O^{3}FA: A Scalable Finite Automata-based Pattern-Matching Engine for Out-of-Order Deep Packet Inspection

Xiaodong Yu, Wu-chun Feng, Danfeng (Daphne) Yao
Department of Computer Science
Virginia Tech
Blacksburg, VA, USA
{xdyu, feng, danfeng}@cs.vt.edu

Michela Becchi
Dept. of Electrical and Computer Engineering
University of Missouri
Columbia, MO, USA
becchim@missouri.edu

ABSTRACT

To match the signatures of malicious traffic across packet boundaries, network-intrusion detection (and prevention) systems (NIDS) typically perform pattern matching after flow reassembly or packet reordering. However, this may lead to the need for large packet buffers, making detection vulnerable to denial-of-service (DoS) attacks, whereby attackers exhaust the buffer capacity by sending long sequences of out-of-order packets. While researchers have proposed solutions for exact-match patterns, regular-expression matching on out-of-order packets is still an open problem. Specifically, a key challenge is the matching of complex sub-patterns (such as repetitions of wildcards matched at the boundary between packets). Our proposed approach leverages the insight that various segments matching the same repetitive sub-pattern are logically equivalent to the regular-expression matching engine, and thus, interchanging them would not affect the final result.

In this paper, we present O^{3}FA, a new finite automata-based, deep packet-inspection engine to perform regular-expression matching on out-of-order packets *without* requiring flow reassembly. O^{3}FA consists of a deterministic finite automaton (FA) coupled with a set of prefix-/suffix-FA, which allows processing out-of-order packets on the fly. We present our design, optimization, and evaluation for the O^{3}FA engine. Our experiments show that our design requires 20x-4000x less buffer space than conventional buffering-and-reassembling schemes on various datasets and that it can process packets in real-time, i.e., without reassembly.

1. INTRODUCTION

Regular-expression matching is a core task in deep packet inspection (DPI), which is a fundamental networking operation. A traditional form of DPI consists of searching the packet payload against a set of patterns. Network intrusion detection systems (NIDS) are an essential part of network security devices. A NIDS receives and processes packets, and then reports the possible intrusions. Some well-known open-source NIDS – such as Snort and Bro – employ DPI as their core; most major networking companies offer their own NIDS solutions (e.g., security appliances from Cisco and Juniper Networks). In NIDS, every pattern represents a signature of malicious traffic; thus, the DPI engine of a NIDS inspects the incoming packets payloads against all available signatures and triggers pre-defined actions if a match is detected. A regular expression can cover a wide variety of pattern signatures [1-3]. Because of their expressive power, regular expressions have been increasingly adopted to express pattern sets in both industry and academia. To allow multi-pattern search, current NIDS mostly represent the pattern-set through finite automata (FA) [4], either in their deterministic or in their non-deterministic form (DFA and NFA, respectively).

A large body of research has focused on developing efficient regular-expression matching engines. For memory-centric solutions, where the automaton is stored in memory, DFA-based approaches are more popular than NFA-based ones because of their predictable memory bandwidth requirements. Specifically, processing an input character involves only one DFA state traversal, which can be translated into a deterministic number of memory accesses. However, this attractive property comes at the cost of potentially large requirements for memory space. In fact, DFAs constructed from large and complex sets of regular expressions may suffer from the state explosion problem, making the storage requirements prohibitively large. State explosion can take place during DFA generation when the corresponding regular expressions have repetitions of wildcards and/or large character sets. Several variants of DFA [5-11] have been proposed to address this problem, and thus, limit the effects of state explosion with varying degree.

In real-world scenarios, a network data stream can span multiple packets. Those packets can arrive at network security devices out of order due to multiple routes, packet retransmission, or NIDS evasion. Thus, the packets must be re-ordered appropriately upon receipt. Previous work analyzing Internet traffic has reported that about 2%-5% of packets are affected by reordering [12-14]. However, these studies have focused on benign traffic; while attackers may intentionally mis-order legitimate traffic to trigger denial-of-service (DoS) attacks [14]. NIDS face challenges [15] when processing data streams that span across out-of-order packets, especially when performing regular-expression matching against traffic containing malicious content that is located *across* packets boundaries. In such cases, the malicious patterns are split and carried by multiple packets; and NIDS cannot detect them by processing those packets individually.

Several solutions have been proposed to address the problem of processing out-of-order packets in NIDS. One approach that is widely adopted in current network devices is *packet buffering* and *stream reassembling* [14, 16-18]. In this case, incoming packets are buffered and packet streams are reassembled based on the information in the header fields. Regular-expression matching is then performed on the reassembled data stream. This approach is intuitive and easy to implement but can be very resource-intensive and vulnerable to DoS attacks, whereby attackers exhaust the packet buffer capacity by sending long sequences of out-of-order packets. Recently, researchers have proposed several new solutions [19-21] aimed to relieve packet-buffer pressure or even avoid packet buffering and reassembling. This is done by tracking all possible traversal paths or leveraging data structures such as suffix trees. While they alleviate the burden of handing out-of-order packets to some extent, these methods are either applicable only to simple patterns (exact-match strings or fixed-length patterns) or suffer from undesirable worst-case properties (and are therefore still vulnerable to DoS attacks).

In this work, we aim to provide a solution that (1) processes out-of-order packets without requiring packet buffering and stream reassembling, (2) relies only on finite automata, and (3) handles regular expressions with complex sub-patterns. One of the main challenges in our design comes from handling regular expressions that include unbounded repetitions of wildcards and large character sets. Why is this a challenge? These sub-patterns can represent unbounded sets of exact-match substrings that cannot be exhaustively enumerated. Our solution leverages the following observation: *all exact-match strings that match a repetition sub-pattern are functionally equivalent from the point of view of the regular-expression matching engine and interchanging them will not affect the final matching result*. Our proposed solution consists of regular DFAs coupled with a set of supporting FAs either in NFA or DFA form. The supporting FAs are used to detect and record – using only a few states (typically no more than five) – segments of packets that can potentially be part of a match across packet boundaries. While processing packets out-of-order, those segments can be dynamically retrieved from the recorded states and can be then used to resolve matches across packet boundaries.

To be efficient, any automata-based solution requires minimizing the number of automata, their size, and the number of states that can be active in parallel. Our approach includes optimizations aimed to achieve these goals. In summary, our contributions are as follows:

- O^{3}FA, a new finite automata-based DPI engine to perform regular-expression matching on out-of-order packets in real time, i.e., without requiring flow reassembly.
- Several optimizations to improve the average-擦色 and worst-case behavior of the O^{3}FA engine and an analysis of how packet ordering affects the buffer size.

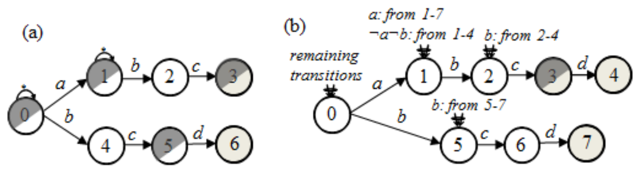


Figure 1. (a) NFA and (b) DFA accepting regular expressions a.*bc and bcd. Accepting states are colored gray. States active at the end of the processing of input acbc are highlighted with a diagonal filling. In the NFA, states 0 and 1 have a self-loop on any characters of the alphabet. In the DFA, state 1 has incoming transitions on character b from states 1 to 7, and incoming transitions from states 1 to 4 on any characters other than a and b (incoming transitions to states 0, 2 and 5 can be read in the same way).

- An evaluation our O^{3}FA engine on various real-world and synthetic datasets, where our results show that the O^{3}FA engine requires 20x-4000x less buffer space than conventional buffering and reassembling-based solutions but with only 0.0006%-5% traversal overhead.

2. BACKGROUND & RELATED WORK

Finite automata (FA) are widely used to perform regular-expression matching. In automata-based approaches, the matching operation is equivalent to a FA traversal that is guided by the content of the input stream. Worst-case guarantees on the input processing time can be met by bounding the amount of per-character processing. As the basic data structure in the regular-expression matching engine, the finite automaton must be deployable on a reasonably provisioned hardware platform. As the size of pattern-sets and the expressiveness of individual patterns increase, limiting the size of the automaton becomes challenging. The exploration space is characterized by a trade-off between the size of the automaton and the worst-case bound on the amount of per-character processing. Non-deterministic and deterministic finite automata (NFAs and DFAs, respectively) are at the two extremes of this exploration space. NFAs have a limited size but can require expensive per-character processing, whereas DFAs offer limited per-character processing but at the cost of a possibly large automaton. As an example, Figure 1 shows the NFA and DFA accepting regular expressions *a.*bc* and *bcd* (the dot-star sub-pattern “.*” represents any segment with any length). In the figure, the states active after processing input stream *acbc* are highlighted using diagonal filling. As can be seen, the NFA consists of fewer states, i.e., seven (7) versus eight (8), while the DFA leads to less per-character processing, i.e.,one (1) versus four (4) concurrently active states.

While offering this attractive traversal property, DFAs can suffer from the well-known state explosion problem. Each DFA state corresponds to a set of NFA states that can be simultaneously active [4]. Therefore, given an *N*-state NFA, the functionally equivalent DFA may potentially have up to 2^N states. This state explosion may limit the DFA’s ability to handle large and complex sets of regular expressions (typically those that include bounded and unbounded repetitions of wildcards or large character sets). Existing proposals targeting DFA-based solutions have focused on two aspects: (i) designing compression schemes aimed at minimizing the DFA memory footprint; and (ii) devising new automata to alternate DFAs in case of state explosion. Alphabet reduction [5, 22-24], run-length encoding [5], default transition compression [22, 25], state merging [7] and delta-FAs [26] fall in the first category, while multiple-DFAs [5, 6], hybrid-FAs [7], history-based-FAs [8], XFAs [9], counting-FAs [10], and JFAs [11] fall in the second category. All of these solutions, however, have been designed to operate on reassembled packet streams.

The classic approach (to tackle the packet reordering problem) buffers the received data packets, reorders them, and finally reassembles the packet stream (or packet flow). Dharmapurikar et al. [14] propose a system to buffer and reorder packets. Their system consists of a packet analyzer, an out-of-order packet processing unit, and a buffer manager. It mitigates the risk of denial-of-service (DoS) attacks by forcing attackers to use multiple attacking hosts. However, the system is still vulnerable to attacks, exhausting the buffer capacity. Similar packet buffering and stream reassembly solutions have also been proposed and adopted in industry (e.g. Cisco [16], Nortel Networks [17], and Netrake [18]).

Despite its widespread adoption, this buffering and reassembling approach is vulnerable to DoS attacks whereby attackers exhaust the buffer capacity by sending long sequences of out-of-order packets. There have been a handful of proposals attempting to reduce these risks by avoiding packet buffering and stream reassembly.

For example, Varghese et al. [27] propose Split-Detect. This system splits the signatures of malicious traffic into pieces, performs deep packet inspection on these sub-signatures, and diverts the TCP packets for reassembly from the fast path to the slow path upon detection of any of these sub-signatures. Split-Detect can achieve up to 90% storage requirement reduction compared to conventional NIDS. However, rather than avoiding stream reassembly, it offloads it to the slow path. In addition, Split-Detect is restricted to exact-match signatures and patterns with a fixed length. In contrast, our approach works on arbitrarily-sized patterns.

Johnson et al. [21] propose a DFA-based solution that, for each packet, performs parallel traversals from each and all DFA states. Because the initial DFA state is unknown when processing an out-of-order packet, any DFA state must be considered a potential initial state. A post-processing step is then be performed to reconstruct valid traversals at packet boundaries. Although this scheme may be effective in the presence of non-malicious traffic (where the traversal is limited to a few DFA states), it does not provide a good worst-case bound, and it may involve a large amount of post-processing.

More recently, Chen et al. [20] propose AC-Suffix-Tree. This scheme avoids packet buffering and stream reassembly by combining an Aho-Corasick DFA with a suffix tree. Zhang et al. [19] propose On-Line Reassembly (OLR), a scheme that stores patterns in a directed acyclic word graph (DAWG). Both of these solutions, however, apply only to exact-match patterns and are unable to handle regular expressions, which are more common in real-world applications. Our approach does not suffer from this limitation.

3. O^{3}FA DESIGN

In this section, we present our approach for performing regular-expression matching on out-of-order packets without requiring prior stream reassembly. The main challenge in this problem comes from *the handling of matches across packet boundaries*. At a high level, our proposed solution couples one or more DFAs with *supporting-FAs*. The DFAs find matches within a packet while the supporting-FAs detect and record segments of packets that can potentially be part of a match across packet boundaries. While processing out-of-order packets, these segments can be dynamically retrieved from the state information collected on the supporting-FAs, and they can subsequently be concatenated to the incoming packet in order to handle cross-packet matches.

To have an intuition of this idea, consider matching input stream *cabcdeab* against pattern *b.*cde*. Let us assume that this input stream spans across two packets: P_1=*cabc* and P_2=*deab*. We can observe that pattern *b.*cde* is matched across packet boundaries (the match starts in P_1 and ends in P_2; the *segments* of P_1 and P_2 involved in the match are underlined). If we use a DFA, this match will be detected only if packets P_1 and P_2 are processed in order. If the packets are processed out of order, we will need a way to detect that segment *de* of P_2 and segment *bc* of P_1 are partial matches (specifically, they match the suffix and the prefix of the considered patterns, respectively). We will then use this information to reconstruct the match. Our proposed supporting-FA serves this purpose. We note that, because *b.*cde* is neither an exact-match string nor a fixed-length pattern, it cannot be handled by previous approaches such as SplitDetect [27], AC-Suffix-Tree [20], and ORL [19].

Because we are concerned about patterns with variable length, we focus on regular expressions containing repetitions of characters (e.g., *c+* and *c**), character sets (e.g, *[c_i-c_j]**), and wildcards (*.**). We note that regular expressions without these features can be handled by traditional methods. For example, a regular expression containing a non-repeated character set *[c_i-c_j]* can be transformed by exhaustive enumeration into a set of exact-match patterns. For readability and in the interest of space, the remaining description focuses on the more general case (wildcard repetitions); however, our solution applies to all kinds of repetitions.

A central question in the O^{3}FA design is as follows: how can we identify the *minimal* packet segments that must be recorded in order to handle cross-packet matches? We note that excessively long segments would pose pressure on the required packet buffer and on the amount of processing involved in the matching operation, thus leading to inefficiencies. Our design leverages the following observations.

Observation 1: If a regular expression *R* is matched across a set of packets $P_1,.., P_N$, then the suffix of P_1 must match a prefix of *R* and the prefix of P^N must match a suffix of *R*.

Observation 2: Given a regular expression *R* in the form $sp_1.*sp_2$ and an input stream *I* containing a matching segment of the form M_1M*M_2, where M_1 matches sp_1 and M_2 matches sp_2, any modifications to *I* that substitutes $M*$ with a shorter segment will not affect the match outcome.

According to Observation 1, O^{3}FA must detect segments of incoming packets that match any suffixes/prefixes of the considered regular expressions. These segments are recorded by storing the corresponding matching states information, and they can be dynamically retrieved and properly concatenated with later-arrival packets to detect cross-boundary matching. For example, while matching regular expression *b.*cde* on packets P_1=*caba*, P_2=*dcac* and P_3=*dead* that arrive in order $P_3 \rightarrow P_1 \rightarrow P_2$, we first detect that segment *de* in P_3 matches suffix *de*, and then that segment *ba* in P_1 matches prefix *b.**. When P_2 arrives, we retrieve those segments and concatenate them with P_2, then conduct regular expression matching on *badcacde* and detect the cross-boundary matching of *b.*cde*. In general, prefix *b.** can match arbitrarily long strings, which may span across any number of intermediate packets.

However, according to Observation 2, in order to reconstruct the match, it is sufficient to record the shortest segment of the input stream that matches the regular expression with the wildcard repetition. In the considered example, rather than recording segment *ba* of packet P_1, we can simply record segment *b*. In addition, if a regular expression *p.*s* is matched across a set of packets $P_1,..,P_N$ such that the suffix of P_1 matches *p* and the prefix of P_N matches *s*, recording the intermediate packets $P_2,..,P_{N-1}$ will not be necessary for matching purposes.

The design is complicated by the fact that multiple regular expressions would require recording multiple segments, possibly leading to inefficiencies. In section 4 we propose a mechanism (that we call Functionally Equivalent Packets) to combine segments related to different regular expressions. As we will discuss, this method leverages the overlap between different segments.

3.1 O^{3}FA Data Structure

We now discuss the design of O^{3}FA, a composite automata-based solution that implements the scheme described above. As mentioned, O^{3}FA consists of two components:

- One or more “regular” **DFAs** used to perform regular expression matching and constructed based on the given regular expression set. Any automata optimization techniques [5-11, 22, 25, 26, 28] can be applied to these DFAs.
- **Supporting-FAs** used to detect and record significant segments of incoming packets. According to the above discussion, supporting-FAs should be constructed to detect segments matching regular expressions’ prefixes and suffixes, and can therefore be of two kinds: *prefix-FAs* and *suffix-FAs*. These automata can be in either NFA or in DFA form.

In order to build the prefix- and suffix-FAs, we split the regular expressions at the positions of the repetition sub-patterns. For example, regular expression *abc.*def.*ghk* will be broken down into three sub-patterns: *.*abc.**, *.*def.** and *.*ghk* (the *.** before *abc* is due to the fact that the original regular expression is unanchored, that is, it can be matched at any position of the input stream). This breakdown is possible because the supporting-FAs are used to record packet segments, and not to perform pattern matching; the short packet segments recorded by breaking down the regular expressions into sub-patterns will be concatenated into larger segments during processing. This breakdown allows significantly simplifying the supporting automata: by allowing dot-star terms to appear only at the beginning or at the end of each pattern, it will avoid state explosion when representing the supporting-FAs in DFA form. The full prefix and suffix sets corresponding to the given sub-patterns are: {*.*abc.*, .*abc, .*ab, .*a, .*def.*, .*def, .*de, .*d, .*ghk, .*gh, .*g*} and {*.*abc.*, abc.*, bc.*, c.*, .*def.*, def.*, ef.*, f.*, .*ghk, ghk, hk, k*}, respectively.

However, some simplifications are possible. First, since the suffixes must be matched at the beginning of packets (Observation 1) and can end anywhere within a packet, the “*.**” at the end of each suffix is redundant. Second, patterns that are common to prefix and suffix sets (e.g. *.*abc*, *.*def*, *.*ghk*) can be removed from the prefix set (these patterns would lead to the detection of the same segments[1]). Third, sub-patterns that are covered by more general patterns belonging to the same set (e.g. *abc* is a special case of *.*abc*) can also be eliminated. After these simplifications, the prefix and suffix sets used to build the prefix- and suffix-FA will be {*.*abc.*, .*ab, .*a, .*def.*, .*de, .*d, .*gh, .*g*} and {*.*abc, bc, c, .*def, ef, f, .*ghk, hk, k*}, respectively. Note that the suffix set contains both anchored and unanchored patterns (the latter start by “*.**”). These two groups of patterns can be compiled in two different suffix-Fas (i.e., an *anchored* and an *unanchored suffix-FA*) to al-

[1] The reason why these patterns are removed by the prefix set will become apparent later. Specifically, since all prefix matches in the middle of packets can be discarded, keeping these patterns in the suffix set ensures that they will be detected by the suffix-FA.

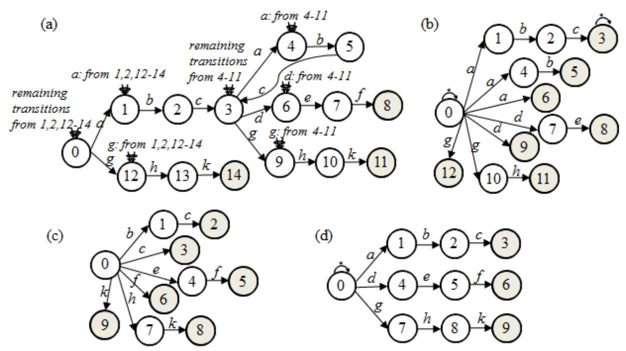


Figure 2. (a) DFA accepting pattern set {abc.*def, ghk}, (b) prefix-FA, (c) anchored suffix-FA and (d) unanchored suffix-FA built upon corresponding prefix set, anchored suffix set and unanchored suffix set. Accepting states are colored gray.

low space optimizations when representing the automata in DFA form.

During processing, upon a match within a supporting-FA, the corresponding accepting state must be recorded, and it will then be used to retrieve the packet segments to be concatenated to the current input packet. This "extended" input packet will then be processed by the "regular" DFA. However, some matches that occur within the supporting-FAs can be discarded, thus diminishing the amount of information that must be recorded to reconstruct relevant packet segments. First, since prefixes need to be matched only at the end of packets (Observation 1), all prefix matches occurring in the middle of any packets can be discarded. Second, if multiple anchored suffixes of a regular expression are matched, only the longest one must be recorded (shorter suffixes will be subsumed by it).

Figure 2 shows an example on regular expression set {*abc.*def, ghk*}, both patterns are unanchored (that is, they can be matched at any position of the input stream). The prefix set, anchored suffix set and unanchored suffix set are {*.*abc.*, .*ab, .*a, .*de, .*d, .*gh, .*g*}, {*bc, c, ef, f, hk, k*} and {*.*abc, .*def, .*ghk*}, respectively. Figure 2 (a)-(d) show the resulting regular DFA, prefix-FA, anchored suffix-FA and unanchored suffix-FA; all supporting-FAs are left in NFA form. We assume three input packets: P_1=*bhab*, P_2=*cegh* and P_3=*adef*, with the arriving order being $P_3 \rightarrow P_1 \rightarrow P_2$. After P_3 is processed, the matching state sets of regular DFA and anchored suffix-FA are empty; the unanchored suffix-FA matching states is 6; the prefix-FA matching states are {8, 9}; since those matches do not happen at the tail of P_3, they will be discarded. Then, we process P_1; the matching state sets of regular DFA, anchored suffix-FA and unanchored suffix-FA are empty; the prefix-FA matching states are {5, 6}; since only matching state 5 is active at the end of P_1 processing, this sole prefix-FA state will be recorded. When P_2 arrives, we should first check the recorded information of its previously processed neighbor packets (i.e., predecessor P_1 and successor P_3): P_1 has a recorded prefix-FA state 5; the retrieved segment is *ab* and should be concatenated to P_2 as a prefix. P_3 has a recorded unanchored suffix-FA state 6; the retrieved segment is *def* and should be concatenated to P_2 as a suffix. Then, the modified P_2 is *abceghdef*; after it is processed with the regular DFA, the matching of the pattern *abc.*def* will be reported.

4. OPTIMIZATIONS

Our basic O^{3}FA design has two limitations: it can lead to false positives (that is, it may report invalid matches) and it can suffer from inefficiencies during processing. In this section, we describe a mechanism – called Index Tags – to avoid false positives, and a suitable format for the supporting-FAs and two auxiliary data structures to improve the matching speed.

4.1 Index Tags

Our initial O^{3}FA engine design may report false positives in the presence of multiple regular expressions. For example, consider a dataset with two regular expressions: {*bc.*d, acd*}. Two input packets P_1:*caaba* and P_2:*cabdc* are received out of order ($P_2 \rightarrow P_1$). Obviously, no matches should be reported on the corresponding input stream *caabacabdc*. However, in our basic O^{3}FA design, the anchored suffix-FA will detect the segment *cabd* of P_2 that matches suffix *c.*d* of the first pattern; when P_1 arrives, segment *cd* will be retrieved and concatenated to P_1 as a suffix, leading to the extended packet *caabacd*. Processing this packet with the regular DFA will lead to the false match *acd* to be reported.

To understand the root cause of this problem, we make the following observation.

Observation 3: Let *R* be a set of regular expressions, *R'* a proper subset of *R*, and *r* a regular expression belonging to *R* but not to *R'*. Let *S* be the set of segments of the input packets that match any prefix or suffix of regular expressions in *R'*. If there exists at least a segment in *S* that also matches a prefix or suffix of regular expression *r*, then a false positive can be reported during processing.

In the example above, let *R* be {*bc.*d, acd*}, and *R'* be {*bc.*d*}. We observe that segment *cd* of P_2 matches pattern *bc.*d* in *R'* as well as pattern *acd* that belongs to *R* but not to *R'*. This fact leads to the false positive indicated above.

Based on this observation, in order to eliminate false positives, we must correlate the matched suffixes and prefixes with the corresponding regular expressions. To this end, we assign an *index tag* to each regular expression, and associate these index tags to the corresponding accepting states within regular and supporting FAs. During processing, we store the index tags associated to all traversed supporting-FA accepting states in a *tag list*. When the regular DFA reports a match, if the index tag of the matched regular expression is in the tag list, then the match is valid; otherwise, it is a false positive. Consider the example above; let *tag1* and *tag2* be the index tags of patterns *bc.*d* and *acd*, correspondingly. When the prefix *cabd* of P_2 is detected to match

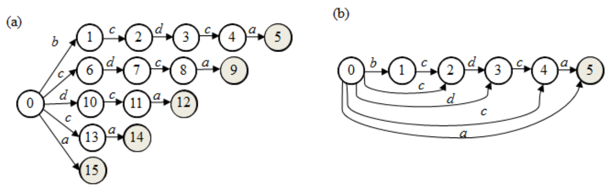


Figure 3. (a) basic asNFA and (b) optimized csNFA built upon anchored suffix set {*bcdca, cdca, dca, ca, a*}. Accepting states are colored gray.

suffix *c.*d* of the first pattern, *tag1* is pushed in the tag list. After the extended packet *caabacd* is processed against the regular DFA, the match of pattern *acd* will be discarded as false positive, since the index tag *tag2* is not in the tag list.

4.2 Compressed Suffix-NFA

As mentioned above, the supporting-FAs may be represented either in NFA or in DFA form. We recall that NFAs are compact but may suffer from multiple concurrent state activations, which may negatively affect the processing time. On the other hand, DFAs have the benefit of a single state activation for each input character at the cost of a potentially large number of states, affecting the memory space required to encode the automaton. In this section, we point out the most effective representation for each of the supporting-FAs.

We recall that, in our O^{3}FA design, the anchored suffix set contains only exact-match patterns. An NFA containing only anchored exact-match patterns can have only one active state. Thus, the anchored suffix-FA can be left in NFA form without loss in processing efficiency. We denote this automaton as **a**nchored **s**uffix-**NFA** (**asNFA**).

The anchored suffix set can have a large amount of redundancy due to the nature of suffixes. An *n*-character pattern can lead to *n-1* suffixes, with every two adjacent suffixes differing in only one character. This creates compression opportunities for asNFA. We propose a **c**ompressed **s**uffix-**NFA** (**csNFA**) representation, which reduces both the asNFA size and bandwidth requirements. Specifically, given the nature of the suffixes of any given pattern, we merge the asNFA states and transitions starting from the tail states. Figure 3 shows an example. Figure 3(a) is the asNFA built upon anchor suffix set {*bcdca, cdca, dca, ca, a*}; Figure 3(b) is the corresponding csNFA. In this example, the compression reduces the number of NFA states from sixteen to six and removes six transitions.

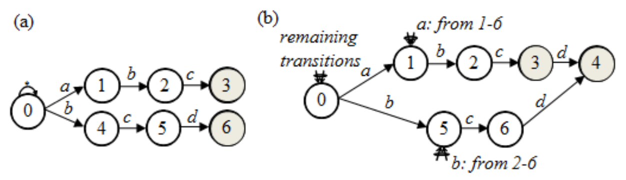
While being more compact, csNFA requires a more elaborate segments retrieval procedure. In an asNFA, segments retrieval can be done by simply tracking back from the recorded matching states to the entry state. However, in the optimized csNFA, this straightforward approach does not work since the backtracking may lead to ambiguity at some states. To address this problem, during csNFA traversal we identify all states that are active after the processing of the first input character and assign state pair <*start_state, end_state*> to each active state, where *start_states* are these active states and *end_states* are last states of the traversed paths originating from them. Only state pairs <*start_state, end_state*> such that *end_states* are accepting states are significant; moreover, as we discussed in Section 3, only the state pair representing the longest matching path needs to be recorded. The matched segment can then be retrieved by tracing the csNFA matching path using the recorded state pair. Since the anchored suffix set includes only exact-match patterns, the *start_states* set has a limited size and the active paths are expected to go dead after the processing of a small number of input characters, reducing the amount of processing. Our experiments in Section 6 confirm the efficiency of this proposed compression scheme.

As an example, we consider the csNFA of Figure 3(b) and input *cadc*. csNFA processes the first input as $\{0\}\xrightarrow{c}\{2,4\}$; we assign state pairs to both active states and track the traversal: $\{2\}\xrightarrow{a}\{\emptyset\}$, $\{4\}\xrightarrow{a}\{5\}$. Since the first path goes dead and the second path reaches the tail state of the csNFA, the traversal leads to two state pairs: <2,2> and <4,5>. Since only state 5 is an accepting state and <4,5> matches the longest segment, only state pair <4,5> needs to be recorded. State pair <4,5> should then be back-traced as $5\xrightarrow{a}4\xrightarrow{c}0$, leading to the retrieval of segment *ca*.

4.3 Prefix- and Suffix-DFA with State Map

We recall that all patterns in the prefix set and unanchored suffix set are unanchored (that is, they may be matched at any position of the input stream). Since their entry state is always active (potentially leading to the concurrent activation of multiple NFA branches), NFAs accepting unanchored patterns tend to have multiple concurrent active states, which negatively affect the processing time. By requiring a single state activation for each input character processed, a DFA representation guarantees minimal processing time, potentially at the cost of a larger memory requirement. However, we recall that patterns in the prefix and suffix sets do not have wildcard repetitions, and thus do not lead to significant state explosion. Thus, the DFA format is suitable for both prefix- and unanchored suffix-FAs; we denote these automata as **p**refix-**DFA** (**pDFA**) and **s**uffix-**DFA** (**sDFA**).

The number of states in a DFA can be minimized through a well-known procedure [4]. In addition, as discussed in Section 3, all prefix matches occurring in the middle of packets can be ignored. This allows further optimizations to the prefix-DFA. Specifically, all accepting states that do not present a self-loop can be made non-accepting, and all self-loops can be removed from the remaining accepting states. This simplification can both reduce the size of the prefix-DFA and simplify the processing (by making a filtering step to remove non-terminal matches unnecessary).



DFA state	1	2	3	4	5	6
NFA states	1	2	3	6	4	5

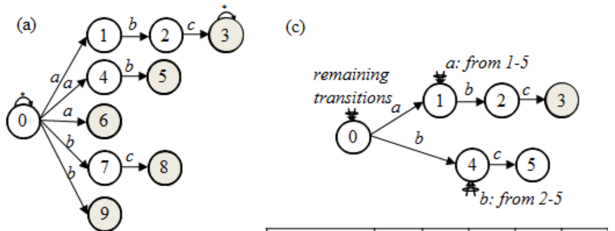
Figure 4. (a) NFA format and (b) equivalent sDFA format with states map for unanchored suffix set {.abc, .*bcd*}. Accepting states are colored gray.**

The use of a DFA representation for these automata, however, has a drawback: it complicates the retrieval of the matching input segments. Since there may be multiple paths leading to the same DFA state, it is not possible to retrieve the input segment solely based on the recorded DFA state. To tackle this problem, we propose using a *state map*, which maps the pDFA/sDFA states to the corresponding NFA states; segment retrieval can then be done by back-tracing NFA paths. Since pDFA and sDFA do not suffer from state explosion, the size of this state map is contained. One DFA state may map to multiple NFA states; in those cases, however, only the NFA state that leads to the longest retrieved segment needs to be included in the state map, allowing a one-to-one mapping.

We illustrate this design through an example. Let us consider pattern *.*abc.*bcd*. The corresponding unanchored suffix and prefix sets are {*.*abc*, *.*bcd*} and {*.*abc.**, *.*ab, .*a, .*bc, .*b*}, respectively. The corresponding automata are shown in Figure 4 and 5. Specifically, Figure 4 (a) and (b) show the unanchored suffix-NFA and the sDFA and state map, respectively. Figure 5(a), (b) and (c) show the prefix-NFA, the reduced prefix-NFA obtained by applying the optimizations discussed above, and the resulting pDFA and state map, respectively. Suppose that the input packet is *bcdbabcdcb*. The traversal of pDFA in Figure 5(c) is: $0 \text{—} b \rightarrow 4 \text{—} c \rightarrow 5 \text{—} d \rightarrow 0 \text{—} b \rightarrow 4 \text{—} a \rightarrow 1 \text{—} b$ $b \rightarrow 2 \text{—} c \rightarrow 3 \text{—} d \rightarrow 0 \text{—} c \rightarrow 0 \text{—} b \rightarrow 4$. The traversed accepting state (state 3) and the final active state (state 4) must be recorded. To retrieve the input segments, we first map those states to NFA states 3 and 7 by looking up the state map, and then back-trace along the NFA. This operation leads to the retrieval of segments *abc* and *b*. Segment retrieval on the sDFA is performed using the same procedure.

4.4 Quick Retrieval Table

Retrieving input segments by back-tracing along NFA paths can be inefficient. To improve efficiency, we propose the use of a *quick retrieval table*, which maps the NFA states directly to portions of regular expressions. This table allows retrieving input segments without back-tracing. A quick retrieval table lookup returns an offset in the relevant regular expression; the input segment can then be extracted directly from the regular expression. This data structure is particularly beneficial in the case of long segments.



DFA state	1	2	3	4	5
NFA states	1	2	3	7	8

Figure 5. (a) original NFA format, (b) optimized NFA, (c) pDFA with states map for prefix set {*.*abc.*, .*ab, .*a, .*bc, .*b*}. Accepting states are colored gray.

As an example, consider regular expression *abcdca*. The anchored suffix set and corresponding csNFA are the same as for the example in Section 4.2. Figure 6 (a) shows the csNFA and Figure 6(b) shows the quick retrieval table, which stores <*index_tag, offset*> pairs. We recall that index tags point to regular expressions. If the recorded state pair is <4, 5>, for example, a lookup in the quick retrieval table will return index tag *tag1* corresponding to pattern *abcdca*, and start- and end- offsets 5 and 6, respectively. This will result in retrieving segment *ca*.

4.5 Functionally Equivalent Packets

Any incoming packet may contain multiple segments that match a prefix or a suffix. For example, the sample packet in Section 4.3 contains two segments that match two different prefixes. All these segments should be recorded and retrieved properly, and all retrieved segments should be processed with the current input packet. A simple approach is to sequentially concatenate the segments retrieved to the current packet and process all modified current packets. For example, supposing that the current packet is *efgh*, using the same example of Section 4.3, then two retrieved segments are *abc* and *b*, and the two corresponding concatenated current packets *abcefgh* and *befgh* should be processed serially. This solution can be highly inefficient. However, concurrently concatenating all retrieved segments to the current packet is not straightforward, since many segments may have overlaps. Our proposed solution is the *Functionally Equivalent Packet* (FEP). The main idea is to construct an alternate packet based on all retrieved segments and then deal with the alternate packet instead of the segments. Such an alternate packet contains all effective information (i.e., all detected segments) of the original packet and thus is functionally equivalent to the original one.

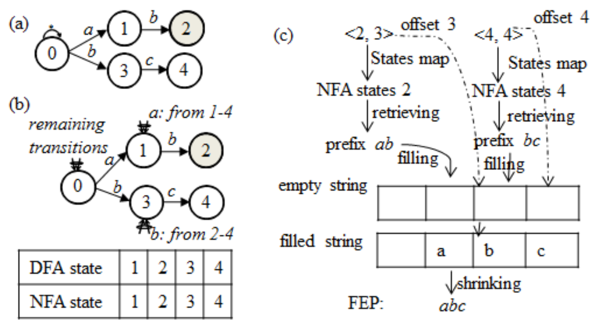


Figure 7. (a) NFA format and (b) equivalent pDFA format with states map built upon prefix set {.*ab.*, .*a, .*bc, .*b}. (c) Construction of functionally equivalent packet to packet P_2=*eabc*.

Consider the example RegEx *ab.*bcd*; the prefix set is {.**ab*.*, .**a*, .**bc*, .**b*}. Figure 7 (a) and (b) are the NFA format and pDFA with a states map for this prefix set. Supposing the input packets are P_1=*afab*, P_2=*eabc* and P_3=*defg*, there is obviously only one matching of *ab.*bcd* across all three packets. Two pDFA states 2 and 4 are recorded after packet P_2 is processed, representing two detected segments that match prefixes .**ab*.* and .**bc*. The retrieved alternate segments are *ab* and *bc*. Directly concatenating both segments to P_3 as *abbcdefg* can cause a false-positive matching. Our FEP design needs only one change that recording <*state*, *offset*> pairs instead of only matched states; the *offset* is the offset of matched segment's last character in the packet. When retrieving segments, all retrieved alternate segments will be filled into an empty string, filling positions accord to their offsets; then, the filled string will be shrunken to get FEP. Figure 7(c) shows the construction of FEP to P_2. <2,3> and <4,4> are two recorded <*state*, *offset*> pairs. The alternate FEP of P_2 is *abc*; substituting P_2 by FEP in the data stream as *afababcdefg* will not affect the matching results.

5. O^{3}FA-BASED SYSTEM

In this section, we describe the design of a regular expression-matching engine based on our proposed O^{3}FA.

5.1 O^{3}FA Engine Architecture

Our O^{3}FA engine consists of four components: (i) a Regular Expression (RegEx) Parser, (ii) a Finite Automata (FA) Kernel, (iii) a State Buffer, and (iii) a Functionally Equivalent Packet (FEP) Constructor.

The **RegEx Parser** operates offline. It first breaks the regular expressions as described in Section 3 and generates the corresponding prefix and anchored/unanchored suffix sets. It then generates the required regular DFAs (rDFA) and supporting-FAs: compressed suffix-NFA (csNFA), prefix-DFA (pDFA) and suffix-DFA (sDFA).

The **FA Kernel** is the operational core of the O^{3}FA engine and performs online packet processing. Specifically, it processes every input packet (possibly extended by the FEP Constructor) against regular and supporting-FAs, and stores the matching state information into the State Buffer.

The **State Buffer** is an auxiliary component that assists both the FA Kernel and the FEP Constructor. This component stores the matching state information generated by the FA Kernel, and it provides information required by FEP reconstruction to the FEP Constructor. As discussed in Section 4, the State Buffer stores: final states from the regular DFA traversal, state pairs <*start_state*, *end_state*> from the csNFA traversal, and <*state*, *offset*> pairs from the pDFA and sDFA traversal. Specifically, the State Buffer stores an entry for each packet processed. During processing, if neither the predecessor nor the successor of the current packet has been previously processed, the current packet is directly processed by the FA Kernel and the resulting matching information is stored in the State Buffer. Otherwise, the FEP Constructor retrieves the predecessor/successor's entry in the State Buffer, and it constructs the FEP of the arrived predecessor/successor packets based upon that state information. The FEP is then concatenated with the current packet, and the modified packet is processed by the FA Kernel. At the end of FA processing, the matching state information is stored in the current packet's entry of the State Buffer, and the related predecessor/successor entries are deleted from the State Buffer since the corresponding information is part of the current packet's entry. Storing

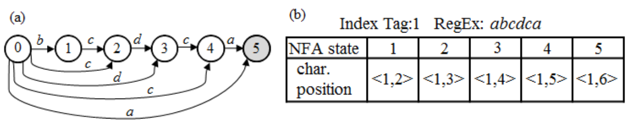

(a)
0 b 1 c 2 d 3 c 4 a 5
c
d
c
a
(b)
Index Tag:1 RegEx: abcdca
NFA state 1 2 3 4 5
char. position <1,2> <1,3> <1,4> <1,5> <1,6>

Figure 6. (a) csNFA and (b) quick retrieval table for anchored suffix set {*bcdca, cdca, dca, ca, a*}. Accepting states are colored gray. Each char. position is a pair of index tag and offset, i.e., <*tag, offset*>

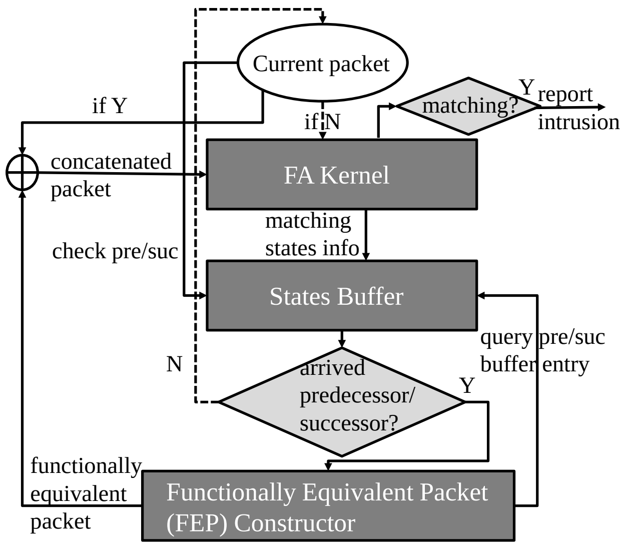

Current packet
if Y
if N
matching?
Y
report intrusion
concatenated packet
FA Kernel
matching states info
check pre/suc
States Buffer
query pre/suc buffer entry
N
arrived predecessor/ successor?
Y
functionally equivalent packet
Functionally Equivalent Packet (FEP) Constructor

Figure 8. Packet processing flow. Dotted arrows indicate alternative paths if there are no arrived successor/predecessor packets.

only state information corresponding to previously processed packets (as opposed to the entire packets) and dynamically clearing entries during processing allow limiting the size of this buffer.

FEP Constructor: The FEP constructor uses state information provided by the State Buffer to reconstruct functionally equivalent packets, as described in Section 4.5.

The packet processing flow is summarized in Figure 8.

6. EXPERIMENTAL EVALUATION

In this section, we provide experimental data to show the feasibility of our O^3FA engine design. Specifically, our experiments are designed to analyze the following aspects: (i) O^3FA memory footprint on reasonably large and complex regular expression sets; (ii) savings in buffer size requirements of O^3FA engine compared to traditional flow reassembly schemes; (iii) memory bandwidth overhead of supporting-FAs; and (iv) O^3FA traversal efficiency.

6.1 Datasets & Streams

In our experiments, we use two real world and six synthetic datasets. The real world datasets contain *backdoor* and *spyware* rules from the widely used Snort NIDS[2] (snapshot from December 2011), and they include 176 and 304 regular expressions, respectively. The synthetic datasets have been generated through the synthetic regular expression generator[3] [29] using tokens extracted from the *backdoor* rules. Each synthetic dataset contains 500 regular expressions. The synthetic *dot-star** datasets contain a varying fraction of dot-star sub-patterns (5%, 10% and 20%); in the synthetic *range** datasets 50% and 100% of the patterns include character sets; finally, the synthetic *exact-match* dataset contains only exact-matching strings.

For each dataset, we generate 16 synthetic traces using the traffic trace generator[3] [29]. This tool allows for generating traces that simulate various amount of malicious activity. This can be realized by tuning parameter p_M, which indicates the probability of malicious traffic. In our experiments, we use four probabilistic seeds and four p_M values: 0.35, 0.55, 0.75 and 0.95, i.e., 16 traces in total for each dataset. All traces have a 1 MB size. Experimental data of each dataset discussed below based on the average performance with four traces generated using different probabilistic seeds.

6.2 Packet Reordering

To simulate out-of-order packet arrival, we break each synthetic stream down into multiple packets and reorder these packets. Packet reordering is driven by two parameters: the out-of-order degree *k* and the stride *s*. Parameter *k* indicates the minimum number of arrived packets that are needed for partial stream reconstruction; parameter *s* indicates the maximum stride between two consecutive packets within each group of *k* packets. Any real-world out-of-order packets flow can be re-generated from original flow follow the driving of proper *k* and *s*.

For example, let us assume a stream consisting of eight packets: P_1 to P_8. If we set *k*=2 and *s*=1, then packets are reordered as $P_2 \rightarrow P_1 \rightarrow P_4 \rightarrow P_3 \rightarrow P_6 \rightarrow P_5 \rightarrow P_8 \rightarrow P_7$; if we set *k*=4 and *s*=1, then packets are reordered as $P_4 \rightarrow P_3 \rightarrow P_2 \rightarrow P_1 \rightarrow P_8 \rightarrow P_7 \rightarrow P_6 \rightarrow P_5$; if we set *k*=4 and *s*=2, the packets' order will be $P_4 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1 \rightarrow P_8 \rightarrow P_6 \rightarrow P_7 \rightarrow P_5$. Obviously, *k*=1 and *s*=1 implies natural ordering, while *k*=(packets number) and *s*=1 leads to reverse ordering (in the example, from P_8 down to P_1).

This packets' reordering scheme allows us to characterize how the packet order affects the performance of the O^3FA engine, and to compare the O^3FA engine with the traditional input stream reassembly method. In our experiments, we break each 1MB stream into 16 packets that each has the 64KB standard TCP packet size. We reorder packets of each stream using three parameter settings: *k*=2/*s*=1, *k*=4/*s*=1 and *k*=4/*s*=2.

Table 1. Memory footprint of FA Kernels (MB)

Dataset	FA Kernel			
	Regular multi-DFAs		Supporting-FAs	
	# of DFA	Memory Footprint	# of FA States	Memory Footprint
Backdoor	8	60	4k	0.62
Spyware	10	56	12k	1.35
Dotstar0.05	15	26	26k	3.58
Dotstar0.1	8	60	25k	3.12
Dotstar0.2	14	100	23k	2.76
Range0.5	1	5.6	24k	2.43
Range1	1	5.8	24k	2.05
Exact-match	1	4.7	17k	1.92

6.3 Experiment Results

6.3.1 *O^3FA Memory Footprint*

First, we evaluate the memory footprint of the O^3FA supporting each of the considered datasets. The *backdoor*, *spyware* and *dotstar** datasets include sub-patterns (e.g. dot-stars) leading to state explosion. To limit state explosion, for these datasets we break the regular DFA into multiple DFAs [6] (the number of DFA ranges from 8 to 15). Due to their simplicity, the *exact-match* and *range** datasets can be supported by a single regular DFA. The total number of regular DFA states ranges from 9k to 254k across the considered datasets. We recall that the supporting-FAs are

[2] https://www.snort.org/

[3] http://regex.wustl.edu/

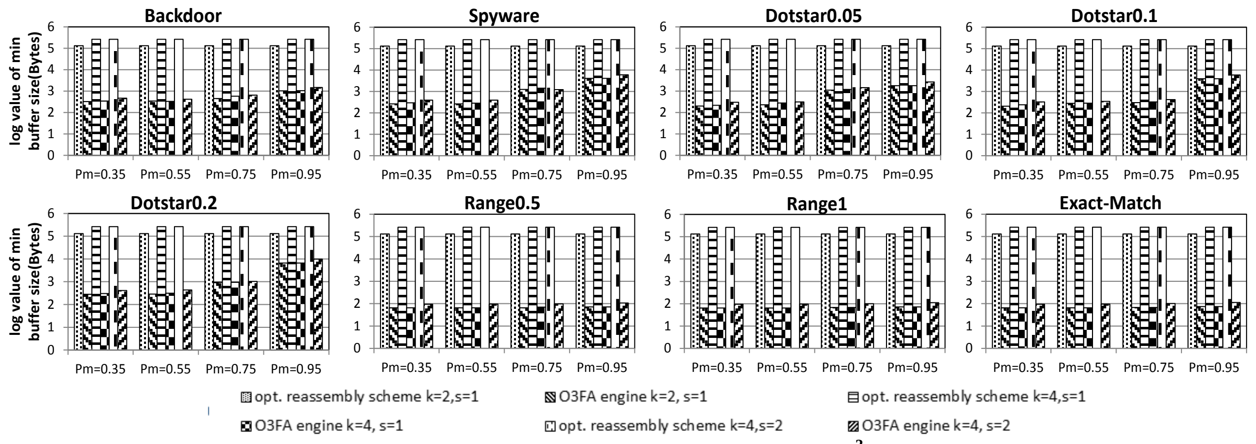


Figure 9. Maximum buffer size requirements for optimized reassembly scheme and O^{3}FA engine on eight datasets. Note that the vertical coordinate is in logarithmic scale.

of three kinds: compressed suffix-NFA (csNFA), prefix-DFA (pDFA) and suffix-DFA (sDFA). As discussed in Section 4, none of the supporting-FAs suffers from state explosion, leading to relatively small automata. The number of csNFA, sDFA and pDFA states ranges from 2k to 13k, from 1k to 13k and from 1k to 9k, respectively. *Range** datasets have larger supporting-FA sizes. This is because all character sets must be exhaustively enumerated before constructing the supporting-FAs, resulting in large prefix and suffix sets. The number of transitions of the csNFA ranges from 5k to 27k. To achieve memory space efficiency, we apply default-transition compression [22] to DFAs. Table 1 shows the estimated memory footprint of the resulting O^{3}FA (we assume 32-bit transitions). As can be seen, O^{3}FA requires about 100MB memory space in the worst case, which does not put pressure to commodity systems. In addition, because supporting-FAs do not suffer from state explosion, their size is in all cases limited, and their memory space overhead is negligible in case of complex datasets including dot-star terms.

6.3.2 Buffer Size Savings

Figure 9 shows the maximum buffer size requirement comparison between the O^{3}FA engine and a optimized flow reassembly scheme adopted from traditional reassembling engine [18]. The O^{3}FA engine uses the state buffer described in Section 5, while the flow reassembly scheme uses a packet buffer. The optimized flow reassembly scheme reassembles a partial stream once the buffered packets allow it and then processes that partial stream and flushes the corresponding packet buffer entries. For each value of p_M, we average the results reported using four probabilistic seeds. In the charts, the six bars represent combinations of the two considered packet processing schemes and the three reordered packet sequences (k=2/s=1, k=4/s=1 and k=4/s=2). In all cases, we report the logarithmic value of the buffer size.

Overall, the O^{3}FA engine with state buffer achieves 20x-4000x less buffer size requirement than does the optimized flow reassembly scheme with packet buffer. We can also see how the packet order and malicious traffic probability affect the buffer size: (i) as could be expected, the degree of packet reordering k affects the packet buffer size, while s does not, and the buffer size has a linear relationship with k; (ii) k has a minor effect on the state buffer size, while s has a major effect on it; (iii) p_M has a major effect on the state buffer size: a higher p_M leads to a larger buffer requirement.

Table 2. Ratio between the number of csNFA states traversed and the number of input characters processed (%)

Dataset	k=2, s=1				k=4, s=1				k=4, s=2			
	P_M=0.35	P_M=0.55	P_M=0.75	P_M=.95	P_M=0.35	P_M=0.55	P_M=0.75	P_M=0.95	P_M=0.35	P_M=0.55	P_M=0.75	P_M=0.95
Backdoor	0.0144	0.0202	0.0278	0.0349	0.0347	0.0337	0.0440	0.1010	0.0144	0.0202	0.0278	0.0349
Spyware	0.0590	0.1002	0.1163	0.1286	0.1158	0.1942	0.2188	0.1853	0.0590	0.1002	0.1163	0.1286
Dotstar0.05	0.0804	0.0838	0.1173	0.2595	0.1733	0.1394	0.1517	0.3927	0.0804	0.0838	0.1173	0.2595
Dotstar0.1	0.0526	0.0715	0.1054	0.2610	0.1129	0.1184	0.1701	0.3974	0.0526	0.0715	0.1054	0.2610
Dotstar0.2	0.0363	0.0611	0.1142	0.2977	0.0531	0.1112	0.1806	0.3622	0.0363	0.0611	0.1142	0.2977
Range0.5	0.0973	0.1015	0.2170	0.2238	0.1839	0.1865	0.3488	0.3831	0.0973	0.1015	0.2170	0.2238
Range1	0.0638	0.1180	0.2181	0.3927	0.1697	0.1910	0.3319	0.6929	0.0638	0.1180	0.2181	0.3927
E-M	0.0391	0.0627	0.1460	0.3140	0.1407	0.1395	0.1959	0.4374	0.0391	0.0627	0.1460	0.3140

Table 3. O^{3}FA traversal overhead compared to conventional stream reassembly methods (%)

Dataset	k=2, s=1				k=4, s=1				k=4, s=2			
	P_M=0.35	P_M=0.55	P_M=0.75	P_M=0.95	P_M=0.35	P_M=0.55	P_M=0.75	P_M=0.95	P_M=0.35	P_M=0.55	P_M=0.75	P_M=0.95
Backdoor	0.0114	0.0102	0.0346	0.3277	0.0211	0.0139	0.0732	0.5140	0.0119	0.0076	0.0288	0.3376
Spyware	0.0059	0.0058	0.1333	2.4362	0.0101	0.0090	0.2635	3.6701	0.0057	0.0049	0.0753	2.4427
Dotstar0.05	0.0103	0.0076	0.2645	1.0132	0.0220	0.0389	0.4492	1.5218	0.0134	0.0221	0.2679	1.0135
Dotstar0.1	0.0041	0.0129	0.0116	2.2671	0.0120	0.0304	0.0183	3.3866	0.0073	0.0136	0.0111	2.2464
Dotstar0.2	0.0083	0.0092	0.0160	3.4655	0.0164	0.0173	0.0225	5.2268	0.0098	0.0101	0.0112	3.4838
Range0.5	0.0007	0.0011	0.0032	0.0137	0.0017	0.0020	0.0054	0.0214	0.0009	0.0012	0.0033	0.0128
Range1	0.0006	0.0011	0.0033	0.0123	0.0014	0.0020	0.0051	0.0153	0.0008	0.0012	0.0033	0.0102
E-M	0.0006	0.0011	0.0033	0.0168	0.0014	0.0022	0.0054	0.0214	0.0008	0.0012	0.0033	0.0159

These effects can be explained as follow. First, since the considered flow reassembly scheme flushes the packet buffer entries after partial stream reconstruction, the packet buffer size is affected only by the minimum number of packets required for partial reassembly, which is controlled by parameter *k*; on the other hand, the size of the state buffer is affected by the number and size of non-empty buffer entries, which are related to the detected segments and the arrived predecessors/successors. The former is affected by p_M, while the latter is affected by the stride parameter *s*. Specifically, *k*=2 and *k*=4 lead to two and four packets being buffered before partial reassembly, while *s* does not affect this number; thus, *k*=4/*s*=1 and *k*=4/*s*=2 lead to the same packet buffer size, and to twice the packet buffer size than the *k*=2/*s*=1 case. However, *s* affects the arrival order of the predecessor/successor of the current packet, thus affecting the size of the state buffer. *s*=1 and *s*=2 lead to two and three entries required (one for previous groups of *k* packets, the others for packets out of current *k* packets having neither a predecessor nor a successor), respectively; thus, *k*=2/*s*=1 and *k*=4/*s*=1 lead approximately to the same state buffer size requirement, while *k*=4/*s*=2 leads approximately to a 1.5x larger state buffer. Because a higher probability of malicious traffic leads to the possible detection of more packet segments by supporting-FAs, resulting in more matching state information being stored in buffer entries, a larger p_M can lead to an increased state buffer size requirement.

6.3.3 Memory Bandwidth Overhead

While in traditional solutions the memory bandwidth requirement of the regular expression matching engine is dominated by the processing of the regular DFAs, O^{3}FA engines have a memory bandwidth overhead due to the processing of supporting-FAs. In particular, while DFA components add a single state traversal (or memory access) per input character, NFA components can potentially have a more significant effect on the memory bandwidth requirement. Table 2 shows the ratio between the number of csNFA states traversed and the number of input characters processed. As can be seen, this ratio is generally small (well below 1), leading to limited memory bandwidth overhead. This small number of NFA state activations can be explained as follows: since the csNFA is anchored, most state activations will die after processing a small number of input characters.

6.3.4 Traversal Overhead

Because of FEP processing, the O^{3}FA engine may process more characters than inputs. These additional characters processed bring traversal overhead over conventional stream reassembly methods. Table 3 shows the O^{3}FA traversal overhead, expressed as a percentage ratio between the number of extra characters processed and the size of the input stream. As can be seen, the traversal overhead (0.0006%-5%) is small enough to be negligible in practice. In other words, O^{3}FA traversal efficiency is comparable to that of conventional stream reassembly methods.

This traversal overhead is affected by both p_M and the number of packets with a previously processed predecessor/successor. In our experiments, *k*=4/*s*=1 packet sequences have longer FEP lengths than do *k*=2/*s*=1 and *k*=4/*s*=2 sequences. In addition, a larger p_M leads to longer FEPs, since it results in more packet segments being detected by supporting-FAs.

In summary, our experiments have shown that: (i) on datasets consisting of a few hundreds regular expressions with varying complexity, the O^{3}FA memory footprint is typically less than 100MB, and the supporting-FAs size is limited (3.5 MB in the worst case); (ii) O^{3}FA state buffers can be up to 20x-4000x smaller than conventional packet buffers; (iii) the O^{3}FA bandwidth is linear in the number of incoming characters and not significantly affected by the NFA components of O^{3}FA; and (iv) the O^{3}FA traversal efficiency is comparable to that of conventional flow reassembly methods.

7. CONCLUSION AND FUTURE WORK

In this paper we have introduced the O^{3}FA engine, a new regular expression-based DPI architecture that can handle out-of-order packets on the fly without requiring packet buffering and stream reassembly. The O^{3}FA at the core of the proposal consists of regular DFA(s) and supporting-FAs, the latter allowing the detection of matches across packet boundaries. We have proposed several optimizations aimed to improve both the matching accuracy and speed of the O^{3}FA engine. Our experimental evaluation shows the feasibility and efficiency of our proposed O^{3}FA engine.

The main goal of this paper is to demonstrate the O^{3}FA idea and engine design; in the future, we aim to deploy this engine on real hardware. In particular, because the automata in O^{3}FA can operate concurrently, the O^{3}FA engine can be implemented on parallel architectures such as FPGAs [30] and GPGPUs [31], potentially leading to higher traversal efficiency. Moreover, since the size of the O^{3}FA state buffer is typically at the KB level, better performance can be achieved by storing this buffer in SRAM (rather than in DRAM).

8. ACKNOWLEDGEMENT

This work was supported in part by the Institute for Critical Technology and Applied Science (ICTAS), an institute dedicated to transformative, interdisciplinary research for a sustainable future (http://www.ictas.vt.edu).

Becchi has been supported by NSF grant CNS-1319748.

9. REFERENCES

[1] J. Newsome, B. Karp, and D. Song, "Polygraph: automatically generating signatures for polymorphic worms," in IEEE Symposium Security and Privacy, *2005*.

[2] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in Proc. of CCS 2003.

[3] Y. Xie, *et al.*, "Spamming botnets: signatures and characteristics," in Proc. of SIGCOMM 2008.

[4] R. M. J. Hopcroft, and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*: Addison Wesley, 1979.

[5] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching," in Proc. of ISCA 2006.

[6] F. Yu, *et al.*, "Fast and memory-efficient regular expression matching for deep packet inspection," in Proc. of ANCS 2006.

[7] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in Proc. of CoNEXT 2007.

[8] S. Kumar, *et al.*, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in Proc. of ANCS 2007.

[9] R. Smith, *et al.*, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in Proc. of SIGCOMM 2008.

[10] M. Becchi and P. Crowley, "Extending finite automata to efficiently match Perl-compatible regular expressions," in Proc. of CoNEXT 2008.

[11] X. Yu, B. Lin, and M. Becchi, "Revisiting State Blow-Up: Automatically Building Augmented-FA While Preserving Functional Equivalence," in JSAC, vol. 32, pp. 1822-1833, 2014.

[12] V. Paxson, "End-to-end Internet packet dynamics," in Proc. of SIGCOMM 1997.

[13] J. Sharad, *et al.*, "Measurement and classification of out-of-sequence packets in a tier-1 IP backbone," in Proc. of INFOCOM 2003.

[14] S. Dharmapurikar and V. Paxson, "Robust TCP stream reassembly in the presence of adversaries," in Proc. of USENIX Security Symposium 2005.

[15] T. Ptacek and T. Newsham, "Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection," *Secure Networks, Inc. Technical Report,* 1998.

[16] A. E. Saldinger, J. Ding, and S. K. Sathe, "Method and apparatus for ensuring ATM cell order in multiple cell transmission lane switching system," US Patent, 1999.

[17] A. S. J. Chapman and H. T. Kung, "Method and apparatus for re-ordering data packets in a network environment," US Patent, 2001.

[18] A. V. Rana and C. A. Garrow, "Queue engine for reassembling and reordering data packets in a network," US Patent 2004.

[19] M. Zhang and J.-b. Ju, "Space-Economical Reassembly for Intrusion Detection System," in *Information and Communications Security*. vol. 2836, ed: Springer Berlin Heidelberg, 2003, pp. 393-404.

[20] X. Chen, *et al.*, "AC-Suffix-Tree: Buffer Free String Matching on Out-of-Sequence Packets," in Proc. of ANCS 2011.

[21] T. Johnson, S. Muthukrishnan, and I. Rozenbaum, "Monitoring Regular Expressions on Out-of-Order Streams," in Proc. of ICDE 2007.

[22] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in Proc. of ANCS 2007.

[23] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," in Proc. Security and privacy in communication networks, 2008.

[24] J. Patel, A. X. Liu, and E. Torng, "Bypassing Space Explosion in High-Speed Regular Expression Matching," in TON, vol. 22, pp. 1701-1714, 2014.

[25] S. Kumar, *et al.*, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in Proc. of SIGCOMM 2006.

[26] D. Ficara, *et al.*, "An improved DFA for fast regular expression matching," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 29-40, 2008.

[27] G. Varghese, J. A. Fingerhut, and F. Bonomi, "Detecting evasion attacks at high speeds without reassembly," in Proc. of SIGCOMM 2006.

[28] R. Smith, C. Estan, and S. Jha, "XFA: Faster Signature Matching with Extended Automata," in Symp. Security and Privacy, *2008*.

[29] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in Proc. of IISWC 2008.

[30] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns," in Proc. of FPL 2003.

[31] X. Yu and M. Becchi, "GPU acceleration of regular expression matching for large datasets: exploring the implementation space," presented at the Proc. of CF 2013.