2016 IEEE 36th International Conference on Distributed Computing Systems

A De-compositional Approach to Regular Expression Matching for Network Security Applications

Eric Norige Alex Liu
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824
Email: {norigeer,alexliu}@cse.msu.edu

***Abstract*—Regular expressions are a very common tool for network security applications because they can match precisely and maintain high matching speed even for many simultaneous patterns. Their core feature is efficient representation as an automaton, where much of the interaction between patterns can be pre-computed and aggregated. Many algorithms have been devised to try and improve this pre-computation to not take exponential space while keeping high performance, but none has met all the requirements of fast, automated construction, small memory image, and high matching speed. We present Match Filtering, a technique for de-composing regular expressions into segments that can be matched independently, while a stateful post-processing engine filters these matches to eliminate those that do not correspond to matches of the original regular expression. Using standard CPU instructions, the post-processing engine can more efficiently represent constructs that would require a multiplicative increase in automaton states. Because the pre-processing is simple, automaton construction can be automated and fast, and because most on-line processing is done by a DFA, its matching speed is close to that of a DFA alone. We demonstrate experimentally 30x smaller, fast (seconds, not minutes) automaton construction and 43% faster matching speeds than state-of-the-art software algorithms.**

I. Introduction

A. Motivation

Deep Packet Inspection (DPI) is a very powerful tool for network security applications to accurately detect malicious network traffic. By inspecting the payload of packets, security applications can block worm traffic that contains remote exploits, a task that is not possible to do well by inspecting packet headers, as network worms have no fixed source, destination or other header. Blocking at the network level is much faster and more efficient than patching endpoints, so it is a favored tool of security administrators.

The category of DPI with the best cost-vs-accuracy tradeoff is regular expression (regex) matching, because it is simple enough for efficient implementation but complex enough to precisely specify attack patterns. A major benefit of regex matching for security applications is the availability of offline pre-processing that greatly speeds online matching of packets. The regex patterns supplied by the user can be pre-processed into a finite automaton that will search for all the patterns simultaneously. This finite automaton representation allows fast processing of packet data by using a set of pre-computed rules to update a single integer state on processing each input character. When this state value is set to one of a pre-determined collection of states, the automaton has found a match, and a corresponding match id is reported.

The foundation for almost all security regex matching solutions is the NFA, a simple structure that keeps a set of "active states" with rules to change which states are active after processing each input character. The downside of the NFA is that the time to process each input character varies with the number of active states, and since the rules are quite irregular, compact organizations of the rules are slow to search. The problems of searching the rules are fixed by converting the NFA into a DFA, which always has a single active state and has a single transition rule for each pair of input character and state. The DFA is constructed by exploring all reachable combinations of active states and pre-computing the result of the NFA transition on each.

Because of the regular structure of a DFA transition table, identifying the transition to apply can be done by direct indexing, but the conversion can potentially exponentially increase the number of rules, as the DFA may need a distinct state for every subset of NFA states. This state explosion makes constructing and storing the resulting DFA impractical for even small numbers of patterns. One of our test cases contains only 11 short regular expressions, but takes 250MB of memory to store as a DFA and 0.1MB to store as an NFA. There appears to be a fundamental tradeoff between the complexity of each transition and the total memory size needed to store the transition function. The goal of this work is to present a novel automaton structure that offers fast processing without requiring a large transition table memory.

B. Limitations of Prior Art

Many papers have attacked the transition table problem, encoding the transition structure of an NFA in different ways to save memory while keeping transition cost low. The most promising solutions largely keep the DFA model of processing, guaranteeing a single active state at a given time, this state

1063-6927/16 $31.00 © 2016 IEEE
DOI 10.1109/ICDCS.2016.63

determines which rules apply. These solutions augment the automaton state with a memory that can be updated as the automaton processes packet data. The existing solutions either have complex automaton transitions or cannot be practically constructed. Extended-FA [8] and HASIC [17] use transitions that check the state of memory to determine the action to take. This makes finding the correct transition much more difficult, as direct lookup of the transition is not practical when the auxiliary memory has more than a few bits, resulting in slower performance. On the other hand, XFA [24] has major issues with automatic construction, requiring significant manual intervention and computational resources to construct XFA automata from novel patterns. The use of EIDD tries to guarantee efficient implementation on an arbitrarily-capable processor, but it often completely prevents the compilation of new patterns. Compiling a new pattern can require significant human intervention to add new constructs to the EIDD tables, which allows the algorithm to start its sometimes hours-long search for an efficient implementation of the non-deterministic update function it constructed.

Id	Regex	# Qs
R1	`vi.*emacs|bsd.*gnu|abc.*mm?o.*xyz`	106
R2	`emacs|gnu|xyz|vi|bsd|abc|mm?o`	23

TABLE I: Related regular expressions and # DFA states

	`vi.emacs..gnu.bsd.gnu..abc.mo.xyz`							
R1		1			2			3
R2	4	1	2	5	2	6	7	3

TABLE II: Matches for R1 and R2 on given string

Match-id: Action
4: Set bit 0
1: Test bit 0 to Match
5: Set bit 1
2: Test bit 1 to Match
6: Set bit 2
7: Test bit 2 to set bit 3
3: Test bit 2 to Match

TABLE III: Filter engine to turn R2 matches into R1 matches

C. Proposed Approach

In this paper, we propose Stateful Match Filtering, a decompositional approach to matching regular expressions. The core idea is that by decomposing a complex pattern into simpler patterns we can post-process the matches of the simpler pattern to get the match results of the complex pattern. The example in Table I shows two regular expressions R1 and R2. While these regular expressions are similar in size and content, the number of DFA states needed to represent R1 is four times as large as for R2. The results of matching these two patterns on an input string are shown in Table II, with numbers in each position indicating which part of each regex matched at that position. R1 matches on the `emacs`, on the second `gnu`, and on the `xyz`, while R2 matches at those positions and at a number of other positions. Because the matches for R2 are a superset of the matches for R1, by filtering the extra matches we could use the DFA for R2 to match the patterns in R1.

A stateful filter to remove the extra matches is shown in Table III. Each match-id that arrives at the filter triggers a very simple program that can examine and modify a few bits and decide whether to match. Note that a stateless filtering would not be able to produce correct results, as match 2 is returned by R2 twice, and in one case, it must be filtered, and in the other it must be permitted.

Applying the filter in Table III to the matches resulting from R2 being applied to the input string, the filter engine first runs action 4 and sets bit 0. Since the action does not allow matching, this first match-id 4 is filtered. Then it takes action 1, tests that bit 0 is set, and since it is, reports a match. It then takes action 2, which tests if bit 1 is set. Since the filter's memory is initialized to 0, bit 1 is not set, so it does not match at this point. Next, Action 5 sets bit 1, and then action 2 again tests bit 1 and allows the match this time. Finally, action 6 sets bit 2, action 7 checks that bit 2 is set and sets bit 3, and action 8 checks bit 3 to allow the final match.

D. Challenges and Proposed Solutions

There are three main challenges to make this kind of solution workable. The first is to achieve fast, automated generation of automata. If the solution requires hours of computation or difficult adjustment by the system operator to be usable, it is much less useful. To overcome this challenge, we give an algorithm for automated transformation of the regular expressions. These transformations introduce additional match ids and the corresponding filter engine rules automatically. This allows most of the generation time to be spent doing standard DFA construction.

The second challenge is to identify an implementation of match filtering that is flexible, efficient and will scale with a large number of match ids to filter. By making the match filtering too powerful, it can become a bottleneck in the solution and prevent high-performance matching. If the match filtering is too weak, it will not be able to scale to handle many complex patterns. We develop a very simple computational model that allows efficient implementations in both hardware and software, and can scale to large numbers of complex patterns.

The third challenge is to achieve correct behavior of the composite system. As there is information lost when splitting the initial regular expression, we have to ensure that the resulting composite system returns the same matches as the original regular expression would find. To do this, we analyze the structure of the regular expression to ensure we do not decompose in ways that would violate the result's correctness. As we build upon a DFA-based matching engine, this means we are easily able to guarantee correct matching of all input

regular expressions in all scenarios at the cost of not being able to eliminate some state explosion. Systems that use only string matching as a primitive can easily lack this property or can have much lower throughput on certain classes of regular expressions due to having to re-implement all subtleties of regex semantics.

E. Key Contributions

We make three contributions:

1) We propose Match Filtering, the first practical solution to security regex matching that is based on stateful filtering of match results from simplified versions of the original patterns.
2) We develop an prototype implementation of decomposition algorithms for construction and filter matching automata engines for pattern matching packet traces.
3) We evaluate our match filtering prototype on a variety of patterns and traffic and compare against the state-of-the-art software solutions.

II. Related Work

We divide related work in three categories, software/general regex matching, TCAM-based regex matching and FPGA-based regex matching. Software/general matching does not assume any specific hardware support, and depending on the algorithm are software-only, or could be implemented on specialized hardware. TCAM-based regex matching uses Ternary Content Addressible Memory, a very fast pattern-searching engine to store and search the automaton's transition table. FPGA-based regex matching uses the flexibility of FPGA to represent the regex pattern as a logic circuit and then process the input using this circuit.

A. Software/general matching

A major variety of software-based matching is deterministic automata extended with additional memory. Two major examples of this are XFA [24] and HFA [15]. HFA extends DFA with auxiliary memory and allows transitions to test and modify the memory. While the HASIC paper [17] made HFA much more efficient, it did not overcome the limitations of having complex transitions that depend on the state of memory. This limitation makes the automaton size larger and makes processing slower. XFA is more similar to Match Filtering Automata, where entering a state could modify the memory and potentially return a match if the memory state met some conditions. Fundamentally, XFA and Match-filtering are different because of the construction method; XFA's construction method is byzantine, requiring a search of a complex expression space to determinize a non-deterministic state update function. Match filtering brings significant improvements in constructability, being automatically constructible in very little time.

Another approach taken by Yu *et al.* [26] and Becchi *et al.* [6] is to produce automata that are guaranteed to have a fixed or bounded number of active states. While these can greatly reduce memory use by allowing multiple active states, using just 2 active states reduces their throughput to 50% of a DFA engine while significantly hampering their compression ability. The matching of compressed traffic is tackled by Bremler-Barr *et al.* [11] and Becchi *et al.* [5], and allows fast search of the contents of GZIP compressed flows without decompression. The techniques they develop can apply to this problem as well, allowing compressed streams to be matched against more complex patterns without needing exponential storage.

The technique that is most similar to ours is that used by Snort [22], which uses an Aho-Corasic-based string-matching engine as a pre-filter to limit what patterns are searched for. Each Snort rule that needs regex matching can have a series of "content" strings that are searched for in the primary pass. By eliminating most regular expressions from consideration through this pre-filtering, the number of simultaneous patterns that need to be matched is greatly reduced. The downside of this technique is that it requires multiple passes over the input content, increasing the total amount of work done and requiring more buffering.

B. TCAM-based matching

While TCAM-based solutions like [12], [18], [21] are very effective at storing the transition table of deterministic automata in a compact way in a TCAM chip, these solutions have two main limitations. The first limitation is that the TCAM chip is very power hungry, which makes these solutions difficult to implement in cost-efficient ways. While small TCAMs can be more efficient than large ones, large TCAMs are needed to be able to handle large pattern sets. Secondly, the search done by TCAM chips is inefficient; each TCAM query causes a highly parallel search of every entry in its memory. While using TCAM may give high throughput, the check-every-possibility approach of TCAM search allows a solution relying on brute-force to be mistaken for an efficient one.

C. FPGA-based matching

There are many FPGA-based solutions for pattern matching [4], [10], [13], [14], [16], [19], [23], [25]. These are able to leverage parallelism of hardware to achieve higher throughput matching, and can take advantage of being able to reprogram the FPGA to adapt to new patterns. The disadvantage of these is two fold: First, reprogramming the FPGA is not easily automatable, and there are many reasons it can fail, making updates more difficult. Second, these solutions generally have problems multiplexing a large number of simultaneous flows, as needed in high-end network security applications. This is because they embed the matching state deep in the operation of the circuit, where it is hard to save and load, and even then, they can have large matching state, over 1Kbit for [4].

When the problems of FPGA or TCAM-based matching are not significant for a particular application, they can be used to implement the de-composed regex matching needed in match filtering, as match filtering is built on top of an arbitrary regex matching solution.

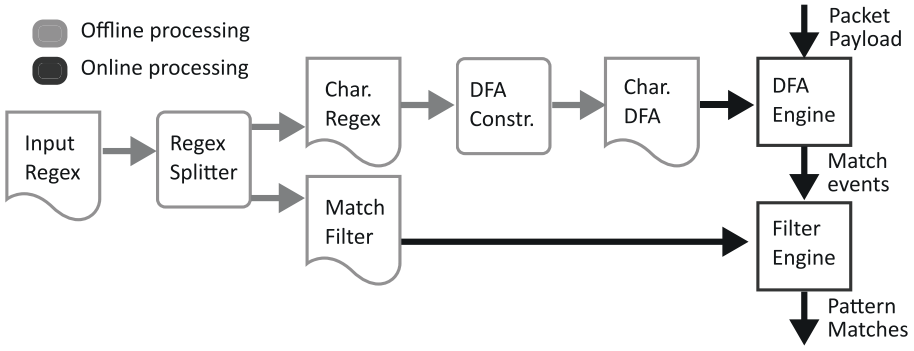

Fig. 1: Match filter generation and use

III. Match Filtering Automata

To construct a match filtering automata, the input pattern is decomposed in such a way that the parts efficiently handled by the DFA engine are separated from those parts that do not have an efficient DFA implementation. After de-composing a complex regular expression into simple pattern matches, a filter engine can post-process the matches of these simple patterns to determine whether any complex pattern was matched. The more types of regular expressions that can be de-composed in this way, the more patterns that can be efficiently implemented.

In this section, we will first give a formal specification of match filtering automata. Then we will lay out the the processing model under which an automaton will be presented with packet data to produce matches. Finally, we will give high level details of the construction process that will be further expanded in Section IV.

A. Definition

A Match-filtering automaton can be defined as a 9-tuple $(Q, \Sigma, \delta, q_0, D_i, D_q, w, D, f)$ Intuitively, the first six components define a DFA that processes input characters and identifies possible matches. The last three components define the stateful match filtering component that determines which of the DFA matches are matches for the original patterns. The DFA is over alphabet Σ with state space Q, start state q_0, transition function $\delta : Q * \Sigma \rightarrow Q$, decision set D_i and decision function $D_q : Q \rightarrow 2^{D_i}$ which indicates for each state which match-ids, if any, are found. Note that D is the final decision set, and D_i is D augmented with new match-ids that will always be filtered. The match filtering component has a w-bit memory, which can be represented as the set of states $M = 2^w$. It also has a filtering transition function $f : M * D_i \rightarrow M * \{Confirm, Drop\}$. Each time the DFA produces a possible match, the match filter processes that match against the memory, creating a new memory state and a decision of whether to confirm or drop that match. For all inputs where the match-id is not an element of D, the match filter must $Drop$ the match. By convention, the initial value of the memory is all zeros.

Note that this model is intended to describe a general pattern of many possible match filtering automata and not only the specific match filtering automaton construction evaluated in this paper. It places no restriction on the complexity of the filtering function f. An efficient implementation of f should inspect a small set (possibly only one) of these memory bits to determine which few bits to modify and whether to return a match. It is also preferred to have a smaller w to reduce the amount of memory that each processing context will need in an environment that simultaneously processes many flows.

To make it easy to describe a particular f, we will simply specify a pseudocode action for each D_i. For example, an match with action only may be written "2a: set 4." This means that the Filter Engine will set memory bit 4 when it receives match-id 2a, and not send a match. A more complex action could be "7: Test 2 to increment 8–12 and match." When receiving a match id 7, this filter would do nothing if bit 2 is not set. If bit 2 was set, it would increment the counter in bits 8–12 and report a match. Using a single action for each match-id allows efficient action lookup, leveraging the multi-match capability of the underlying DFA to trigger many actions as necessary, although this requires an efficient hand-off between DFA and filter engine, as the filter engine will be triggered more often. Further, there is an ambiguity in the behavior of the automaton under these multi-match conditions, as the order of processing the actions can lead to different behavior. While our construction avoids this undefined behavior, it must be accounted for in each implementation of this model.

We will not give a more specific definition of what is allowed in actions, as the only real limit is what can be efficiently computed by whatever hardware this automaton runs on. This means that for an ASIC implementation, the actions will need to have a structure that is implementable efficiently in binary logic, so as to be small and fast, but can have significant parallelism without much cost. For a software implementation, a few CPU instructions is usually sufficient to give great flexibility, although more complex processing is not forbidden if it leads to potentially useful in some circumstances.

B. Matching process

The general flow of matching is illustrated in Figure 1, where the black lines and boxes represent data flows and

processing engines active for this online processing. The packet payloads are sent to the DFA engine, which reads from the Character DFA and sends match events to the Filter Engine. When a match-id arrives at the Filter Engine, it looks up the corresponding action, runs that action to update its state and potentially permits a match to pass through it.

Given a filter automaton constructed for a set of patterns and a flow payload to inspect, the process of matching those patterns is more formally as follows. First, the DFA state q and memory m are initialized to $(q_0, 0^w)$. For each byte of payload c, the DFA is advanced to its next state by its transition function, $q := f(c, q)$. Then, possible matches are identified by looking up $D_i(q)$ and processing the actions for each match. For each action function f, we compute $(m, pass) := act(m)$, updating the memory and reporting a match at the current position if $pass == Confirm$. After all actions are processed, DFA processing continues at the next input character. To handle many flows arriving in multiplexed fashion, all that is necessary is to keep a (q, m) pair for each flow, as all parsing state is kept therein. Note that the DFA processing and the match filtering do not have to be done in lock-step, but the DFA processing could put matches with the position of the match into a queue, and the match filtering could read from that queue and report matches as they're found.

C. Construction

The general pattern for constructing the match filter automaton is shown in grey boxes and arrows on the left side of Figure 1. The input regex is first processed by a regex splitter that decomposes more complex regular expressions into simpler ones, returning new regular expressions. These new regular expressions are processed by standard DFA construction rules to produce the Character DFA that will match packet payloads in the filter automaton. The regex splitter also outputs a collection of match filters that capture the relationships between split regular expressions, for use in the Filter Engine. When the input is a collection of regular expressions, each triggering a different match id, the regex splitting can be done separately to each, producing a decomposed regular expression and match filter actions. All these de-composed regular expressions can be combined as the input regexes would be, with a giant n-way union operator. To combine the filters, simply modify the actions so they are globally unique and concatenate the action tables and remove overlaps in memory use.

IV. Regex splitting details

This section describes some common regular expression patterns that cause large increases in the size of DFAs. Better matching engines can be created by de-composing these problematic regexes into simpler regexes whose matches can be easily filtered to produce correct results. We use the notation `{{x}}` as part of a regular expression to indicate that when the prefix of the regular expression before the annotation has been matched, match-id x should be reported. An unmarked set of regular expressions is interpreted as having implicit `{{1}}`, `{{2}}`, etc. appended to each rule. To describe the splitting patterns on general regular expressions, we will use the capital letters `A`, `B`, `C`, etc. to refer to arbitrary regular expressions and lower case letters `a`, `b`, `c`, etc. as characters in the input alphabet. For example, `A{{1}}|B{{1}}` can refer to any regular expression for which matching either `A` or `B` results in match id 1, also written `(A|B){{1}}`.

A. Dot Star

A common pattern in security regular expressions is `.*A.*B{{1}}`, which we call dot-star. It is used to find occurrences of regex `B` after regex `A` has matched. This pattern is capable of causing a multiplicative increase in the number of DFA states when it is added to a pattern set. This is because all DFA states that can be active before starting the match of `A` must have a corresponding distinct state that can become active after matching `A`, doubling the number of states needed. Each pattern with dot-star contributes another multiplicative factor to the size of the final DFA, which can greatly increase its size.

To combat this size increase, we will de-compose this pattern into `.*A{{1a}}|.*B{{1}}`. Adding this decomposed pattern to a pattern set will cause only an additive increase in the number of DFA states, instead of the multiplicative increase caused by the original dot-star pattern. The memory of the match filtering component will be used instead of DFA states to record the successful matching of `A`. The de-composition is valid only provided that following two conditions are met.

First, both match ids $1a$ and match id 1 must be filtered; $1a$ cannot be reported, but must set a bit flag, and match id 1 must be reported only when that bit is set. If we choose to use bit 0 of memory for this filter, we can write the filters compactly as: $1a$: Set 0, 1: Test 0 to Match. Logically, this corresponds to remembering whether pattern `A` has been matched and only matching `B` after such a match.

Second, in order to de-compose `.*A.*B{{1}}`, no suffix of `A` can be a prefix of `B`. For example, if this rule is used to de-compose `.*abc.*bcd{{1}}` into `.*abc{{1a}}|.*bcd{{1}}` as above, the result will incorrectly report a match on input `abcd`. This happens because after matching `abc`, bit 0 will be set, and then `d` will trigger the "1: Test 0 to Match" action and allow the match to succeed, even though it should fail. This problem occurs because the de-composed patterns allow overlap, where `B` begins matching before `A` finishes matching.

This de-composition step can be used multiple times on a single regex: `.*A.*B.*C{{1}}` can be de-composed twice, resulting in `.*A{{1a}}|.*B{{1b}}|.*C{{1}}` with two memory bits used for filtering. In this case, the match filters are: $1a$: Set 0, $1b$: Test 0 to Set 1, $1c$: Test 1 to Match. Similar requirements on `A`, `B`, and `C` exist as before; no suffix of `A` can be a prefix of `B` and no suffix of `B` can be a prefix of `C`. In this way, this de-composition can be generalized to patterns with any number of dot-stars.

B. Almost Dot Star

For IDS pattern sets, an even more common pattern than dot-star is almost-dot-star: `.*A[^X]*B{{1}}`. Regexes in line-oriented protocols like HTTP commonly search for two strings on the same line. This can be done as `.*abc[^\n]*xyz`, which will match `abc` followed by `xyz` only if there's no line break between them. Note that `X` is a character class, not a full regular expression. This pattern can be de-composed to `.*A{{1a}}|.*[X]{{1b}}|.*B{{1}}` under the two conditions that follow. Note that the de-composed regex searches for `X` and not the negated `X` as was present in original pattern.

First, all three match-ids $1a$, $1b$ and 1 must be filtered so that $1a$ and $1b$ are never reported and 1 is only sometimes reported. A single bit of memory will be used to track whether `A` has been matched and `X` has not been matched since then. If this bit is set, it is correct to report matches on pattern 1 as `A` has matched and there hasn't been a `X` since. Specifically the match filters are, $1a$: Set 0, $1b$: Clear 0, 1: Test 0 to Match.

Regex	`.*abc{{1a}}|.*[\n]{{1b}}|.*xyz{{1}}`							
Input	`abc:...\n...:xyz\nabc:xyz\n`							
Raw M's	`1a`	`1b`		`1`	`1b`	`1a`	`1`	
Actions	S	C		T	C	S	T	
Filtered M's							1	

TABLE IV: Raw matches, filter actions and filtered matches of a split regex on given input; Actions are S=Set, C=Clear, T=Test to match

Decomposing the above example, we get `.*abc{{1a}}|.*[\n]{{1b}}|.*xyz{{1}}` with three match filters. When this filter-match automaton is presented with the input string shown in Table IV, it will find the six matches listed as Split M's. The $1a$ pattern matches on the first line (before the first `\n`) and sets the memory bit, but that memory bit is cleared by the $1b$ pattern before the 1 pattern matches in the second line. Only in the third input line does the full pattern match because $1a$ is followed by 1 without an intervening $1b$.

The second condition to be able to de-compose this pattern is similar to the non-overlap requirement of dot-star de-composition. The pattern `A` is not permitted to have a suffix that matches any prefix of `B`. In addition, the characters in `X` cannot appear in `B`, as this would result in the memory bit being cleared during the processing of `B`, preventing all matches. It is only allowed for characters in `X` to be in non-final positions in `A`, as the clearing of the memory bit will be overridden by the setting of that bit when `A` is matched. Characters in `X` being in final positions of `A` would result in the memory bit being set and cleared simultaneously, so we disallow de-composition in this case.

Excessive application of this pattern can result in loss of throughput. Even in a hardware implementation, action processing is not free, so it is important to not cause an excessive number of matches that need to be filtered. Improper application of this pattern can cause excessive numbers of matches, for example, the pattern `.*abc[a-f]*xyz{{1}}` could be decomposed into `.*abc{{1a}}`, `.*[^a-f]{{1b}}`, `.*xyz{{1}}`. Running these patterns on an input would produce match `1b` on every character of the input except for `a-f`, very likely a large proportion of the input. Processing actions for such a number of matches can reduce the throughput of even a very efficient implementation.

The root cause of this problem is the number of matches generated by the second component of the de-composed pattern. If the character class `X` is small, the cost of reporting and filtering matches from `.*[X]{{1b}}` is likely to be low, but if `X` was nearly the entire alphabet, almost every input character would result in matches being generated and processed. Even with a small character class `X`, a similar loss of performance could be triggered by maliciously generated traffic that repeat long strings of characters in `X`. There are mitigation strategies for this problem, such as adjusting the $1b$ pattern to `.*[X]+[^X]{{1b}}`, so that only one match is generated on the first character not in `X` to be found after a sequence of one or more characters from `X`. For our implementation, we choose to instead use a size threshold of 128 characters to determine whether or not to de-compose. This means that if `X` has 128 or more characters in it, we will not apply the almost-dot-star pattern. This rule works well for common security regexes, which use character classes that are small sets or everything but a small set.

C. Using patterns to build MFA

The algorithm to construct an MFA from an input regex is presented in Algorithm 1. The core of this algorithm is the Regex Splitter, a function that takes the input regex and rewrites it into a collection of simpler regexes. These simpler regexes can be turned into a standard DFA to maximize the matching speed of the result. Regex splitting also produces a corresponding collection of match filtering rules that are used to transform the set of matches from the simpler regexes into matches of the original regex. The regex splitting algorithm can be divided into two parts: `RegexSplit` and `Decomp`.

The first part, `RegexSplit`, is largely management code. It orchestrates the decomposition process across all elements of $rxes$ and tracks the usage of auxiliary memory so each decomposed pattern gets different bit positions. Note that as it runs, it removes the de-composed regexes from the original list and appends new regexes to the list. It is important that the new regexes be appended to the list so that `Decomp` can be run on them to further decompose as necessary.

The second part, `Decomp` analyzes the structure of the input regexes and creates a collection of new regexes from pieces of the input regexes. The original match-id, n, is also preserved in the final component of the initial regex, and it will be filtered to only result in a match when the original regex would match.

In the pseudocode, both Dot-star and Almost-dot-star are illustrated. The details of traversing parsed regular expressions to determine if any patterns match and to divide the regex into components are omitted for simplicity. We also omit

the tests for prefix/suffix compatibility, as these are standard algorithms. Once the regex has been divided into pieces, the recipe for the replacement pieces is fixed, so we assemble new regexes in a set manner. Some of these new regexes have new match ids, labeled n' and n'' in the pseudocode. These new match ids will always be match-filtered, but they will be able to trigger changes in the state of the match filter. Similarly, the match filters generated for each pattern follow a fixed scheme, so their generation is straightforward. In the prototype implementation, we also handle anchored regular expressions that must match a specific pattern at the start of the input. To do this, we simply prepend this pattern to each of the de-composed regexes so they only match when that specific pattern is at the start.

To implement Match Filters, we use a simple bytecode with 4 integers specifying the behavior of each action. The first integer specifies the memory index, if any, that must be true for this action to have any effect. The second integer specifies what memory index, if any, to set if this action takes effect. The third integer specifies what memory index, if any, to clear if this action takes effect. Setting and clearing are mutually exclusive, so this could be further compacted to save space, although at the cost of more expensive action processing. The final integer specifies the match-id to report. While the match-id could be replaced with a boolean, this would not have significant gain and would have some cost. Because of memory alignment, the extra space saved by storing just a bool would be wasted, and storing the full match-id makes it slightly easier to track Match Filters.

We use a multi-part bytecode because multiple actions can be generated for a single match-id, so we merge these into a single bytecode that can execute them all. For example, the action "$1a$: Test bit 1 to set bit 2" could result from merging "Test bit 1" with "Set bit 2".

Algorithm 1: Regex Splitter

```
1  Function RegexSplit(rxes)
       Input: Regex-set rxes
       Output: Match filters
       /* Index of next memory bit        */
2      bit_i ⟵ 0;
       /* Set of match filters to return  */
3      match_filters ⟵ {};
       /* Decompose each regex in turn    */
4      foreach rx ∈ rxes do
5          if rx matches some decompose pattern then
6              (split_rx, match_fs) ← Decomp(rx, bit_i);
7              Remove rx from rxes;
8              Append split_rx to rxes;
9              bit_i ← bit_i + 1;
10             Add match_fs to match_filters;
11     return match_filters;
12 Function Decomp(rx, i)
13     if rx has form .*A.*B{{n}} then
14         rxes ← .*A{{n'}}|.*B{{n}};
15         match_fs ← {n': Set bit i,
16                     n: Test bit i to match};
17     else if rx has form .*A[^X]*B{{n}} then
18         rxes ← .*A{{n'}}|[X]{{n''}}|.*B{{n}};
19         match_fs ← {n': Set i, n": Clear i,
20                     n: Test i to match};
21     return (rxes,match_fs);
```

V. Experimental Results

The important properties of a regex engine include

- Automaton size
- Automaton construction time
- Matching throughput

We present comparison results of Match Filtering Automata (MFA) with DFA, NFA, HFA (as represented by HASIC [17]) and XFA [24]. DFA is a baseline to compare against for performance. NFA is a baseline to compare against for construction time and automaton size. HASIC and XFA are the state-of-the-art software algorithms for achieving high performance at small automaton size.

A. Patterns and Traces

To compare different construction costs, each algorithm must be used to construct automata for the same sets of patterns. The patterns we use come from various security applications, and have the number of regular expressions, NFA states and DFA states summarized in Table V. The S-patterns and B-patterns come from Snort [22] and Bro [20], and are publicly available [7]. The C-patterns come from a major networking vendor and are proprietary. The 'p' versions of some pattern sets are constructed by restoring some commented patterns from the original C7, S31 and B217 sets.

In general, the C patterns use dot star and almost dot star patterns heavily, often having multiple per pattern. The S patterns are a mix of many almost dot star and long string matches with a few dot star patterns. The S patterns often have an anchored component, meaning it is expected to match at the beginning of the flow. This makes the matching problem much easier, as if the patterns aren't found at the beginning, they don't have to be searched for later in the flow. The B pattern has many unanchored string matches, with a small number of dot stars mixed in to make things more difficult. Pattern set B217p could not be constructed as a DFA, so its number of DFA states is unavailable.

To compare the performance of various solutions, it is necessary to apply the constructed automata to trace files to identify any matching traffic in them. We use both synthetic and real-life traces. The real-life trace files we use are from the DARPA intrusion detection data set (DP) [1], the Inter-service academy Cyber Defence Competition (CD) [2], and the Nitroba University Harrassment Scenario (N) [3]. The DP traces total 4.1GB and are the Monday, Wednesday and Thursday traces from week 5. The CD traces total 550M

Set	RegExes	NFA Qs	DFA Qs	MFA Qs
B217p	224	2,553	—	5,332
C7p	11	295	244,366	104
C8	8	99	3,786	341
C10	10	123	19,508	81
S24	24	702	10,257	766
S31p	40	1,436	39,977	1,584
S34	34	1,003	12,486	1,499

TABLE V: RegEx set Properties

and are the traces 11–13 and 110–113. The N trace is 60 MB. All these traces are `.pcap` files with packet-level details and not pre-assembled flows. While the DARPA traces are ancient for many purposes, they still show similar performance characteristics to newer traces, so they give further evidence of quality.

The synthetic inputs were created by Becchi *et al.*'s flow generator [9]. This tool takes as input a collection of regular expressions and can create trace files with varying difficulties. For this experiment, we use the commonly tested difficulties p_M of 0.35, 0.55, 0.75 and 0.95. We also test a purely random trace as a baseline for non-matching traffic.

B. Experimental Setup

We measure the construction time by measuring cpu-seconds taken to construct the automaton. To measure automaton size, we determine the amount of contiguous memory needed to store all static transition tables and auxiliary structures needed by the parsing engine. To measure performance, we run the engine on a trace and measure the number of cpu cycles needed to examine the trace, as reported by rtdsc instruction. This number of cycles is divided by the payload size of the packets in the trace to get Cycles per Byte (CpB). As we are not able to construct XFA automata, we present estimated throughput results using the same methodology in [17]. All construction and matching algorithms are implemented in 3700 lines of OCaml 4.02.1, and are run on a i7–4500U CPU, using only a single core.

C. Construction Costs

Pattern	NFA	DFA	HFA	MFA
B217p	0.5	—	108	2.6
C7p	0.1	250	4	0.05
C8	0.1	4	0.8	0.16
C10	0.1	20	2	0.04
S24	0.2	10	6	0.37
S31p	0.4	41	16	0.77
S34	0.3	13	9	0.73

Fig. 2: Memory Image Sizes (MB)

Table 2 gives memory image sizes in megabytes for automata of the various patterns encoded using each algorithm. As expected, the NFA are the smallest, taking 0.5 MB or less for each of the patterns. This is because these automata have very few states and transitions to store. At the other extreme, DFA are the largest; even though each transition is very easy to store, these automata have vastly more states and transitions than other solutions, so require much space. HFA are in the middle, having to store many fewer transitions than the DFA because of the compressed transition structure, but the size of each transition is larger in HASIC, making them far from as compact as the NFA. MFA has very good results in this comparison, achieving a similar image size to NFA and taking an average of 30x less space than HFA. Almost all the memory image bytes used in MFA are for the DFA automaton, with filters taking up an average of less than 0.2% of each image. The total memory image of MFA can be so small because the use of filters reduces the number of DFA states needed effectively without making the transitions any more complicated.

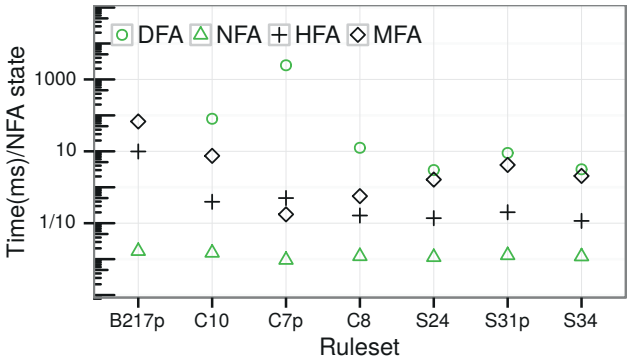

Fig. 3: Construction times for DFA, HFA, NFA and MFA

Construction times are shown in Figure 3, and largely correspond to memory image sizes. NFA construction is by far the fastest, as it has very little work to do even for complex patterns. DFA fails to construct B217p, and takes a relatively long time on each of the other patterns, as it has to spend much work to expand each NFA state into a large number of DFA states. While the total construction time for DFA may often be small, the results for C7p show that DFA construction can be slow even small numbers of regexes because DFA construction depends heavily on the number of DFA states constructed. Even when the regex set has good behavior, adding a single extra regex with multiple dot-stars can increase construction time to many times what it was. HASIC does its construction very quickly, using its approximation to entirely skip critial NFA states and generating its automaton faster than MFA in many cases. Match Filter Automata construction isn't able to construct the final automaton as fast as HASIC because the DFA in MFA has more states than the corresponding HFA. While the additional states require more subset construction time, but MFA's construction time is still orders of magnitude better than plain DFA, as it has eliminated most state explosion.

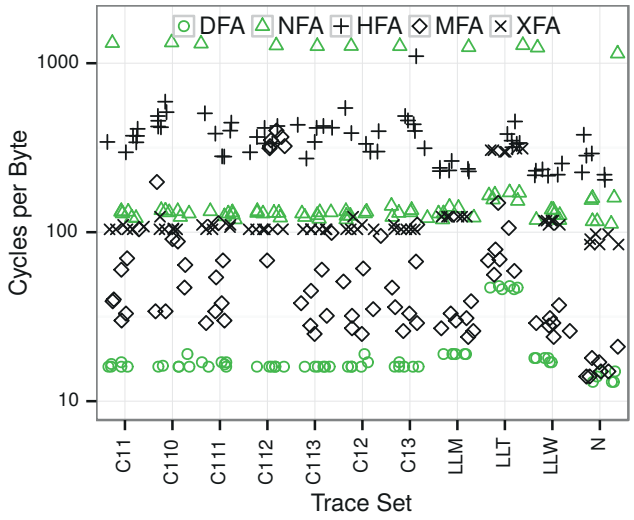

Fig. 4: Throughput measurements in cycles per byte

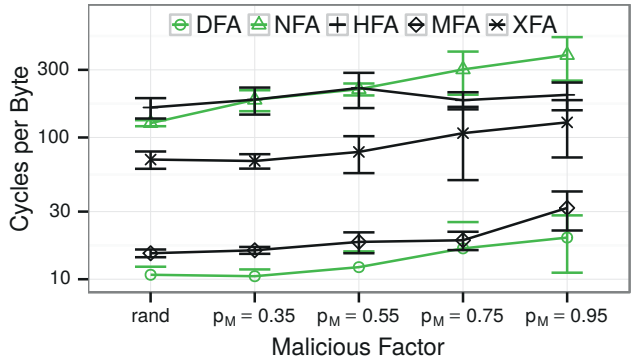

Fig. 5: Synthetic throughput in cycles per byte

D. Processing Throughput

The results of running all patterns against all real-life traces using all algorithms are summarized in Figure 4. Each point represents a single pattern on a single trace using a single algorithm; each algorithm gets its own point shape, and the traces are organized horizontally in no particular order; Cx is CD trace x, LLx is DP trace x, and N is the Nitroba trace. The DFA engine beat all others on matching speed, as when the DFA is manageable, it runs very quickly, with this implementation requiring an average of 19 cycles to process each byte. The variation in throughput that exists in DFA results seems related to the trace, as some traces may have shorter packets, increasing the relative cost of flow reassembly. NFA exhibits a bimodal distribution, with most trace and pattern combinations needing around 130 cycles per byte, but the B217p pattern requires ten times that, over 1300 cycles per byte. As B217p has some very short patterns, so we speculate that the number of active NFA states is about 10 times higher when matching the B217p pattern than others, thus requiring much more processing time.

The memory-augmented automata (HFA, MFA and XFA) have mixed results on the real-life througput tests. HFA does poorly in this test, usually taking an average of 360 cycles to process each byte. This seems to be because the simulation was run in OCaml code which is not as optimized for speed as the C++ simulator used in the HASIC paper. The estimated performance of XFA is slightly better than that of NFA in this test, giving it a mid-range performance result with an average of 125 cycles per byte. For MFA, the performance varies more than any other, but in general, it performs very well on all traces except C112, which it performs quite poorly on, taking an average of 306 cycles per byte. Ignoring this trace's results, MFA takes 49 cycles per byte, an average of 43% faster than XFA. MFA is also the only algorithm that was able to complete the B217p test using an OCaml implementation, using an average of 123 cycles per byte despite the complexities of this pattern.

When measuring our five algorithms on synthetic traces, we can see better how well they stand up to increased numbers of matches. Results of these tests ate shown in Figure 5. Each algorithm has its mean results connected by a line, with NFA and HFA taking the longest at the top of the graph, MFA and DFA at the bottom of the graph and XFA in the middle. For all algorithms, there is some increase in processing time as more matches are injected into the traffic. DFA still performs the best, but its performance falls by 83% between the least and most malicious traffic. Worst is NFA, which starts the slowest and has its performance fall by over 2x over the same range. In the middle, XFA and HFA are mediocre, but neither having as bad a decrease in performance over the variety of traffics as NFA. Finally, MFA does quite well, losing a bit more performance at high maliciousness than DFA. This is due to the extra work that its filtering engine has to do on traffic with many matches. But even with heavily matching traffic, it still has better performance than any other algorithm (except DFA) on any traffic.

VI. Conclusion

The methods presented here are effective at dealing with state space explosion while still being automatically generable, and without producing an overly complex automaton that performs slowly. This work has picked the low hanging fruit; further work can still be done to add more patterns that can be de-composed. Notable among the missing patterns is counting conditions, such as `/abc.{n}xyz/`. These require the filter to count how many times a certain sub-pattern is matched, which is more complex than just keeping a bit flag, but is solvable in the general framework presented here. Also, additional work can be done to to eliminate the restrictions on the existing patterns. This could be done by reducing the number of matches produced by the de-composed pattern, by adding conditions before trailing segments that prevent matching too close to the previous segment. It might also be done by tracking the offsets of previous matches and using this information to correctly filter matches even when the segments can overlap. While it is not a silver bullet for all possible regular expressions, this approach will only become more

powerful as additional effort is put into implementing efficient de-compositions and filters to efficiently match commonly used patterns.

Acknowledgements

This material is based in part upon work supported by the National Science Foundation under Grant Number CNS-1318563, Michigan State University Discretionary Funding Initiative, the National Natural Science Foundation of China under Grant Numbers 61472184 and 61321491, and the Jiangsu High-level Innovation and Entrepreneurship (Shuangchuang) Program.

REFERENCES

[1] Darpa intrusion detection evaluation data set. www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1998data.html, 1998.

[2] Us army itoc research cdx 2009 trace. http://www.itoc.usma.edu/research/dataset/index.html, 2009.

[3] Nitroba university harrassment scenario trace. http://digitalcorpora.org/corpora/scenarios/nitroba-university-harassment-scenario, 2014.

[4] M. Bando, N. Artan, and H. Chao. Scalable lookahead regular expression detection system for deep packet inspection. *Networking, IEEE/ACM Transactions on*, 20(3):699 –714, june 2012.

[5] M. Becchi, A. Bremler-Barr, D. Hay, O. Kochba, and Y. Koral. Accelerating regular expression matching over compressed http. 2015.

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

[7] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proc. ANCS*, 2007.

[8] M. Becchi and P. Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *Proc. CoNEXT*, pages 1–12, 2008.

[9] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *Proc. IEEE IISWC*, 2008.

[10] I. Bonesana, M. Paolieri, and M. D. Santambrogio. An adaptable fpga-based system for regular expression matching. In *Proc. DATE*, pages 1262–1267, 2008.

[11] A. Bremler-Barr, S. T. David, D. Hay, and Y. Koral. Decompression-free inspection: Dpi for shared dictionary compression over http. In *INFOCOM, 2012 Proceedings IEEE*, pages 1987–1995. IEEE, 2012.

[12] A. Bremler-Barr, D. Hay, and Y. Koral. Compactdfa: Generic state machine compression for scalable pattern matching. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[13] C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Proc. Field-Programmable Logic and Applications*, pages 956–959, 2003.

[14] J. Kořenek and V. Košař. Nfa split architecture for fast regular expression matching. In *Proc. ANCS*, pages 14:1–14:2, New York, NY, USA, 2010. ACM.

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

[16] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang. Optimization of regular expression pattern matching circuits on fpga. In *Proc. DATE*, pages 12–17, 2006.

[17] A. X. Liu, E. Norige, and S. Kumar. A few bits are enough-asic friendly regular expression matching for high speed network security systems. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–10. IEEE, 2013.

[18] C. Meiners, J. Patel, E. Norige, E. Torng, and A. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In *Proc. 19th USENIX Security*, 2010.

[19] A. Mitra, W. Najjar, and L. Bhuyan. Compiling pcre to fpga for accelerating snort ids. In *Proc. ANCS*, pages 127–136, New York, NY, USA, 2007. ACM.

[20] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[21] K. Peng, S. Tang, M. Chen, and Q. Dong. Chain-based dfa deflation for fast and scalable regular expression matching using tcam. In *Proc. ANCS*, pages 24–35, Washington, DC, USA, 2011. IEEE Computer Society.

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

[23] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *Proc. IEEE FCCM*, pages 227–238, 2001.

[24] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proc. ACM SIGCOMM 2008 Conf. on Data communication*, SIGCOMM '08, pages 207–218, New York, NY, USA, 2008. ACM.

[25] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna. Compact architecture for high-throughput regular expression matching on fpga. In *Proc. ANCS*, pages 30–39, 2008.

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