($S_1 - S_3$) are handled by the simple-string detector. Each successful detection of simple strings is followed by the corresponding correlation operations. To guarantee timely and orderly detection of the simple string and variable string , the detection of any variable string should be completed before the end of the next adjacent simple string. For instance, in this example, variable string $V_1 - V_3$ should be handled before S_3. The detection states of these three variable strings should be read from the modules and the state machine should be updated accordingly before the content S_3 is handled. This timing concept can be represented as follows: $[(Length\ of\ V_4\ and\ S_3) \geq (Detection\ processing\ time\ of\ S_2,\ V_1,\ V_2\ ,\ and\ V_3)]$. Note that in the LaFA prototype design on an FPGA, all detection modules take the same fixed processing time (*i.e.,* all modules require the same amount of clock cycle to complete the job). Remember that simple strings that are shorter than five characters are handled by the short simple-string detector. The worst case is that S_3 is the shortest simple string (5 characters), and V_4 is a repetition type of component with a minimum repetition of zero (0). Under this condition, we rewrite the RegEx accordingly. Define base V_4 as v (*i.e.,* $v\{x, y\}$). The RegEx will be rewritten to following three RegExes:

$$r_a : S_1V_1V_2V_3S_2S_3 \quad (1)$$

$$r_b : S_1V_1V_2V_3S_2vS_3 \quad (2)$$

$$r_c : S_1V_1V_2V_3S_2vvS_3 \quad (3)$$

$$r_d : S_1V_1V_2V_3S_2v\{3, y\}S_3 \quad (4)$$

5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our scheme for memory requirements and speed. Then, we compare its performance with existing schemes. The important aspect of LaFA is that it is purely based on RegEx sets and does not depend on the input string, thus the analysis here covers any traffic including worst-case. In other words, LaFA is secure against Denial-of-Service (DoS) attacks.

5.1 RegEx Data Sets

We selected multiple diverse RegEx sets from those RegExes that are widely used in the RegEx research. We also used rules that are used in [11] for comparison (Snort24, Snort31, and Snort34). From the Snort rules, we selected web-related rules that are extensive and very common in traffic over the Internet (web-client, web-misc, and web-php) and in spyware rules (spyware-put). SnortComb is a combination of Snort rules used to confirm the scalability of a RegEx set. As different sources, RegEx sets from Bro and Linux Layer 7 were also used. The version of each RegEx set is as follows: Snort - Version 2.6.0, L7 - update date 2008-04-23, and Bro - version 1.2.1.

5.2 System Design Consideration

We show simulation results of two main design considerations below.

i. Unrolling RegEx Sets

If a RegEx contains an alternation operator (|), it will be translated into a branch in an FA. We replace this RegEx with two RegExes, where each of the new RegExes contains one branch of the alternation. We call this operation *unrolling* due to its resemblance to the loop unrolling operation in Compiler. Although the number of RegExes increases after unrolling, the resulting FA after unrolling introduces new venues for optimization. In fact, it is possible to simplify the detection, reduce system complexity, and reduce resource usage by unrolling.

In general, if there are l alternations in a RegEx, where each alternation consists of a alternatives, this RegEx will expand to a^l RegExes. The results of unrolling are shown in table 2. It shows the number of original RegExes and unrolled RegExes for each RegEx sets. The last column shows the expansion ratio. For practical RegEx sets, a RegEx does not expand to more than 4.45 and most of them are less than 2 times. Even after unrolling, our final performance shows significant improvement in terms of memory requirements.

ii. Memory Load Balancing

A simple string may appear in multiple RegExes. Detection of the simple string that is enclosed in n rules requires n operations for a single track. As shown in section 3.2, we implement n tracks to avoid contention for the constant detection time. Table 3 shows the number of concurrent operations per simple-string detection for different RegEx sets. For example, in RegEx set "Snort24", 44 simple strings appear only once, 4 of them appear twice, and no simple string appears more than 2 times. Thus, the number of operations for this RegEx set is two. Even for large RegEx sets (SnortComb), a maximum up to nine operations per detection is required. In practice, the number of concurrent operations will be smaller than in the worst case because only active RegExes need to be processed. From the table, it is observed that the percentage of simple strings having one operation per detection is more than 80%. A very small portion of simple strings has a higher number of operation. This result shows that programming node information is trivial as each track takes care of one operation. This is because after spreading such high operation content among multiple tracks, the rest of them are almost free to go to any track. This helps to balance the size of node memories in each track. As a comparison, table 3 also shows the number of simultaneous active RegEx by NFA implementations. From the results, it is clear that the number of simultaneous operations can be easily handled in the LaFA architecture, while those in NFA are impossible.

5.3 Memory Requirements

This section discusses memory requirements and the scalability of LaFA. The clear separation of detection modules and state storage allows us to store required information in separate memories. To obtain the total system memory requirement, we analyze the memory requirements indi-

Table 2: Number of RegExes Before and After Unrolling.

RegEx Sets	Original	Unrolled	Expansion Ratio
Snort24	24	36	1.50
Snort31	31	69	2.23
Snort34	34	40	1.18
SnortMisc	49	147	3.00
SnortPHP	16	34	2.13
SnortWebCL	532	564	1.70
SnortSpy	630	668	1.06
Bro	827	841	1.02
LinuxL7	111	494	4.45

Table 3: Number of operations per simple-string detection.

RegEx	Number of Operations									Only One	Max. Op.	
Sets	1	2	3	4	5	6	7	8	9	Operation	LaFA	NFA
Snort24	44	4	0	0	0	0	0	0	0	91.67%	2	16
Snort31	55	5	0	1	0	0	0	0	0	90.16%	4	38
Snort34	44	6	1	0	0	0	0	0	0	86.27%	3	11
SnortMisc	24	10	1	3	0	1	0	0	0	61.54%	6	40
SnortPHP	34	0	0	0	0	0	0	0	0	100.00%	1	25
SnortWebCL	531	8	3	1	0	1	0	0	0	97.61%	6	313
SnortSpy	644	58	11	5	4	1	3	3	0	88.34%	8	390
SnortComb	1223	76	12	8	3	5	2	3	1	91.75%	9	423
Bro	547	57	4	0	0	0	0	0	0	89.97%	3	263
LimuxL7	148	17	9	4	1	0	0	0	0	82.68%	5	328

vidually. We start by analyzing the detection module, the timestamp history for simple strings, and the node memory requirement. This section also presents a summary of the system memory requirement sorted by RegEx sets and compares it with other schemes.

i. Detection Modules

The analysis of memory requirements for detection modules primarily depends on the detection approach *i.e.,* in-line lookup or buffered lookup. In this section, buffered lookup modules are analyzed. In-line lookup modules are summarized in table 6. The size of memory is carefully analyzed using the given RegEx sets. To determine the worst-case scenario, the analysis involves all possible combinations of the input string. The memory requirements are computed based on the results.

Here we analyze two special buffered lookup modules, CLM and TLM. These lookup modules have different architectures; hence, the memory requirements are analyzed accordingly. We begin the analysis with CLM. The CLM memory requirement is determined by the number of timestamps stored. To understand this, assume that CLM keeps only one timestamp entry for each character. Consider following the RegEx rule and input sequence (times are shown under the input character):

$$\begin{aligned} RegEx &: abc[\hat{}x]\{3\}xyz \\ Input &: a\ b\ c\ x\ x\ x\ x\ y\ z \\ Time &: 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \end{aligned}$$

This example illustrates a situation that requires more than one timestamp entries. Character "x" appears four times at times 4, 5, 6, and 7. CLM stores the timestamps. However, timestamps 4, 5, and 6 are overwritten by time 7 because only one timestamp entry is assigned. In this situation, RegEx element *[ˆx]{3}* cannot be verified correctly. At least two timestamp entries must be reserved in CLM for this example RegEx.[1] All RegExes are examined in this manner. Analysis of TLM is simpler than one for CLM. It takes the distance from a TLM component to a simple string that is a trigger to the TLM verification.

The results are shown in table 4. The second and third columns show the RegExes that are detected using CLM and the maximum timestamp history needed to be stored, respectively. The fourth column gives the corresponding memory requirements for the history buffer. The last column shows the number of repetition RegExes that are rewritten for in-line lookup. For RegExes that require larger or infinite buffers, we rewrite the RegEx component for in-line lookup for both CLM and TLM. The number of rewritten RegExes is shown in the table.

[1]It is not necessary to store four timestamps because to verify the RegEx element *[ˆx]{3}*, only one timestamp history is required.

Table 4: CLM and TLM Memory Requirements

RegEx Set	Maximum Buffer Size		Memory Size		Rewritten Components	
	CLM	TLM	CLM	TLM	CLM	TLM
Snort24	0	1	n/a	1kbits	0	0
Snort31	0	2	n/a	1kbits	0	0
Snort34	0	5	n/a	1kbits	0	0
SnortMisc	3	4	9kbits	1kbits	2	4
SnortPHP	1	1	3kbits	1kbits	0	1
SnortWebCL	2	33	6kbits	1kbits	3	0
SnortSpy	1	14	3kbits	1kbits	0	11
SnortComb	5	16	15kbits	1kbits	8	10
Bro	1	95	3kbits	1kbits	0	7
LinuxL7	0	41	n/a	1kbits	0	4

ii. Timestamp History for Simple Strings

Upon detecting a simple string, its time of detection is verified against that calculated using the information fetched from the node. The node provides the distance from the previous simple string in RegEx, which is used to calculate the actual timestamp of detection for the current simple string. This actual calculated timestamp is verified against the detected timestamp of the simple string. The detection process continues as explained in section 3.2 if, and only if, the timing verification for the simple string is successful.

To understand why we need to store the timestamp history for the simple string, let us reconsider RegEx r_4 with the input string as "$\underbrace{abc}_{S_2}\underbrace{abc}_{S_2}\underbrace{xyz}_{S_3}$". In this example, S_2 is detected twice and, hence, the timestamp entry of the first S_2 is overwritten by the timestamp entry of the second S_2. We cannot verify the timestamp of S_3 once S_3 is detected. Thus, even if the input belongs to the RegEx set, it cannot be detected successfully. To overcome this, we can store two timestamp histories for the simple string S_2. Table 5 gives the worst-case timestamp history storage requirements for simple strings.

iii. Node memory requirements

Now, we consider the memory requirements to store node information. A node contains different information types within a fixed width of 96 bits. Table 5 summarizes the total memory requirements to store node information for different RegEx rule sets. From the table, it is clear that information for the entire node can be stored in the on-chip memory of the current state-of-the-art FPGA.

iv. Memory Requirements for the System

We discussed the memory requirements of detection modules, timestamp history for simple strings, and nodes. Ta-

Table 5: Time and Node Memory Requirements (TS:Time Stamp, SS:Simple String)

RegEx Sets	Max TS per Char	# of SS	Time Mem.	# of Nodes	Node Mem.
Snort24	1	48	1kbits	110	11kbits
Snort31	2	61	2kbits	394	37kbits
Snort34	5	51	3kbits	144	14kbits
SnortMisc	31	39	15kbits	682	64kbits
SnortPHP	1	34	1kbits	73	7kbits
SnortWebCL	17	544	110kbits	1347	127kbits
SnortSpy	1	729	9kbits	2388	224kbits
SnortComb	31	1334	485kbits	4471	420kbits
Bro	13	606	93kbits	1635	154kbits
LinuxL7	29	180	61kbits	2914	274kbits

ble 6 summarizes the total memory requirements of the LaFA architecture. Detection modules are duplicated by the number of tracks and the results include duplication. Simple-string detection is not a main concern of this paper and we are not describing particular schemes. Since the simple-string detector is part of the LaFA architecture, estimated memory requirements of simple-string detector is also listed. This estimation is based on simple-string detection studies [5, 8, 24]. Now we combine all this information to determine the overall system memory requirements. Figure 7 compares memory requirements between our scheme and four other schemes in terms of the number of implemented rules (XFA [22,23], HybridFA [10], SplittingFA [17], CD^{2}FA [19]). We can see that the memory requirement of LaFA is an order of magnitude smaller than the other schemes. LaFA's memory requirements trend among different sizes of rules sets is shown as a line in Figure 7. The trend shows the memory requirements increase approximately linearly with the number of rules.

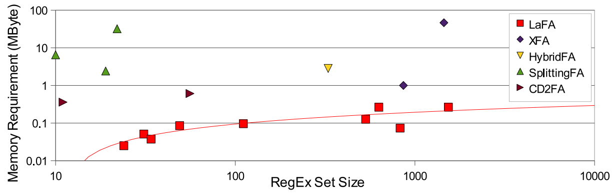


Figure 7: Scalability comparisons with other schemes.

5.4 Speed

To verify the feasibility of LaFA in terms of memory and speed, we implemented a prototype design on a Xilinx Virtex-4 xc4vfx100ff1152-11 FPGA, which has approximately 10 Mbits of Block RAM. The implementation consists of 3k RegEx components that are accommodated in one engine having three tracks. The logic utilization is 2100 slices (approx 5% of total available). The design achieved a clock frequency of 250 MHz, which corresponds to 2-Gbps throughput per engine. Based on the memory requirements presented in the table 6, it is clear that one LaFA engine can be duplicated on a single FPGA chip multiple times, which results in increased throughput. Thus for SnortComb requiring maximum memory (approximately 2.2 Mbits), 4 engines can fit on one Virtex-4 FPGA and throughput of 8 Gbps can be achieved. As memory is the dominant part of design, with more advanced FPGAs such as Virtex-6 having 38 Mbits of Block RAM and supporting 100-Gbps interface, 17 engines for SnortComb can fit on one FPGA resulting in 34-Gbps throughput. With other RegEx sets (such as Bro and Linux Layer 7) requiring much lesser memory, more engines can fit on one Virtex-6 FPGA and throughput in excess of 100 Gbps can be achieved.

6. RELATED WORK

Current state-of-the-art hardware architectures can reach higher speeds but they require prohibitively large memory, so they are either space-efficient (NFA) or high-speed (DFA), but not both.

Pure NFA implementations will be too slow in software, but implementations on hardware can be feasible with a high level of parallelism [24]. Their implementation uses solely logic gates. Although such schemes achieve high-speed RegEx detection, the inflexibility of implementing signatures on logic gates limits the updatability and scalability of NFA implementations.

The memory requirement of DFA implementation can be very high for certain types of RegExes. Some approaches [7, 9, 10, 12, 16–19, 21, 23, 26] aim to reduce the memory requirement by reducing the number of states. These approaches replace the DFA for these problematic RegExes with NFA or other architectures to minimize memory consumption, while using DFA to implement the rest of the RegExes [7-15]. HybridFA [10] keeps the problematic DFA states as NFAs (other parts use DFA). [17] proposes History-based Finite Automata (H-FA), which remembers the transition history to avoid creating unnecessary states, and History-based counting Finite Automata (H-cFA), which adds counters to reduce the number of states. The XFA proposed in [23] formalized and generalized the DFA state explosion problem and showed a reduction by using an idea similar to the H-FA . In [7], the author introduces a DFA-based FPGA solution. Multiple micro-controllers, each of which independently computes highly complex DFA operations, are used to avoid DFA state explosions. [21] is similar to [7] in that it compiles RegExes into simple operation codes so that multiple specifically designed micro-engines (micro-controller) work in parallel. However, even after replacing the problematic DFA states with more memory-efficient structures in the above schemes, the memory consumption of the rest of the DFA is still significant.

Other approaches are proposed to reduce DFA memory by reducing the number of transitions. For instance, D^{2}FA [18] merges multiple common transitions, called default transitions, to reduce the total number of transitions. However, this approach may require a large number of transitions for some cases, leading to an increase in the number of memory accesses per input byte. In addition, D^{2}FA construction is complex and requires a significant amount of resources. Several researchers follow up on the D^{2}FA idea [9, 19]. CD^{2}FA [19] and MergeDFA [9] resolve the problem of D^{2}FA by proposing multiple state transitions per character. MergeDFA bounds the number of worst-case transitions to $2m$, where m is the length of the input string. Although bounded, MergeDFA still requires a relatively large number of memory accesses. CD^{2}FA can achieve one transition per input character; however, it requires a perfect hash function to do so. Although, these schemes successfully address some issues in the D^{2}FA, they still cannot achieve satisfactory results for all three design objectives, namely, flexibility for adding new signatures, efficient resource usage, and high-

Table 6: Total Memory Requirements for ten RegEx sets. (SSD:Simple-String Detector)

	Snort24	Snort31	Snor34	SnortMisc	SnortPHP	SnortWCL	SnortSpy	SnortComb	Bro	Limux7
RegEx Set Size	24	31	34	49	16	532	630	1537	827	111
# of Tracks	2	4	3	6	1	6	8	9	3	5
CLM	n/a	n/a	n/a	54kbits	3kbits	36kbits	24kbits	135kbits	9kbits	n/a
TLM	2kbits	4kbits	3kbits	6kbits	1kbits	6kbits	8kbits	9kbits	3kbits	5kbits
RDM	88kbits	176kbits	132kbits	264kbits	44kbits	264kbits	352kbits	392kbits	132kbits	184kbits
Short String	96kbits	192kbits	144kbits	288kbits	48kbits	288kbits	384kbits	432kbits	144kbits	240kbits
Frequent	2kbits	3kbits	2kbits	4kbits	1kbits	4kbits	5kbits	6kbits	2kbits	4kbits
Nodes	11kbits	37kbits	14kbits	64kbits	7kbits	128kbits	225kbits	422kbits	155kbits	275kbits
Timestamps	1kbits	2kbits	3kbits	15kbits	1kbits	110kbits	9kbits	485kbits	93kbits	61kbits
Sub Total	201kbits	415kbits	299kbits	696kbits	106kbits	837kbits	1008kbits	1883kbits	539kbits	770kbits
SSD	6kbits	4kbits	8kbits	3kbits	4kbits	198kbits	78kbits	281kbits	65kbits	18kbits
Total	207kbits	419kbits	307kbits	699kbits	110kbits	1035kbits	1086kbits	2164kbits	604kbits	788kbits

speed detection.

In [26], the authors focused on optimization at the RegEx level, before the FA is generated. They proposed rule rewriting for particular RegEx patterns that cause state explosions. They also suggested grouping (splitting) the DFA into multiple groups to reduce the number of states.

7. CONCLUSION

In this paper, we propose LaFA, an on-chip RegEx detection system that is highly scalable. The scalability of existing schemes is generally limited by the traditional per character state processing and state transition detection paradigm. The main focus of existing schemes is in optimizing the number of states and the required transitions, but not the suboptimal character-based detection method. Furthermore, the potential benefits of the reduced number of operations and states using out-of-sequence detection method have not been explored. We propose to clearly separate detection operations from state transitions. This opens up the opportunities to further optimize traditional FAs. LaFA employs a novel lookahead technique to reorder the sequence of pattern detections. We have a higher probability of declaring a mismatch before evaluating complex patterns and requiring fewer operations. Independent detection modules that are hardware compatible are introduced to replace variable strings (*i.e.,* complex patterns and shallow patterns). This solves the scalability problem of traditional FAs and greatly increases the memory efficiency of LaFA. Compared to today's state-of-the-art RegEx detection system, LaFA requires an order of magnitude less memory. A single commercial FPGA chip can accommodate up to twenty-five thousand (25k) RegExes. Based on the throughput of our LaFA prototype on FPGA, we estimate that when used with a Snort RegEx signature set, a 34-Gbps throughput can be achieved. We believe that LaFA is a highly competitive, scalable, and hardware-feasible solution to the RegEx detection problem.

8. REFERENCES

[1] Bro intrusion detection system. *http://www.bro-ids.org.*

[2] Snort network intrusion detection system. *http://www.snort.org.*

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, 2006.

[4] A. N. Arslan. Multiple Sequence Alignment Containing a Sequence of Regular Expressions. In *CIBCB*, pages 1–7, 2005.

[5] N. S. Artan, M. Bando, and H. J. Chao. Boundary hash for memory-efficient deep packet inspection. In *ICC*, pages 1732–1737, 2008.

[6] N. S. Artan and H. J. Chao. TriBiCa: Trie bitmap content analyzer for high-speed network intrusion detection. In *INFOCOM*, pages 125–133, 2007.

[7] Z. K. Baker, H.-J. Jung, and V. K. Prasanna. Regular expression software deceleration for intrusion detection systems. In *FPL*, pages 1–8, 2006.

[8] M. Bando, N. S. Artan, and H. J. Chao. Highly memory-efficient LogLog Hash for deep packet inspection. In *GLOBECOM*, pages 1–6, 2008.

[9] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *INFOCOM*, pages 1064–1072, 2007.

[10] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *CoNEXT*, 2007.

[11] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *ANCS*, pages 145–154, 2007.

[12] M. Becchi and P. Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *CoNEXT*, 2008.

[13] Y. H. Cho and W. H. Mangione-Smith. Deep network packet filter design for reconfigurable devices. *Trans. on Embedded Computing Sys.*, 7(2):1–26, 2008.

[14] Y. S. Chung, W. H. Lee, C. Y. Tang, and C. L. Lu. RE-MuSiC: a tool for multiple sequence alignment with regular expression constraints. *Nucleic Acids Res. (Web Server issue)*, (35):W639–644, 2007.

[15] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel bloom filters. *Micro*, 24(1):52–61, 2004.

[16] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro. An improved DFA for fast regular expression matching. *SIGCOMM Comput. Commun. Rev.*, 38(5):29–40, 2008.

[17] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS*, pages 155–164, 2007.

[18] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM*, pages 339–350, 2006.

[19] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *ANCS*, pages 81–92, 2006.

[20] J. Levandoski, E. Sommer, and M.Strait. Application layer packet classifer for linux. *http://l7-filter.sourceforge.net.*

[21] M. Paolieri, I. Bonesana, and M. D. Santambrogio. ReCPU: A parallel and pipelined architecture for regular expression matching. In *IFIP, VLSI - SoC.*, pages 19–24, 2007.

[22] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *SP*, pages 187–201, 2008.

[23] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM*, 2008.

[24] I. Sourdis, D. Pnevmatikatos, and S. Vassiliadis. Scalable multigigabit pattern matching for packet inspection. In *VLSI*, volume 16, pages 156–166, 2008.

[25] N. Wirth. *Compiler Construction.* Addison Wesley, 1996.

[26] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS*, pages 93–102, 2006.

[27] F. Yu, R. Katz, and T. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *ICNP*, pages 174–183, 2004.