Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia

Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner
Washington University

George Varghese
University of California, San Diego

ABSTRACT
The importance of network security has grown tremendously and a collection of devices have been introduced, which can improve the security of a network. Network intrusion detection systems (NIDS) are among the most widely deployed such system; popular NIDS use a collection of signatures of known security threats and viruses, which are used to scan each packet's payload. Today, signatures are often specified as regular expressions; thus the core of the NIDS comprises of a regular expressions parser; such parsers are traditionally implemented as finite automata. Deterministic Finite Automata (DFA) are fast, therefore they are often desirable at high network link rates. DFA for the signatures, which are used in the current security devices, however require prohibitive amounts of memory, which limits their practical use.

In this paper, we argue that the traditional DFA based NIDS has three main limitations: first they fail to exploit the fact that normal data streams rarely match any virus signature; second, DFAs are extremely inefficient in following multiple partially matching signatures and explodes in size, and third, finite automaton are incapable of efficiently keeping track of counts. We propose mechanisms to solve each of these drawbacks and demonstrate that our solutions can implement a NIDS much more securely and economically, and at the same time substantially improve the packet throughput.

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

General Terms
Algorithms, Design, Security.

Keywords
DFA, regular expressions, deep packet inspection.

1. INTRODUCTION
Network security has recently received an enormous attention due to the mounting security concerns in today's networks. A wide variety of algorithms have been proposed which can detect and combat with these security threats. Among all these proposals, signature based Network Intrusion Detection Systems (NIDS) have become a commercial success and have seen a widespread adoption. A signature based NIDS maintains signatures, which characterizes the profile of known security threats (*e.g.* a virus, or a DoS attack). These signatures are used to parse the data streams of various flows traversing through the network link; when a flow matches a signature, appropriate action is taken (*e.g.* block the flow or rate limit it). Traditionally, security signatures have been specified as string based exact match, however regular expressions are now replacing them due to their superior expressive power and flexibility.

When regular expressions are used to specify the signatures in a NIDS, then finite automaton are typically employed to implement them. There are two types of finite automaton: Nondeterministic Finite Automaton (NFA) and Deterministic Finite Automaton (DFA) [2]. Unlike NFA, DFA requires only one state traversal per character thereby yielding higher parsing rates. Additionally, DFA maintains a single state of execution which reduces the "per flow" parse state maintained due to the packet multiplexing in network links. Consequently, DFA is the preferred method.

DFAs are fast, however for the current sets of regular expressions, they require prohibitive amounts of memory. Current solutions often divide a signature set into multiple subsets, and construct a DFA for each of them. However, multiple DFAs require multiple state traversals which reduce the throughput, and increase the "per flow" parse state. Large "per flow" parse state may also create a performance bottleneck because they may have be loaded and stored for every packet due to the packet multiplexing.

The problems associated with the traditional DFA based regular expressions stems from three prime factors. First, they take no interest in exploiting the fact that normal data streams rarely match more than first few symbols of any signature. In such situations, if one constructs a DFA for the entire signatures, then most portions of the DFA will be unvisited, thus the approach of keeping the entire automaton active appears wasteful; we call this deficiency *insomnia*. Second, a DFA usually maintains a single state of execution, due to which it is unable to efficiently follow the progress of multiple partial matches. They employ a separate state for each such combination of partial match, thus the number of states can explode combinatorially. It appears that if one equips an automaton with a small auxiliary memory which it will use to register the events of partial matches, then a combinatorial explosion can be avoided; we refer to this implement a 32-bit counter. We call this deficiency *acalculia*.

In this paper, we propose solutions to tackle each of these three drawbacks. We propose mechanisms to split signatures such that, only one portion needs to remain active, while the remaining portions can be put to sleep under normal conditions. We also propose a cure to amnesia, by introducing a new machine, which is as fast as DFA, but requires much fewer number of states. Our final cure to acalculia extends this machine, so that it can handle events of counting much more efficiently.

The remainder of the paper is organized as follows. Due to space limitation, background is presented in the technical report [35]. Section 2 explains the drawbacks of traditional implementations. Our cure to insomnia is presented in Section 3. Section 4 presents the cure to amnesia, and section 5 presents the cure to acalculia. Section 6 reports the results, and the paper concludes in Section 7.

2. Regular Expressions in Networking

Any implementation of regular expressions in networking has to deal with several complications. The first complication arises due to multiplexing of packets in the network links. Since packets belonging to different flows can arrive interspersed with each other, any pattern matcher has to de-multiplex these packets and reassemble the data stream of various flows before parsing them. As a consequence, the architecture must maintain the parse state after parsing any packet. Upon a switch from a flow x to a flow y, the machine will first store the parse state of the current flow x and load the parse state of the last packet of the flow y.

Consequently, it is critical to limit the parse state associated with the pattern matcher because at high speed backbone links, the number of flows can reach up to a million. NFAs are therefore not desirable in spite of being compact, because they can have a large number of active states. On the other hand, DFA requires a single active state; thus the amount of parse state remains small.

The second complication arises due to the high network link rates. In a 10 Gbps network link, a payload byte usually arrives every nano-second. Thus, a parser running at 1GHz clock rate will have a single clock cycle to process each input byte. NFAs are unlikely to maintain such parsing speeds because they often require multiple state traversals for an input byte; thus DFAs appear to be the only resort. Due to these complications, one can conclude that a pattern matching machine for networking applications must satisfy these dual objectives *i*) fast parsing rates or few transitions per input byte, and *ii*) less "per flow" state.

Although, DFAs appear to meet both of these goals, they often suffer from state explosion, *i.e.* the total number of states in a DFA can be exponential in the length of the regular expression. The problems with a DFA based approach can be divided into the following three main categories.

2.1 Three Key Problems of Finite Automata

In this section, we introduce the three deficiencies of traditional finite automata based regular expressions approach:

1. Traditional regular expressions implementations often employ the complete signatures to parse the input data. However, in NIDS applications, the likelihood that a normal data stream completely matches a signature is low. Traditional approach therefore appears wasteful; rather, the tail portions of the signatures can be isolated from the automaton, and put to sleep during normal traffic and woken up only when they are needed. We call this inability of the traditional approach *Insomnia*. The number of states in a machine suffering from insomnia may unnecessarily bloat up; the problem becomes more severe when the tail portion is relatively complex and long. We present an effective cure to insomnia in section 3.

2. The second deficiency, which is specific to DFAs, can be classified as *Amnesia*. In amnesia, a DFA has limited memory; thus it only remembers a single state of parsing and ignores everything about the earlier parse and the associated partial matches. Due to this tendency, DFAs may require a large number of states to track the progress of both the current match as well as any previous partial match. Although amnesia keeps the per flow state required during the parse small, it often causes an explosion in the number of states, because a separate state is required to indicate every possible combination of partial match. Intuitively, a machine which has a few flags in addition to its current state of execution can utilize these flags to track multiple matches more efficiently and avoid state explosions. We propose such a machine in section 4, which efficiently cures DFAs from amnesia.

3. The third deficiency of the finite automata can be tagged with the label *Acalculia* due to which it (both NFA and DFA) is unable to efficiently count the occurrences of certain sub-expressions in the input stream. Thus, whenever a regular expression contains a length restriction of k on a sub-expression, the number of states required by the sub-expression gets multiplied by k. With length restrictions, the number of states in a NFA increases linearly, while in a DFA, it may increase exponentially. It is desirable to construct a machine which is capable of counting certain events, and uses this capability to avoid the state explosion. We propose such machines in section 5. We now proceed with our cures to these three deficiencies. Our first solution is cure from insomnia.

3. Curing DFA from Insomnia

Traditional approach of pattern matching constructs an automaton for the entire regular expression (reg-ex) signature, which is used to parse the input data. However, in NIDS applications, normal flows rarely match more than first few symbols of any signature. Thus, the traditional approach appears wasteful; the automaton unnecessarily bloats up in size as it attempts to represent the entire signature even though the tail portions are rarely visited. Rather, the tail portions can be isolated from the automaton, and put to sleep during normal traffic conditions and woken up only when they are needed. Since the traditional approach is unable to perform such selective sleeping and keeps the automaton awake for the entire signature, we call this deficiency *insomnia*.

In other words, insomnia can be viewed as the inability of the traditional pattern matchers to isolate frequently visited portions of a signature from the infrequent ones. Insomnia is dangerous due to two reasons *i*) the infrequently visited tail portions of the reg-exes are generally complex (contains closures, unions, length restrictions) and long (more than 80% of the signature), and *ii*) the size of fast representations of reg-exes (*e.g.* DFA) usually are exponential in the length and complexity of an expression. Thus, without a cure from insomnia, a DFA of hundreds of reg-exes may become infeasible or will require enormous amounts of memory.

An obvious cure to insomnia will essentially require an isolation of the frequently visited portions of the signatures from the infrequent ones. Clearly, frequently visited portions must be implemented with a fast representation like a DFA and stored in a fast memory in order to maintain high parsing rates. Moreover, since fast memories are less dense and limited in size, and fast representations like DFA usually suffer from state blowup, it is vital to keep such fast representations compact and simple. Fortunately, practical signatures can be cleanly split into simple prefixes and suffixes, such that the prefixes comprise of the entire frequently visited portions of the signature. Therefore, with such a clean separation in place, only the automaton representing the prefixes need to remain

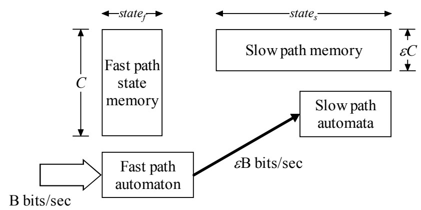


Figure 1: Fast path and slow path processing in a bifurcated packet processing architecture.

active at all times; thereby, curing the traditional approach from insomnia by keeping the suffix automaton in a sleep state most of the times.

There is an important tradeoff involved in such a prefix and suffix based pattern matching architecture. The general objective is to keep the prefixes small, so that the automaton which is awake all the time remains compact and fast. At the same time, if prefixes are too small then normal data streams will match them often, thereby waking up the suffixes more frequently than desired. Note that, during abnormal conditions the automaton representing the suffixes will be triggered more often; however, we discuss such scenarios later. Under normal conditions, the architecture must therefore balance the tradeoff between the simplicity of the fast automaton and the dormancy of the slow automaton.

We refer to the automaton which represents the prefixes as the *fast path* and the remaining as the *slow path*. Fast path remains awake for the entire input data stream, and activates the slow path once it finds a matching prefix. There are two expectations. First, slow path should be triggered rarely. Second, it should process a fraction of the input data; hence it can use a slow memory and a compact representation like a NFA, even if it is relatively slow. In order to meet these expectations, normal data streams must not match the prefixes of the signatures or match them rarely. Upon a prefix match, the slow path processing should not continue for a long time. The likelihood that these two expectations will be met during normal traffic conditions will depend directly upon the signatures and the positions where they are split into prefixes and suffixes. Thus, it is critical to decide the split positions and we describe our procedure to compute these in the next section.

3.1 Splitting the regular expressions

The dual objectives of the splitting procedure are that the prefixes remain as small as possible, and at the same time, the likelihood that normal data matches these prefixes is low. The probability of matching a prefix depends upon its length and the distribution of various symbols in the input data. In this context, it may not be acceptable to assume a uniform random distribution of the input symbols (*i.e.* every symbol appears with a probability of 1/256) because some words appear much more often than the others (*e.g.* "HELO" in an ICMP packet). Therefore, one needs to consider a *trace driven probability distribution* of various input symbols [6]. With these traces, one can compute the matching probability of prefixes of different lengths under normal and anomalous traffic. This will determine the rate at which slow path will be triggered.

In addition to the "matching probabilities", it is important to consider the probabilities of making transitions between any two states of the automaton. This probability will determine how long the slow path will continue processing once it is triggered. These transition probabilities are likely to be dependent upon the previous stream of input symbols, because there is a strong correlation between the occurrences of various symbols, *i.e.* when and where they occur with respect to each other. The transition probabilities as well as the matching probabilities can be assigned by constructing an NFA of the regular expressions signatures and parsing the same against normal and anomalous traffic.

More systematically, given the NFA of each regular expression, we determine the probability with which each state of the NFA becomes active and the probability that the NFA takes its different transitions. Once these probabilities are computed, we determine a cut in the NFA graph, so that *i*) there are as few nodes as possible on the left hand side of the cut, and *ii*) the probability that states on the right hand side of the cut is active is sufficiently small. This will ensure that the fast path remains compact and the slow path is triggered only occasionally. While determining the cut, we also need to ensure that the probability of those transitions which leaves some NFA node on the right hand side and enters some other node on the same side of the cut remains small. This will ensure that, once the slow path is triggered, it will stop after processing a few input symbols. Clearly, the cut computed from the normal traffic traces and from the attack traffic are likely to be different, thus the corresponding prefixes will also be different. We adopt the policy of taking the longer prefix. More details of the cutting algorithm are present in the technical report [35].

3.2 The bifurcated pattern matching

We now present the bifurcated pattern matching architecture. The architecture (shown in Figure 1) consists of two components: fast path and slow path. The fast path parses every byte of each flow and matches them against the prefixes of all reg-exes. The slow path parses only those flows which have found a match in the fast path, and matches them only against the corresponding suffixes.

Notice that, the parsing of input data is performed on a per flow basis. In order to keep parsing of each flow discrete, the "per flow parse state" has to be stored. With millions of active flows, parse states have to be stored in an off-chip memory, which may create a performance bottleneck because upon any flow switch we will have to store and load this information. With the minimum IP packet size being 40 bytes, we may have to perform this load and store operation every 40 ns at 10 Gbps rates. Thus, it is important to minimize the "per flow parse states". This minimization is critical in the fast path because all flows are processed by the fast path. It does not pose a similar threat to the slow path because it processes a fraction of the payload of a small number of flows.

Consequently, the fast path automaton has two objectives: 1) it must require small per flow parse state, and 2) it must be able to perform parsing at high speed, in order to meet the link rates. One obvious solution which will satisfy this dual objective is to construct a single composite DFA of all prefixes. A composite DFA will have only one active state per flow and will also require only one state traversal for an input character. Thus, if there are C flows in total, we will need $C \times state_f$ memory, where $state_f$ is the bits needed to represent a DFA state. At this point in discussion we will proceed with a composite DFA in the fast path, later in section 4, we will propose an alternative to a composite DFA which is more space efficient and yet satisfies our dual objectives.

Slow path on the other hand handles, say ε fraction of the total number of bytes processed by the fast path. Therefore, it will need to store the parse state of εC flows on an average. If we keep ε small, then unlike the fast path, we neither have to worry about minimizing the "per flow parse state" nor do we have to use a fast representation, to keep up with the link rates. Thus, a NFA may

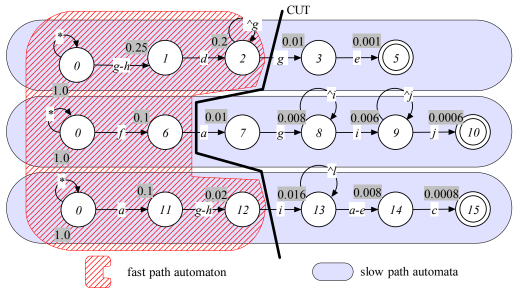


Figure 2: NFA and the cut between prefix and suffix

suffice to represent the slow path. Nevertheless slow path offers another key advantage, *i.e.* a composite automaton for all suffixes is not required because we need to parse the flows against only those suffixes whose prefixes have been matched.

However, there is a complication in the slow path. Slow path can be triggered multiple times for the same flow, thus there can be multiple instances of per flow active parse states even though we may be using a DFA. Consider a simple example of an expression `ababcda`, which is split into `ab` prefix and `abcda` suffix, and a packet payload `"xyababcdpq"`. The slow path will be triggered twice by this packet, and there will be two instances of active parse states in the slow path. In general it is possible that *i*) a single packet triggers the slow path several times, in which case signaling between the fast and slow path may become a bottleneck and *ii*) there are multiple active states in the slow path, which will require complicated data-structures to store the parse states.

These problems will exacerbate when the slow path will process packets much slower than the fast path and will handle its triggers sequentially. With the above packet, slow path will be triggered first after the fast path parses `"xyababcdpq"` and second after `"xyababcdpq"`. Upon first trigger, it will parse the payload `"xyababcdpq"` and stop after it sees `p`. Upon second trigger, it will parse the payload `"xyababcdpq"`, thus repeating the previous parse. Due to these complications, we propose a packetized version of bifurcated packet processing architecture.

3.3 Packetized bifurcated pattern matching

The objective of the packetized bifurcated packet processing is to minimize the signaling between the fast path and the slow path. More specifically if we ensure that the fast path triggers the slow path at most once for every packet, then the slow path will not repeat the parsing of the same packet payload. This objective can be satisfied by slightly modifying the slow path automaton, so that it parses the packets against the entire signature, and not just the suffixes. With the slow path representing the entire signature, the subsequent triggers for this signature will be captured within the slow path. Hence, they can be ignored.

In order to better understand how the slow path is constructed and triggered, let us consider a simple example of 3 signatures:

$r_1 = \texttt{.}^*\texttt{[gh]d[\^{}g]}^*\texttt{ge}$

$r_2 = \texttt{.}^*\texttt{fag[\^{}i]}^*\texttt{i[\^{}j]}^*\texttt{j}$

$r_3 = \texttt{.}^*\texttt{a[gh]i[\^{}l]}^*\texttt{[ae]c}$

The NFA for these signatures are shown in figure 2 (a composite DFA for these signatures will contain 92 states). In the figure, the probabilities with which various NFA states are activated are also highlighted. A cut between the fast and slow path is also shown which divides the states so that the cumulative probability of the slow path states is less than 5%.

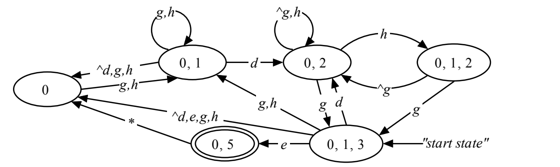


Figure 3: DFA and start state for r_1 in the slow path

With this cut, the prefixes will be $p_1 = \texttt{[gh]d[\^{}g]}^*\texttt{g}$; $p_2 = \texttt{f}$; and $p_3 = \texttt{j[gh]}$ and the corresponding suffixes will be $s_1 = \texttt{e}$; $s_2 = \texttt{ag[\^{}i]}^*\texttt{i[\^{}j]}^*\texttt{j}$; and $s_3 = \texttt{i[\^{}l]}^*\texttt{[ae]c}$. As highlighted in the same figure, fast path consists of a composite DFA of the three prefixes p_1, p_2, and p_3, which will have only 14 states, while the slow path comprises of three separate DFAs, one for each signature r_1, r_2, and r_3, rather than just the suffixes s_1, s_2, and s_3.

Whenever the fast path will find a matching prefix, say p_i in a packet, it will trigger the corresponding slow path automaton representing the signature r_i. Once this automaton is triggered, all subsequent triggers corresponding to the prefix p_i for the signature r_i can be ignored because during the process of matching r_i in the slow path, such triggers will also be detected. Thus, for any given packet processed in the fast path, the state of the slow path "active or asleep" associated with each signature is maintained, so that the subsequent triggers for any given signature can be masked out.

However, we have to be careful in initiating the of triggering the slow path automaton representing any signature r_i. Specifically, we have to ensure that the slow path automaton begins at a state which indicates that the prefix p_i of the signature r_i has already been detected. Consider the DFA for the first signature (r_1) of the above slow path, shown in Figure 3. Instead of beginning at the usual start state, 0 of this DFA, we begin its parsing at the state (0,1,3), which indicates that the prefix p_1 has just been detected; the parsing continues from this point onwards in the slow path.

In general case, the start state of the slow path automaton will depend upon the fast path DFA state which triggers the slow path. More specifically, the slow path start state will be the minimal one which encompasses all partial matches in the fast path.

The above procedure describes how we initiate the slow path automaton for a prefix match in any given packet. The decision that the slow path should remain active for the subsequent packets of the flow depends on the state of the slow path automaton at which the packet leaves it. If this final DFA state comprises any of the states of the slow path NFA, then the implication is that the slow path processing will continue; else the slow path will be put to sleep. For example, in the Figure 3, unless the final state upon a packet parsing is either (0,1,3) or (0,5), the subsequent packets of the flow will not be parsed by this automaton; in other words this automaton will no longer remain active.

Let us now consider the parsing of a packet payload `"gdgdgh"`. The fast path state traversal is illustrated below; the slow path will be triggered twice, but the second trigger will be ignored.

$$0 \xrightarrow{g} 0,1 \xrightarrow{d} 0,2 \xrightarrow{g} \underset{\substack{\uparrow\\ r_1}}{0,1,3} \xrightarrow{d} 0,2 \xrightarrow{g} \underset{\substack{\uparrow\\ r_1}}{0,1,3} \xrightarrow{h} 0,1$$

Upon the first trigger, the slow path DFA (shown in Figure 3) for the signature r_1 will begin its execution at the state (0,1,3) and will parse the remaining packet payload `"dgh"`. The parsing will finish at the DFA state (0, 1). Since this state does not contain any of the states of the slow path NFA, this slow path automaton will be put to

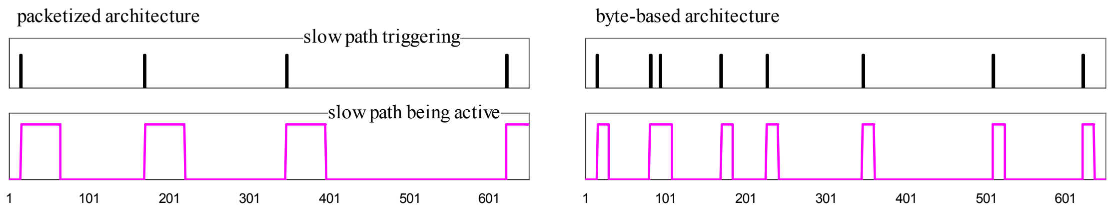


Figure 4: Fast path and slow path processing in a bifurcated packet and byte based processing architectures.

sleep. On the other hand if the remaining packet payload were "dge", the packet would leave the slow path in the state (0,5). Thus, in this case, the slow path processing will remain active for the subsequent packets of the flow.

In contrast with the previous byte based pattern matching architecture, the proposed packetized architecture has a drawback that it keeps the slow path automaton active until the packet is completely parsed in the slow path. Thus, the slow path may end up processing many more bytes, unlike in the byte level architecture. This drawback arises due to the difference in the processing granularity; the byte based pattern matcher will halt the slow path as soon as the next input character leads to a suffix mismatch, whereas the packetized pattern matcher will retain the slow path active till the last byte of the packet is parsed. Nevertheless, the packetized architecture maintains the triggering probability at a much lower value, since the recurrent signaling of prefixes belonging to the same signature is suppressed.

Let us experimentally evaluate the performance of the packetized pattern matching architecture against the byte level architecture. Both architectures are likely to operate well when the input traffic is benign and the slow path is triggered with very low probability, say 0.01%. Therefore, we consider an extreme situation where the 1% of the contents of the input data stream consists of the entire signatures. Thus, the triggering probability of the slow path will be around 1%. We use 36 Cisco signatures whose average length is 33 characters, and assume that packets are 200 bytes long. In Figure 4, we plot a snapshot of the timeline of the triggering events, and the time intervals during which the slow path is active. It is apparent that slow path in the packetized architecture remains active for relatively longer durations. Consequently, the signatures have to be split accordingly in the packetized architecture, so that the slow path will handle such loads.

3.4 Protection against DoS attacks

In bifurcated packet processing architecture, a small fraction of packets from the normal flows might be diverted to the slow path, even though a normal data stream is not likely to match any signature. The slow path processing is provisioned in a way that it can sustain the rate at which such false packet diversions from normal flows occur. Therefore, it is unlikely, that these packets from normal flows will overload the slow path. However, a flow which frequently matches prefixes, may overload the slow path by triggering it more often than desired. This opens up a possibility of a Denial of service attack.

A denial of service attack, in fact is much more threatening to the end-to-end data transfer. Consider a packet from a normal flow getting diverted to the slow path. If the slow path is overloaded, then this packet will either get discarded or encounter enormous processing delays. If the sending application retransmits this packet, it will further exacerbate the overload condition in the slow path. The implication on the end-to-end data transfer is that it may never be able to deliver this packet, and complete the data transmission. This clearly signals a need to protect these normal flows from such repeated packet discards. To accomplish this objective, we need some mechanism in the slow path to distinguish such packets of normal flows from the packets of the anomalous or attack flows, which are overloading the slow path. We now propose a lightweight algorithm which performs such classification at very high speed and with high accuracy.

Our algorithm is based upon statistical sampling of packets from each flow. For each flow, we compute an *anomaly index* which is a "moving average" of the number of its packets which matches one of the prefixes in the fast path. The moving average can either be a "simple moving average (SMA)" or an "exponential moving average (EMA)". For simplicity we only consider the SMA, wherein we compute the average number of packets which matches some prefix over a window of n previous packets. We call a flow well-behaving, if less than ε fraction of its packets finds a match, simply because such a flow will not overload the slow path. Flows which find more matches are referred to as anomalous. If the sampling window n is sufficiently large, then the anomaly indices of the well-behaving flows are expected to be much smaller than those of the anomalous/attack flows. However, longer sampling windows will require more bits per flow to compute the anomaly index. Consequently there is a trade-off between the accuracy of the anomaly indices and the "per flow" memory needed to maintain them. We attempt to strike a balance between this accuracy and the cost of implementation.

Let us say that we are given with at most k-bits for every flow to represent its anomaly index. Since a flow is declared anomalous as soon as its anomaly index exceeds ε, we set ε as the upper bound of the anomaly index. Thus, when all k-bits are set, it represents an anomaly index of ε. Consequently, the per flow sampling window, n comprises of $2^k/\varepsilon$ packets; for every packet which matches a prefix, the k-bit counter is incremented by $1/\varepsilon$ and for other packets it is decremented by 1 (note that a flow is a threat only if more than ε fraction of its packets are diverted to the slow path, or the mean distance between packets which are diverted is smaller than $1/\varepsilon$ packets). Thus, the probability that a flow which indeed is anomalous is not detected will be $O(e^{-n})$. If ε is 0.01, then 8-bit anomaly counter will result in a false detection probability of well below 10^{-6}. This analysis assumes that the events of packet diverts to the slow path is uniformly distributed. In case of any other distribution, the accuracy of the detection of anomalous flows is likely to improve while the probability that a normal flow is falsely detected as anomalous may also increase.

The anomaly counters in fact, indicates the degree to which a flow loads the slow path. Consequently, they can be used to classify not just the anomalous flows but also the well behaving flows. The flows can be prioritized in the slow path according to the degree of their anomaly; the implication being that the slow path will first process the flows with smaller anomaly indices. The slow path thus

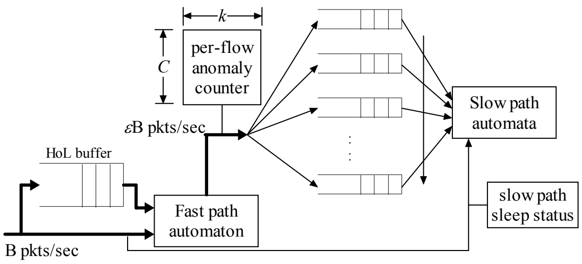


Figure 5: Fast path and slow path processing in a bifurcated packet processing architecture.

consists of multiple queues which will store the requests from various flows according to their anomaly indices. Queues associated with smaller anomaly indices are serviced with higher priority. Hence, even if a well behaving flow accidentally diverts its packets to the slow path, it will be serviced quickly in spite of the presence of large volumes of anomalous packets.

3.5 Binding things together

Having described the procedure to split the reg-ex signatures into simple prefixes and relatively complex suffixes as well as mechanisms needed to put the suffix portions to sleep, we are now ready to discuss some further issues. In these pattern matching architectures, the first issue is that it often becomes critical to prevent a receiver from receiving a complete signature. This has an interesting implication. Whenever a packet is diverted to the slow path, no subsequent packets of the same flow can be forwarded in the fast path, until the slow path packet is completely processed. If this policy is not adhered to, then signatures that span across multiple packets might not be detected. This indicates that in any flow, if a packet is accidentally diverted to the slow path, subsequent packets of the flow can create a head of line (HoL) blocking in the fast path. Thus, in order to avoid such HoL blockings, a HoL buffer is maintained (shown in Figure 5), which stores the packets that can not be processed currently.

The above discussion again bolsters the premise that the normal flows must be guarded against anomalous/attack flows which may overload the slow path. Without such protection, whenever a diverted packet of a normal flow gets either delayed or discarded in the heavily loaded slow path, subsequent packets of the flow cannot be forwarded; thus the flow will essentially become dead. In case of TCP, the discarded packet will get retransmitted after the time-out; nevertheless, it will again get diverted to the slow path, and congestion will ensue. Since DoS protection is crucial, we have performed a thorough evaluation of DoS protection, and the results are summarized in the technical report [35]

4. H-FA: Curing DFAs from Amnesia

DFA state explosion occurs primarily due amnesia, or the incompetence of the DFA to follow multiple partial matches with a single state of execution. Before proceeding with the cure to amnesia, we re-examine the connection between amnesia and the state explosion. As suggested previously, DFA state explosions usually occur due to those signatures which comprise of simple patterns followed by closures over characters classes (*e.g.* $.^*$ or [a-z]*). The simple pattern in these signatures can be matched with a stream of suitable characters and the subsequent characters can be consumed without moving away from the closure. These characters can begin to match either the same or some other reg-ex, and such situations of multiple partial matches have to be followed. In fact, every permutation of multiple partial matches has to be followed. A DFA represents each such permutation with a separate state due to its inability to remember anything other than its current state (amnesia). With multiple closures, the number of permutations of the partial matches can be exponential, thus the number of DFA states can also explode exponentially.

An intuitive solution to avoid such exponential explosions is to construct a machine, which can remember more information than just a single state of execution. NFAs fall in this genre; they are able to remember multiple execution states, thus they avoid state explosion. NFAs, however, are slow; they may require $O(n^2)$ state traversals to consume a character. In order to keep fast execution, we would like to ensure that the machine maintains a single active state. One way to enable single execution state and yet avoid state explosion is to equip the machine with a small and fast *cache*, to register key events during the parse, such as encountering a closure. Recall that the state explosion occurs because the parsing get stuck at a single or multiple closures; thus if the history buffer will register these events then one may avoid several states. We call this class of machine History based Finite Automaton (H-FA).

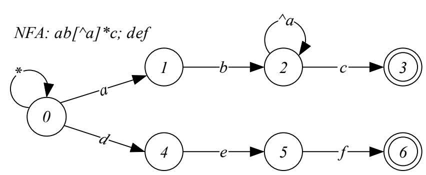
The execution of the H-FA is augmented with the history buffer. Its automaton is similar to a traditional DFA and consists of a set of states and transitions. However, multiple transitions on a single character may leave from a state (like in a NFA). Nevertheless, only one of these transitions is taken during the execution, which is determined after examining the contents of the history buffer; thus certain transitions have an associated condition. The contents of the history buffer are updated during the machine execution. The size of the H-FA automaton (number of states and transitions) depends upon those partial matches, which are registered in the history buffer; if we judiciously choose these partial matches then the H-FA can be kept extremely compact. The next obvious questions are: *i*) how to determine the partial matches? *ii*) Having determined them, how to construct an automaton? *iii*) How to execute the automaton and update the history buffer? We now desribe H-FA which attempts to answer these questions.

4.1 Motivating example

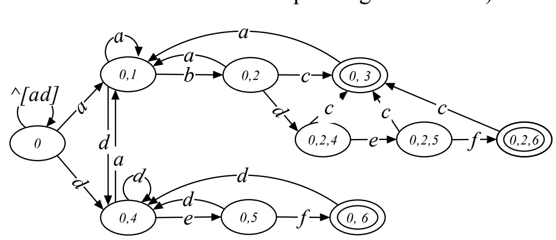
We introduce the construction and executing of H-FA with a simple example. Consider two reg-ex patterns listed below:

r_1 = `.*ab[^a]*c`; r_2 = `.*def`;

These patterns create a NFA with 7 states, which is shown below:

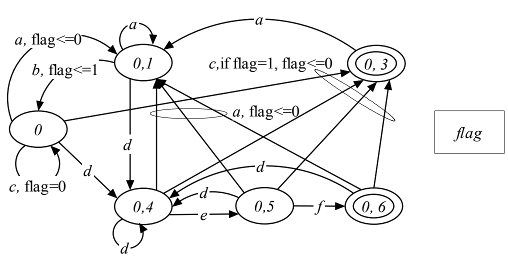


Let us examine the corresponding DFA, which is shown below (some transitions are omitted to keep the figure readable):



The DFA has 10 states; each DFA state corresponds to a subset of NFA states, as shown above. There is a small blowup in the number of states, which occurs due to the presence of the Kleene closure $[\hat{}a]^*$ in the expression r_1. Once the parsing reaches the Kleene closure (NFA state 2), subsequent input characters can begin to match the expression r_2, hence the DFA requires three additional states (0,2,4), (0,2,5) and (0,2,6) to follow this multiple match. There is a subtle difference between these states and the states (0,4), (0,5) and (0,6), which corresponds to the matching of the reg-ex r_2 alone: DFA states (0,2,4), (0,2,5) and (0,2,6) comprise of the same subset of the NFA states as the DFA states (0,4), (0,5) and (0,6) plus they also contain the NFA state 2.

In general, those NFA states which represent a Kleene closure appear in several DFA states. The situation becomes more serious when there are multiple reg-exes containing closures. If a NFA consists of n states, of which k states represents closures, then during the parsing of the NFA, several permutations of these closure states can become active; 2^k permutations are possible in the worst case. Thus the corresponding DFA, each of whose states will be a set of the active NFA states, may require total $n2^k$ states. These DFA state set will comprise of one of the n NFA states plus one of the 2^k possible permutations of the k closure states. Such an exponential explosion clearly occurs due to amnesia, as the DFA is unable to remember that it has reached these closure NFA states during the parsing. Intuitively, the simplest way to avoid the explosion is to enable the DFA to remember all closures which has been reached during the parsing. In the above example, if the machine can maintain an additional flag which will indicate if the NFA state 2 has been reached or not, then the total number of DFA states can be reduced. One such machine is shown below:



This machine makes transitions like a DFA; besides it maintains a flag, which is either set or reset (indicated by <=1, and <=0 in the figure) when certain transitions are taken. For instance transition on character `a` from state (0) to state (0,1) resets the flag, while transition on character `b` from state (0,1) to state (0) sets the flag. Some transitions also have an associated condition (flag is set or reset); these transitions are taken only when the condition is met. For instance the transition on character `c` from state (0) leads to state (0,3) if the flag is set, else it leads to state (0). This machine will accept the same language which is accepted by our original NFA, however unlike the NFA, this machine will make only one state traversal for an input character. Consider the parse of the string "`cdabc`" starting at state (0), and with the flag reset.

$$(0)\xrightarrow[\text{because flag is reset}]{c}(0)\xrightarrow{d}(0,4)\xrightarrow{a}\underset{\substack{\uparrow\\ \text{reset flag}}}{(0,1)}\xrightarrow{b}\underset{\substack{\uparrow\\ \text{set flag}}}{(0)}\xrightarrow[\text{because flag is set}]{c}(0,3)$$

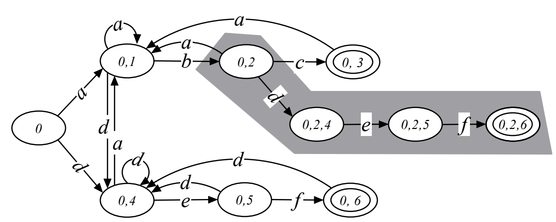
In the beginning the flag is reset; consequently the machine makes a move from state (0) to state (0) on the input character `c`. On the other hand, when the last character `c` arrives, the machine makes a move from state (0) to state (0,3) because the flag is set this time. Since state (0,3) is an accepting state, the string is accepted. Such a machine can be easily extended to maintain multiple flags, each indicating a closure. The transitions depend upon the state of all flags and they will be updated during certain transitions. As illustrated by the above example, augmenting an automaton with these flags can avoid state explosion. However, we need a more systematic way to construct these H-FAs, which we propose now.

4.2 Formal Description of H-FA

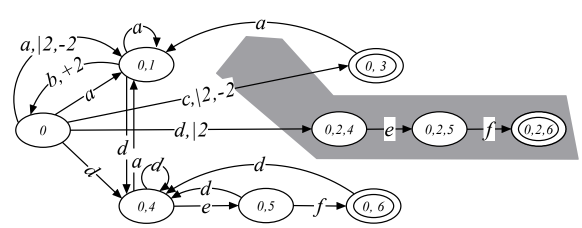
History based Finite Automata (*H-FA*) comprises of an automaton and a set called history. The transitions have *i*) an accompanied *condition* which depends upon the state of the *history*, and *ii*) an associated *action* which are inserts or remove from the history set, or both. *H-FA* can thus be represented as a 6-tuple $M = (Q, q_0, \Sigma, A, \delta, H)$, where Q is the set of states, q_0 is the start state, Σ is the alphabet, A is the set of accepting states, δ is the transition function, and H the history. The transition function δ takes in a character, a state, and a history state as its input and returns a new state and a new history state.

$$\delta : Q \times \Sigma \times H \rightarrow Q \times H$$

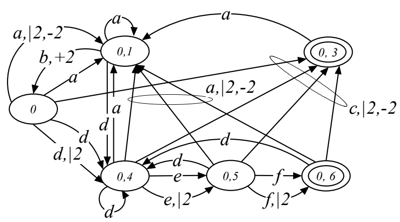
H-FAs can be synthesized either directly from a NFA or from a DFA. For clarity, we explain the construction from a combination of NFA and DFA. To illustrate the construction, we consider our previous example of the two reg-exes. First, we determine those NFA states of the reg-exes, which are registered in the history buffer (generally these are the closure NFA states). The first reg-ex, r_1 contains a closure represented by the NFA state 2; hence we keep a single flag in the history for this state. Afterwards, we identify those DFA states, which comprise of these closure NFA states, in this instance the NFA state 2. We call these DFA states (which are also highlight below) *fading states*:



In the next step, we attempt to remove the NFA state 2 from the fading DFA states. Notice that, if we will make a note that the NFA state 2 has been reached by setting the history flag, then we can remove the NFA state 2 from the fading states subset. The consequence is that these fading states may overlap with some DFA states in the non-fading region, thus they can be removed. Transitions which originated from a non-fading state and led to a fading state and vice-versa will now set and reset the history flag, respectively. Furthermore, all transitions that remain in the fading region will have an associated condition that the flag is set. Let us illustrate the removal of the NFA state 2 from the fading state (0, 2). After removal, this state will overlap with the DFA state (0); the resulting conditional transitions are shown below:



Here a transition with "|*s*" means that the transition is taken when history flag for the state *s* is set; "+*s*" implies that, when this transition is taken, the flag for *s* is set, and "-*s*" implies that, with this transition, the flag for *s* is reset. Notice that all outgoing transitions of the fading state (0,2) now originates from the state (0) and has the associated condition that the flag is set. Also those transitions which led to a non-fading state resets the flag and incoming transitions into state (0,2) originating from a non-fading state now has an action to set the flag. Once we remove all states in the fading region, we will have the following H-FA:



Notice that several transitions in this machine can be pruned. For example the transitions on character `d` from state (0) to state (0,4) can be reduced to a single unconditional transition (the pruning process is later described in greater detail). Once we completely prune the transitions, the H-FA will have a total of 4 conditional transitions; remaining transitions will be unconditional. When there are multiple closures, then multiple flags will be used and the procedure will be repeatedly applied to synthesize the H-FA.

The above example demonstrates a general method of the H-FA construction from a DFA. In order to achieve the maximum space reduction for a given number of history flags, the algorithm should only register those NFA states in the history buffer which appear most frequently in the DFA states. Afterwards, the above procedure can be repeatedly applied. With multiple flags in the history buffer, some transitions may have conditions that multiple history flags are set. Moreover, some transitions may either set or reset multiple flags. If there are *n* flags in the history buffer and *h* represents this *k*-bit vector, then a condition *C* will be a *k*-bit vector, which becomes true whenever all those bits of *h* are set whose corresponding bits in *C* are also set.

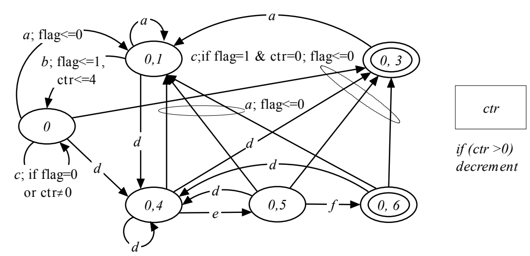
The representation of conditions as vectors eases out the pruning process, which is carried out immediately after the construction. The pruning process eliminates any transition with condition C_1, if another transition on condition C_2 exists between the same pair of states, over the same character such that the condition C_1 is a subset of the condition C_2 (*i.e.* C_2 is true whenever C_1 is true) and the actions associated with both the transitions are the same. In general, pruning process eliminates a large number of transitions, and it is essential in reducing the memory requirements of H-FAs. However, even after pruning, there can be a blowup in the number of transitions. In the worst-case, if we eliminate *k* NFA states from the DFA by employing *k* history flags then there can be up to 2^k additional conditional transitions in the resulting H-FA, thus there will be little memory reduction. However, such worst-cases are rare; normally there is only a small blowup in the number of transitions. Analysis of the blowup and implementation of history buffer is presented in great detail in the technical report [35].

5. H-cFA: Curing DFAs from Acalculia

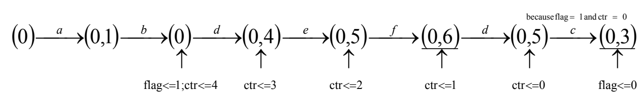
We now propose "History based counting finite Automata" or H-cFA, which efficiently cures traditional FA from acalculia, due to which a FA is unable to efficiently count the occurrences of certain sub-expressions. We begin with an example; we consider the same set of two reg-exes with the closure in the first reg-ex replaced with a length restriction of 4, as shown below:

r_1 = `.*ab[^a]`^{4}`c`; r_2 = `.*def`;

A DFA for these two reg-exes will require 20 states. The blowup in the number of states in the presence of the length restriction occurs due to acalculia or the inability of the DFA to keep track of the length restriction. Let us now construct an H-cFA for these reg-exes. The first step in this construction replaces the length restriction with a closure, and constructs the H-FA, with the closure represented by a flag in the history buffer. Subsequently with every flag in the history buffer, a counter is appended. The counter is set to the length restriction value by those conditional transitions which set the flag, while it is reset by those transitions which reset the flag. Furthermore, those transitions whose condition is a set flag are attached with an additional condition that the counter value is 0. During the executing of the machine, all positive counters are decremented for every input character. The resulting H-cFA is shown below:



Consider the parse of the string "`abdefdc`" by this machine starting at the state (0), and with the flag and counter reset.



As the parsing reaches the state (0,1), and makes transition to the state (0), the flag is set, and the counter is set to 4. Subsequent transitions decrements the counter. Once the last character *c* of the input string arrives, the machine makes a transition from state (0,5) to state (0,3), because the flag is set and counter is 0; thus the string is accepted. This example illustrates the straightforward method to construct H-cFAs from H-FAs. Several kinds of length restrictions including "greater than *i*", "less than *i*" and "between *i* and *j*" can be implemented. Each of these conditions will require an appropriate condition with the transition. For example, "less than *i*" length restriction will require that the conditional transition becomes true when the history counter is greater than 0.

From the hardware implementation perspective, a greater than or less than condition requires approximately equal number of gates needed by an equality condition, hence different kinds of length restrictions are likely to have identical implementation cost. In fact, a reprogrammable logic can be devised equally efficiently, which can check each of these conditions. Thus, the architecture will remain flexible in face of the frequent signature updates. This simple cure to acalculia is extremely effective is reducing the number of states, specifically in the presence of long length restrictions. Snort signatures comprises of several long length restrictions, hence H-cFA is extremely valuable in implementing these signatures. We now present our detailed experimental results, where we highlight the effectiveness of our cures to the three reg-ex problems.

Table 1. Splitting results: Left columns show the properties of complete reg-ex, while right columns show the properties of prefixes

Source	# of rules	Regular expressions implementation before split					Regular expressions prefix features after split				
		Avg. ASCII length	# of closures	# of length restrictions	Number of DFA	Total memory	Avg. ASCII length	# of closures	# of length restrictions	Number of DFA	Total memory
Cisco	68	44.1	70	15	6	973 MB	19.8	19	1	1	152 MB
Linux	70	67.2	31	0	4	30.7 MB	21.4	11	0	2	15.8 MB
Bro	648	23.64	0	0	1	3.77 MB	16.1	0	0	1	1.23 MB
Snort rule 1	22	59.4	9	11	5	114.6 MB	36.9	6	6	3	32.1 MB
Snort rule 2	10	43.72	11	10	2	64.2 MB	16	1	2	1	6.5 MB
Snort rule 3	19	30.72	8	6	N/A	N/A	13.8	5	1	2	2.42 MB

6. Experimental Evaluation

We have carried out a comprehensive set of experiments in order to evaluate the effectiveness of our proposed cure to the three problems, insomnia, amnesia, and acalculia. Our primary signature sets are the regular expressions used in the security appliances from Cisco Systems [33]. These rule sets comprise of more than 750 moderately complex regular expressions. Cisco often uses DFAs to implements these rules; consequently, due to the state explosion, they employ more than a gigabyte of memory; still the parsing rates remains sub-gigabits/s. We also considered the reg-ex signatures used in the open source Snort and Bro NIDS, and in the Linux layer-7 application protocol classifier. Linux layer-7 protocol classifier comprises of 70 rules, while Snort rules consists of more than a thousand and half reg-exes. In Snort, these reg-exes need not be matched simultaneously, because before a packet is parsed, it is classified, and based upon the classification, only a subset of the reg-exes are considered. Therefore, we only group those Snort signatures which correspond to the overlapping header rules, *i.e.* those header rules which a single packet can match (we present results of three such groups). For the Bro NIDS, we present results for the HTTP signatures, which contain 648 reg-exes.

Since Cisco rules comprise of a large number of patterns, our first step in implementing them involves grouping these rules into two sets: one consisting of all those signatures which do not contain a closure, while the second containing all signatures with at least one closure. Clearly, the first set can be compiled into a composite DFA without any difficulty. It is the second set of reg-exes, which are problematic and requires our cure mechanisms; therefore all our results are over these signatures. First we present the result of our splitting algorithm, which leads to cure from insomnia.

6.1 Reg-ex splitting results

For reg-ex splitting, our representative experiment sets the slow path packet diversion probability at 1%, and computes the cut in the reg-exes. Our normal traffic traces were derived from the MIT DARPA Intrusion Detection Data Sets [29], while the anomalous traffic traces were provided to us by Cisco Systems. We have also created synthetic anomalous traces, by inserting some signatures into the normal traffic trace. With these traces, we have split the reg-exes into prefixes and suffixes. Afterwards the prefixes are extended by one or two more characters to ensure that slow path remains substantially less loaded. We summarize the result of the splitting process on the reg-exes in Table 1.

In this table, we first list the properties of the original reg-exes and the memory needed to implement them. Notice that most of these reg-ex sets are sub-divided into multiple sets. Each set is compiled into a separate DFA, because it is difficult to compile all reg-exes into as a single composite DFA (due to state explosion). The implication of this sub-division is that since each DFA is executed simultaneously, the parsing rate for a given memory bandwidth will reduce. In the same table, on the right hand side, we list the properties of the prefixes after the splitting. Notice that these prefixes can be compiled into fewer DFAs, which will yield higher parsing rates and less per flow state. Additionally, these DFAs are relatively compact however their memory requirements are still much higher compared to the current embedded memory densities. The prime reason is that the prefixes still contain a small number of closures which lead to a moderate state explosion. We now present the results of our cure to amnesia, which avoids such state explosion in the prefix automaton.

Table 2. Results of the H-FA and H-cFA construction, there results are for the prefix portions of the reg-exes

Source	# of closures, # of length restriction	DFA		Composite H-FA / H-cFA					% space reduction with H-FA	H-FA parsing rate speedup
		# of automata	total # of states	# of flags in history	# of counters in history	Total # of states	Max # of transitions / character	Total # of transitions		
Cisco64	14, 1	1	132784	6	0	3597	2	1215450	94.69	-
Cisco64	14, 1	1	132784	13	0	1861	8	682718	96.77	-
Cisco68	19, 1	1	328664	17	0	2956	8	1337293	97.03	-
Snort rule 1	6, 6	3	62589	5	6	583	8	238107	97.40	3x
Snort rule 2	1, 2	1	12703	1	2	71	2	27498	98.58	-
Snort rule 3	5, 1	2	4737	5	1	116	4	46124	93.48	2x
Linux70	11, 0	2	20662	9	0	1304	8	546378	81.63	2x

6.2 H-FA and H-cFA construction results

For the prefixes, we construct H-FAs, which dramatically reduces the total memory. Snort prefixes comprise of several long length restrictions therefore we construct H-cFAs for these. We find that H-cFA is extremely effective in reducing the memory; without using the counting capability of H-cFA, a composite automaton for Snort prefixes explodes in size. In Table 2, we report the results from our representative experiments. We highlight the number of flags and counters that we employ in the history buffer. For Cisco rules, we also show how varying the number of flags affects the H-FA size. In general, with more history flags, the H-FA is more compact. Notice that the traditional DFA compression techniques including the D^2FA [34] can be applied to H-FA, thereby further reducing the memory.

The table also highlights an important result: the blowup in the number of conditional transitions in the H-FA generally remains very small. In a DFA there are 256 outgoing transitions, while in most of the H-FAs there are less than 500. Thus, there is less than 2-fold blowup in the number of transitions; on the other hand reduction in the number of states is generally a few orders of magnitude, thus the net effect is significant memory reduction. Due to space restrictions, we are currently unable to present further details of the H-FA and H-cFA construction.

7. CONCLUDING REMARKS

In this paper, we propose several mechanisms to enhance the performance of regular expressions based parsers, which are widely used to implement network intrusion detection systems. We begin by identifying the three key limitations of traditional approach, and categorized them as *insomnia*, *amnesia* and *acalculia*. We propose solutions for each of the limitation, and show that our solutions are orthogonal with respect to each other; hence they can be employed in unison.

Based upon experiments which were carried out on real signatures sets drawn from a collection of widely used networking systems, we show that our solutions are indeed effective. It can reduce the memory requirements of the state-of-the-art regular expressions implementations by up to 100 times, while also enabling a two to three fold increase in the packet throughput. We also pay adequate attention to several complications that appears in real networks, *e.g.* DoS protection, multiple simultaneous flows, and packet multiplexing. Therefore, we believe that the proposed solutions can aid in implementing network intrusion detection and prevention systems much more securely and economically.

REFERENCES

[1] R. Sommer, V. Paxson, “Enhancing Byte-Level Network Intrusion Detection Signatures with Context,” ACM conf. on Computer and Communication Security, 2003, pp. 262--271.

[2] J. E. Hopcroft and J. D. 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] Bro: A System for Detecting Network Intruders in Real-Time. http://www.icir.org/vern/bro-info.html

[5] M. Roesch, “Snort: Lightweight intrusion detection for networks,” In Proc. 13th Systems Administration Conference (LISA), USENIX Association, November 1999, pp 229–238.

[6] S. Antonatos, et. al, “Generating realistic workloads for network intrusion detection systems,” In ACM Workshop on S & P, 2004.

[7] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” Comm. of the ACM, 18(6):333–340, 1975.

[8] B. Commentz-Walter, “A string matching algorithm fast on the average,” Proc. of ICALP, pages 118–132, July 1979.

[9] S. Wu, U. Manber,” A fast algorithm for multi-pattern searching,” Tech. R. TR-94-17, Dept. of Comp. Science, Univ of Arizona, 1994.

[10] Fang Yu, et al., “Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection”, UCB tech. report, 2005.

[11] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, “Deterministic memory-efficient string matching algorithms for intrusion detection,” IEEE Infocom 2004, pp. 333--340.

[12] L. Tan, and T. Sherwood, “A High Throughput String Matching Architecture for Intrusion Detection and Prevention,” ISCA 2005.

[13] I. Sourdis and D. Pnevmatikatos, “Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching,” Proc. IEEE Symp. on Field-Prog. Custom Computing Machines, Apr. 2004, pp. 258–267.

[14] S. Yusuf and W. Luk, “Bitwise Optimised CAM for Network Intrusion Detection Systems,” IEEE FPL 2005.

[15] R. Sidhu and V. K. Prasanna, “Fast regular expression matching using FPGAs,” In IEEE Symposium on Field- Programmable Custom Computing Machines, Rohnert Park, CA, USA, April 2001.

[16] C. R. Clark and D. E. Schimmel, “Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns,” In Proceedings of 13th International Conference on Field Program.

[17] J. Moscola, et. al, “Implementation of a content-scanning module for an internet firewall,” IEEE Workshop on FPGAs for Custom Comp. Machines, Napa, USA, April 2003.

[18] R. W. Floyd, and J. D. Ullman, “The Compilation of Regular Expressions into Integrated Circuits”, Journal of ACM, vol. 29, no. 3, pp 603-622, July 1982.

[19] Scott Tyler Shafer, Mark Jones, “Network edge courts apps,” http://infoworld.com/article/02/05/27/020527newebdev_1.html

[20] TippingPoint X505, www.tippingpoint.com/products_ips.html

[21] Cisco IOS IPS Deployment Guide, www.cisco.com

[22] Tarari RegEx, www. tarari.com/PDF/RegEx_FACT_SHEET.pdf

[23] N.J. Larsson, “Structures of string matching and data compression,” PhD thesis, Dept. of Computer Science, Lund University, 1999 .

[24] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, “Deep Packet Inspection using Parallel Bloom Filters,” IEEE Hot Interconnects 12, August 2003. IEEE Computer Society Press.

[25] Z. K. Baker, V. K. Prasanna, “Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs,” in Field Prog. Logic and Applications, Aug. 2004, pp. 311–321.

[26] Y. H. Cho, W. H. Mangione-Smith, “Deep Packet Filter with Dedicated Logic and Read Only Memories,” Field Prog. Logic and Applications, Aug. 2004, pp. 125–134.

[27] M. Gokhale, et al., “Granidt: Towards Gigabit Rate Network Intrusion Detection Technology,” in FPL, Sept. 2002, pp. 404–413.

[28] J. Levandoski, E. Sommer, and M. Strait, “Application Layer Packet Classifier for Linux”. http://l7-filter.sourceforge.net/.

[29] “MIT DARPA Intrusion Detection Data Sets,” http://www. ll.mit.edu/IST/ideval/data/2000/2000_data_index.html.

[30] Vern Paxson et al., “Flex: A fast scanner generator”.

[31] SafeXcel Content Inspection Engine, hardware regex acceleration IP.

[32] Network Services Processor, OCTEON CN31XX, CN30XX Family.

[33] Will Eatherton, John Williams, “An encoded version of reg-ex database from cisco systems provided for research purposes”.

[34] S. Kumar et al, “Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection”, in ACM SIGCOMM'06, Pisa, Italy, September 12-15, 2006.

[35] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, “Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia”, Washington University technical report, 2006.