Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs

Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park*
Geoff Langdale+, Jiayu Hu, and Heqing Zhu

Intel Corporation **KAIST* *+branchfree.org*

Abstract

Regular expression matching serves as a key functionality of modern network security applications. Unfortunately, it often becomes the performance bottleneck as it involves compute-intensive scan of every byte of packet payload. With trends towards increasing network bandwidth and a large ruleset of complex patterns, the performance requirement gets ever more demanding.

In this paper, we present Hyperscan, a high performance regular expression matcher for commodity server machines. Hyperscan employs two core techniques for efficient pattern matching. First, it exploits graph decomposition that translates regular expression matching into a series of string and finite automata matching. Unlike existing solutions, string matching becomes a part of regular expression matching, eliminating duplicate operations. Decomposed regular expression components also increase the chance of fast DFA matching as they tend to be smaller than the original pattern. Second, Hyperscan accelerates both string and finite automata matching using SIMD operations, which brings substantial throughput improvement. Our evaluation shows that Hyperscan improves the performance of Snort by a factor of 8.7 for a real traffic trace.

1 Introduction

Deep packet inspection (DPI) provides the fundamental functionality for many middlebox applications that deal with L7 protocols, such as intrusion detection systems (IDS) [9, 10, 28], application identification systems [4], and web application firewalls (WAFs) [3]. Today's DPI employs regular expression (regex) as a standard tool for pattern description as it flexibly represents various attack signatures in a concise form. Not surprisingly, numerous research works [16, 18, 32, 38, 39, 41, 42] have proposed efficient regex matching as its performance often dominates that of an entire DPI application.

Despite continued efforts, the performance of regex matching on a commodity server still remains impractical to directly serve today's large network bandwidth. Instead, the de-facto best practice of high-performance DPI generally employs multi-string pattern matching as a pre-condition for expensive regex matching. This hybrid approach (or prefiltering) is attractive as multi-string matching is known to outperform multi-regex matching by two orders of magnitude [21], and most input traffic is innocent, making it more efficient to defer a rigorous check. For example, popular IDSes like Snort [9] and Suricata [10] specify a string pattern per each regex for prefiltering, and launch the corresponding regex matching only if the string is found in the input stream.

However, the current prefilter-based matching has a number of limitations. First, string keywords are often defined manually by humans [1]. Manual choice does not scale as the ruleset expands over time, and improper keywords would waste CPU cycles on redundant regex matching. Second, string matching and regex matching are executed as two *separate* tasks, with the former leveraged only as a trigger for the latter. This results in duplicate matching of the string keywords when the corresponding regex matching is executed. Third, current regex matching typically translates an entire regex into a single finite automaton (FA). If the number of deterministic finite automaton (DFA) states becomes too large, one must resort to a slower non-deterministic finite automaton (NFA) for matching of the whole regex.

In this paper, we present Hyperscan, a high performance regex matching system that exploits regex decomposition as the first principle. Regex decomposition splits a regex pattern into a series of disjoint string and FA components [2]. This translates regex matching into a sequence

[1] The content option in Snort and Suricata are determined by humans with domain knowledge.

[2] We refer to a subregex that contains regex meta-characters or quantifiers, which has to be translated into either a DFA or an NFA for

of decomposed subregex matching whose execution and matching order is controlled by fast string matching. This design brings a number of benefits. First, our regex decomposition identifies string components automatically by performing rigorous structural analyses on the NFA graph of a regex. Our algorithm ensures that the extracted strings are pre-requisite for the rest of regex matching. Second, string matching is run as a part of regex matching rather than being employed only as a trigger. Unlike the prefilter-based design, Hyperscan keeps track of the state of string matching throughout regex matching and avoids any redundant operations. Third, FA component matching is executed only when all relevant string and FA components are matched. This eliminates unnecessary FA component matching, which allows efficient CPU utilization. Finally, most decomposed FA components tend to be small, so they are more likely to be able to be converted to a DFA and benefit from fast DFA matching.

Beyond the benefits of regex decomposition, Hyperscan also brings a significant performance boost with single-instruction-multiple-data (SIMD)-accelerated pattern matching algorithms. For string matching, we extend the shift-or algorithm [13] to support efficient multi-string matching with bit-level SIMD operations. For FA matching, we represent a state with a bit position while we implement state transitions and successor state-set calculation with SIMD instructions on a large bitmap. We find that our SIMD-accelerated string matching outperforms state-of-the-art multi-string matching by a factor of 1.3 to 2.5. We also find that our SIMD-accelerated regex matching achieves 24.8x to 40.1x performance improvement over PCRE [6] widely adopted by DPI middleboxes such as Snort and Suricata.

In summary, we make the following contributions:

- We present a novel regex matching strategy that exploits regex decomposition. Regex decomposition performs rigorous graph analysis algorithms that extract key strings of a regex for efficient matching, and drives the order of pattern matching by fast string matching. This drastically improves the performance.
- We develop SIMD-accelerated pattern matching algorithms for both string matching and FA matching to leverage CPU's compute capability on data parallelism.
- Our evaluation shows Hyperscan greatly helps improve the performance of real-world DPI applications. It improves the performance of Snort by 8.7x for a real traffic trace.
- We share our experience with developing Hyperscan and present lessons learned through commercialization.

matching as an FA component.

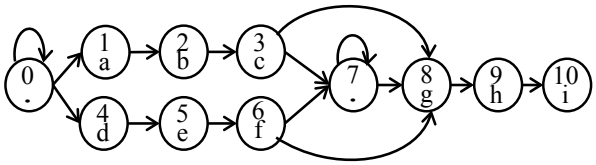


Figure 1: Glushkov NFA for (**abc|def**)**.*ghi**

2 Background and Motivation

DPI is a common functionality in many security middleboxes, and its performance has been mainly driven by that of regex matching [19, 41]. There has been a large body of research that improves the performance of regex matching. Due to space constraint, we briefly review only a few, categorizing them by their approach.

String matching is a subset of regex matching, which requires specialized algorithms [12, 24, 29] to achieve high performance. The most popular one is the Aho-Corasick (AC) algorithm [12] that uses a variant of DFA for fast multi-string matching. It runs in O(n) time complexity where n is the number of input bytes. Unfortunately, AC suffers from frequent cache misses due to large memory footprint and random memory access pattern, which significantly impairs the performance. In addition, the model of processing one byte at a time creates a sequential data dependency that stalls instruction pipelines of modern processors. DFC [21] employs a set of small bitmap filters that quickly pass out innocent traffic by checking the first few bytes of string patterns against the input stream. Each matched input moves onto the verification stage for full pattern comparison. DFC substantially reduces memory accesses and cache misses by using small and cache-friendly data structures, which outperforms AC by 2 to 3.6 times. The string matcher of Hyperscan takes the two-stage matching similar to DFC, but its bucket-based shift-or algorithm benefits from SIMD instructions, which further improves the performance beyond that of DFC.

An NFA implements a space-efficient state machine even for complex regexes. Despite its small memory footprint, the execution is typically slow as each input character triggers O(m) memory lookups (m = # of current states). For this reason, a DFA is preferred to an NFA whenever a regex can be translated into the former. One place where NFA might be preferred is a logic-based design that maps automata to hardware accelerators such as FPGA [14, 22, 23, 34, 35, 40]. An FPGA-based design can exploit parallelism by running multiple finite automata simultaneously and does not suffer from sequential state transition table lookups. On the down side, it is limited to a small ruleset due to its hardware constraints. Also, it

suffers from the DMA overhead of moving data between the CPU and the FPGA device. This overhead can impose prohibitive latency, especially when input data is not large (as would be the case for scanning of small packets).

We use Glushkov NFA [27] for Hyperscan, which is widely used due to its two useful properties. First, it does not have epsilon transitions, which simplifies state transition implementation. Second, all transitions into a given state are triggered by the same input symbol. Figure 1 shows an example of a Glushkov NFA. Each circle represents a state whose id is shown as a number, and each character represents the input symbol by which any previous state transitions into this state. For example, one can transition into state 8 from state 3, 6, or 7 only for an input symbol, 'g'. The second property implies that the total number of states is bounded by the number of input symbols and a few special states – a start state, states with '.', etc.

A DFA achieves high performance as it runs in O(1) per each input character. Its main disadvantage, however, is a large memory footprint and a potential of state explosion at transforming an NFA to a DFA. Thus, most works on DFA focus on memory footprint reduction [15, 17, 18, 20, 26, 31, 32, 33]. D^2FA [32] compresses the memory space by sharing multiple transitions of states with a similar transition table and by establishing a default transition between them. A-DFA [18] presents useful features such as alphabet reduction that classifies alphabets into smaller groups, indirect addressing that reduces memory bandwidth by translating unique state identifiers to memory address, and multi-stride structure that processes multiple characters at a time.

An extended FA is a proposal that restructures the state-of-the-art FA to address state explosion. XFA [38, 39] associates update functions with states and transitions by having a scratch memory that compresses the space. HFA [16] presents a hybrid-FA that achieves comparable space to that of an NFA by making head DFAs and trailing NFA or DFAs. The theory behind it is to discover boundary states so that one can conduct partial conversion of an NFA to a DFA to avoid exponential state explosion from a full conversion.

Prefilter-based approaches are the most popular way to scale performance of regex matching in practice. Both Snort and Suricata extract string keywords from regex rules and perform unified multi-string matching with the Aho-Corasick algorithm. Expensive regex matching is only needed if AC detects literal strings in the input. SplitScreen [36] applies a similar approach to ClamAV [30], a widely-used anti-malware application, and achieves a 2x speedup compared to original ClamAV.

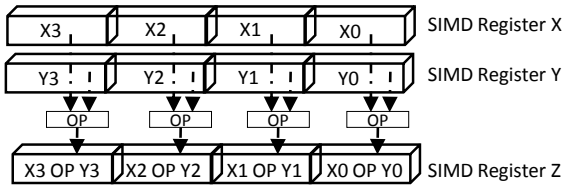


Figure 2: Typical two-operand SIMD operation

A SIMD instruction executes the same operation on multiple data in parallel. As shown in Figure 2, a SIMD operation is performed on multiple lanes of two SIMD registers independently, and the results are stored in the third register. Modern CPU supports a number of SIMD instructions that can work on specialized vector registers (SSE, AVX, etc.). The latest AVX512 instructions support up to 512-bit operations simultaneously.

Despite its great potential, few research works have exploited SIMD instructions for regex matching. Sitaridi et al. propose a SIMD-based regex matching design [37] for database, which uses a gather instruction to traverse DFA for multiple inputs simultaneously. However, it cannot be applied to our case as regex matching for DPI is typically performed on a single input stream.

Summary and our approach. Most prior works on regex matching attempt to build a special FA that performs as well as a DFA while its memory footprint is as small as an NFA. However, one common problem in all these works is that FA restructuring inevitably imposes extra performance overhead compared to the original DFA. For example, XFA takes multiple tens to hundreds of CPU cycles per input byte, which is slower than a normal DFA by one or two orders of magnitude. In contrast, the prefilter-based approach looks attractive as it benefits from multi-string matching most time, which is faster than multi-regex matching by a few orders of magnitude. However, it is still suboptimal as it must perform duplicate string matching during regex matching, and wrong choice of string patterns would trigger redundant regex matching (as shown in Section 6.2). To avoid the inefficiency, we take a fresh approach that divides a regex pattern into multiple components, and leverages fast string matching to coordinate the order of component matching. This would minimize the waste of CPU cycles on redundant matching and thus improves the performance. In addition, we develop our own multi-string matching and FA matching algorithms carefully tailored to exploit SIMD operations.

3 Regular Expression Decomposition

In this section, we present the concept of regex decomposition, and explain how Hyperscan matches a sequence of regex components against the input. Then, we introduce

graph-based decomposition whose graph analysis techniques reliably identify the strings in regex patterns most desirable for string matching.

3.1 Matching with Regex Decomposition

The key idea of Hyperscan is to decompose each regex pattern into a disjoint set of string and subregex (or FA) components, and to match each component until it finds a complete match. A string component consists of a stream of literals (or input symbols). Subregex components are the remaining parts of a regex after all string components are removed. They may include one or more meta-characters or quantifiers in regex syntax (like '^', '$', '*', '?', etc.) that need to be translated into an FA for matching. Thus, we refer to it as an **FA** component.

Linear regex. We start with a simple regex where each component is concatenated without alternation. We call it a linear regex. Formally, a linear regex pattern that contains at least one string can be represented as the following production rules:

1. *regex* $\rightarrow$ *left* **str FA**
2. *left* $\rightarrow$ *left* **str FA** | **FA**

where **str** and **FA** are both indivisible components, and **FA** can be empty. A linear regex without any string is implemented as a single DFA or NFA. In practice, however, we find that 87% to 94% of the regex rules in IDSes have at least one extractable string, so a majority of real-world regexes would benefit from decomposition. The production rules imply that if we find the rightmost string in a linear regex pattern, we can recursively apply the same algorithm to decompose the rest of the pattern. One complication lies in a subregex with a repetition operator such as $(R)?$, $(R)*$, $(R)+$, and $(R)\{m,n\}$, where R is arbitrarily complex. Hyperscan treats $(R)?$ and $(R)*$ as a single FA since R is optional while it converts $(R)+ = (R)(R)*$, and $(R)\{m,n\} = (R)...(R)(R)\{0,n-m\}$ ((R) appears m times)). Then, it decomposes their prefixes and treats the suffix as an FA.

In general, a decomposable linear regex can be expressed as $/FA_n\ str_n\ FA_{n-1}\ str_{n-1}\ \cdots\ str_2\ FA_1\ str_1\ FA_0/$. For any successful match of the original regex, all strings must be matched in the same order as they appear. Based on the observation, Hyperscan applies the following three rules for regex matching.

1. String matching is the first step. It scans the entire input to find all **strs**. Each successful match of **str** may trigger the execution of its neighbor **FA** matching.
2. Each **FA** has its own switch for execution. It is off by default except for the leftmost **FA** components.
3. For a generalized form like /*left* **FA str** *right*/ where "left" or "right" is an arbitrary set of decomposed components including an empty character. Only if all components of "left" are matched successfully, the switch of **FA** is turned on. Only if **str** is matched successfully and the **FA** switch is on, **FA** matching is executed. Finally, only if **FA** is matched successfully, the leftmost **FA** of "right" is turned on.

Let's take one example regex, /.*foo[^X]barY+/, and consider two input cases. The regex pattern is decomposed into $/FA_2\ str_2\ FA_1\ str_1\ FA_0/$, where FA_2=".*", str_2="foo",FA_1="[^X]", str_1="bar", FA_0="Y+".

- **Input="XfooZbarY"**: This is overall a successful match. First, the string matcher finds str_2 ("foo"), and triggers matching of FA_2 (".*") against "X" since the leftmost **FA** switch is always on. Then, the switch of FA_1 ("[^X]") is turned on. After that, the matcher finds str_1 ("bar"), which triggers matching of FA_1 against "Z", and its success turns on the the switch of FA_0 ("Y+"). Since FA_0 is the rightmost **FA**, it is executed against the remaining input, "Y".
- **Input="XfoZbarY"**: This is overall a failed match. First, the string matcher finds str_1 ("bar"), and sees if it can trigger matching of FA_1 ("[^X]"). Then, it figures out that the switch of FA_1 is off since str_2 ("foo") was not found, and thus none of FA_2 and str_2 was a successful match. So, the matching of regex FA_1 terminates, ensuring no waste of CPU resources.

Our implementation tracks of the input offsets of matched strings and the state of matching individual components, which allows invoking appropriate **FA** matching with a correct byte range of the input.

Regex with alternation. If a regex includes an alternation like $(A|B)$, we expand the alternation into two regexes only if both A and B are decomposed into **str** and **FA** components (decomposable). If not, $(A|B)$ is treated as a single FA. In case A or B itself has an alternation, we need to recursively apply the same rule until there is no alternation in their subregexes. Then, each expanded regex would become a linear regex, which would benefit from the same rules for decomposition and matching as before.

Pattern matching with regex decomposition presents its main benefit – it minimizes the waste of CPU cycles on unnecessary **FA** matching because **FA** matching is executed only when it is needed. Also, it increases the chance of fast DFA matching as each decomposed FA is smaller, so it is more likely to be converted into a DFA. In contrast, the prefiltering approach has to execute matching of the entire regex even when it is not needed (e.g., matching "bar" in the example above would trigger regex matching even if "foo" is not found), and regex matching must re-match the string already found in string

matching. Furthermore, conversion of a whole regex into a single FA is not only more complex, but often ends up with a slower NFA to avoid state explosion. In terms of correctness, pattern matching with regex decomposition produces the same result as the original regex, but we leave its formal proof as our future work.

3.2 Rationale and Guidelines

In practice, performing regex decomposition on its textual form is often tricky as some string segments might look hidden behind special regex syntax. We provide several such examples below:

- Character class (or character-set). $/b[il1]l\backslash s\{0,10\}/$ includes a character class that can be expanded to three strings ("bil", "bll" and "b1l") while naïve textual extraction might find only 'b' and 'l'.
- Alternation. The alternation sequence in $/(.*\backslash x2d(h|H)(t|T)(t|T)(p|P))/$ makes it harder to discover "http" sequences from textual extraction.
- Bounded repeats. From the perspective of text, the strings with a minimum length of 32 are hidden from bounded repeats in $/[\backslash x40\backslash x90]\{32,\}/$.

To reliably find these strings, we perform regex decomposition on the Glushkov NFA graph [27], which would benefit from structural graph analyses. We describe useful guidelines for finding the strings effective for regex matching.

1. Find the string that would divide the graph into two subgraphs, with the start state in one subgraph, and the set of accept states in the other. Matching such a string is a necessary condition for any successful match on the entire regex. If the start and accept states happen to be in the same subgraph, the corresponding FA will always have to run regardless of a string match.

2. Avoid short strings. Short strings are prone to match more frequently, and are likely to trigger expensive FA matching that fails.[3]

3. Expand small character-sets [4] to multiple strings to facilitate decomposition. This would not only increase the chance of successful decomposition but also lead to a longer string if a character-set intercepts a string sequence (i.e. "$document[\backslash x22\backslash x27]object$").

4. Avoid having too many strings. Having too many strings for matching would overload the matcher and degrade the entire performance. So, it is important to find a small set of "good" strings effective for regex matching.

[3] Our current limit is 2 to 3 characters.

[4] Our current implementation treats a character-set that expands to 11 or smaller strings as a small character-set.

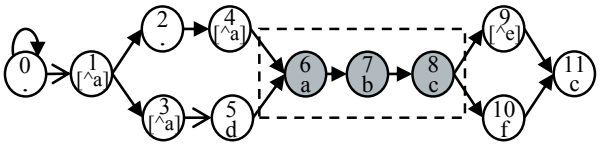


Figure 3: Dominant path analysis

3.3 Graph-based String Extraction

We develop three graph analysis techniques that discover strings in the critical path for matching. We describe the key idea of each algorithm below, and provide more detailed algorithms in an appendix.

Dominant path analysis. A vertex u is called a *dominator* of vertex v if every path from the start state to vertex v must go through vertex u. A dominant path of vertex v is defined as a set of vertices W in a graph, where each vertex in W is a dominator of v and the vertices form a trace of a single path. Dominant path analysis finds the longest common string that exists in all dominant paths of any accept state. For example, Figure 3 shows the string on the dominant path of the accept state (vertex 11).

The string selected by the analysis is highly desirable for matching as it clearly divides start and accept states into two separate subgraphs, satisfying the first guideline. The algorithm calculates the dominant path per each accept state, and finds the longest common prefix of all dominant paths. Then, it extracts the string on the chosen path. If a vertex on the path is a small character-set, we expand it and obtain multiple strings.

Dominant region analysis. If the dominant path analysis fails to extract a string, we perform dominant region analysis. It finds a region of vertices that partition the start state into one graph and all accept states into the other. More formally, a dominant region is defined as a subset of vertices in a graph such that (a) the set of all edges that enter and exit the region constitute a cut-set of the graph, (b) for every in-edge (u, v) to the region, there exist edges (u, w) for all w in {w : w is in the region and w has an in-edge}, where (u, v) refers to an edge from vertex u to v in the graph, and (c) for every out-edge (u, v) from the region, there exist edges (w, v) for all w in {w : w in is in the region and w has an out-edge}.

If a discovered region consists of only string or small character-set vertices, we transform the region into a set of strings. Since these strings connect two disjoint subgraphs of the original graph, any match of the whole regex must match one of these strings. Figure 4 shows one example of a dominant region with 9 vertices. Vertices 5, 6, and 7 are the entry points with the same predecessors and vertices

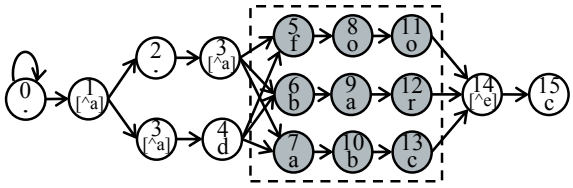


Figure 4: Dominant region analysis

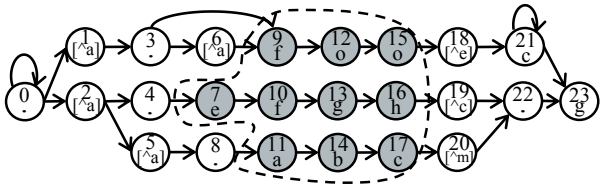


Figure 5: Network flow analysis

11, 12, and 13 are the exits with the same successor. We can extract strings, "*foo*", "*bar*", and "*abc*", as a result of dominant region analysis.

The algorithm for dominant region analysis first creates a directed acyclic graph (DAG) from the origin graph to avoid any interference from back edges. Then, it performs topological sort on the DAG, and iterates each vertex to see if it is added to the current candidate region, its boundary edges form a valid cut-set. We repeat this to discover all regions in the graph. Since we only analyze the DAG, the back edges of the original graph might affect the correctness. Thus, for each back edge, if its source and target vertices are in different regions, we merge them (and all intervening regions) into a single region. Finally, we extract the strings from the dominant region.

Network flow analysis. Since dominant path and dominant region analyses depend on a special graph structure, they may not be always successful. Thus, we run *network flow* analysis for generic graphs. For each edge, the analysis finds a string (or multiple strings) that ends at the edge. Then, the edge is assigned a score inversely proportional to the length of the string(s) ending there. The longer the string is, the smaller the score gets. With a score per edge, the analysis runs "max-flow min-cut" algorithm [25] to find a minimum cut-set that splits the graph into two that separate the start state from all accept states. Then, the "max-flow min-cut" algorithm discovers a cut-set of edges that generate the longest strings from this graph.

Figure 5 shows a result of network flow analysis, extracting a string set of "*foo*", "*efgh*", and "*abc*" that would divide the whole graph into two parts.

Effectiveness of graph analysis. Our graph analysis effectively produces "good" strings for most of real-world rules. Table 1 shows that 97.2% to 99.2% of decomposable real-world regex rules benefit from dominant path

Ruleset	Total	Decomp	D-Path	D-Reg	N-flow
S-V	1,663	1,563	1,551	32	16
S-E	7,564	6,756	6,575	100	203
Suri	7,430	6,501	6,318	94	201

Table 1: Effectiveness of graph analysis on real-world rulesets. S-V: Snort Talos (May 2015), S-E: Snort ET-Open 2.9.0, Suri: Suricata rulesets 4.0.4. D-Path, D-Reg, and N-flow refer to dominant path, dominant region, and network flow analysis, respectively. Decomp is the total number of decomposable rules. Note that one regex could benefit from multiple graph analyses, so the sum of graph analyses is larger than the Decomp fields.

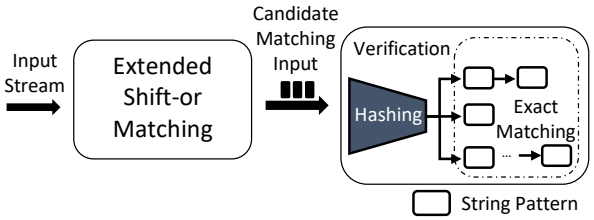


Figure 6: Two-stage matching with FDR

analysis while remaining patterns exploit dominant region and network flow analysis. These strings prove to be highly beneficial for reducing the number of regex matching invocations, as shown in Section 6.2.

4 SIMD-accelerated Pattern Matching

In this section, we present the design of multi-string and FA matching algorithms that leverage SIMD operations of modern CPU.

4.1 Multi-string Pattern Matching

We introduce an efficient multi-string matcher called FDR. [5] The key idea of FDR is to quickly filter out innocent traffic by fast input scanning. As shown in Figure 6, FDR performs extended *shift-or matching* [13] to find candidate input strings that are likely to match some string pattern. Then, it verifies them to confirm an exact match.

Shift-or matching. We first provide a brief background of shift-or matching that serves as the base algorithm of FDR. The shift-or algorithm finds all occurrences of a string pattern in the input bytestream by performing bitwise `shift` and `or` operations, as shown in Figure 7. It uses two data structures – a *shift-or mask* for each character c in the symbol set, (sh-mask('c')), and a *state mask* (st-mask) for matching operation. sh-mask('c') zeros all bits whose bit position corresponds to the byte position of c in the string pattern while all other bits are set to 1. The bit position in a sh-mask is counted from the *rightmost* bit while the byte position in a pattern is counted from the *leftmost* byte. For example, for a string pattern, "aphp",

[5] It is named after the 32nd President of the U.S.

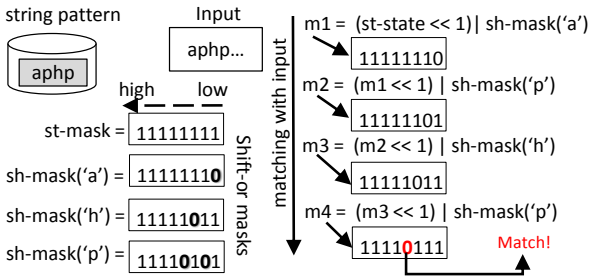


Figure 7: Classical shift-or matching

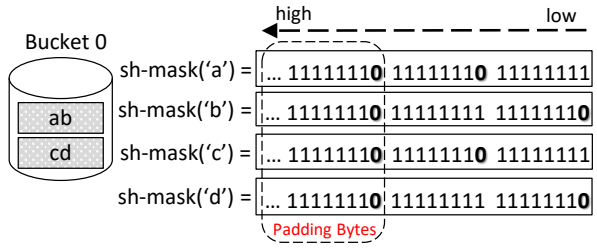


Figure 8: Example shift-or masks with two patterns at bucket 0. No other buckets contain 'a', 'b', 'c', 'd' in their patterns.

sh-mask('p') = 11110101 as 'p' appears at the second and the fourth position in the pattern. If a character is unused in the pattern, all bits of its sh-mask are set to 1. The algorithm keeps a st-mask whose size is equal to the length of a sh-mask. Initially, all bits of the st-mask are set to 1. The algorithm updates the st-mask for each input character, 'x' as st-mask = ((st-mask $\ll$ 1) | sh-mask('x')). For each matching input character, **0** is propagated to the left by one bit. If the zero bit position becomes the length of the pattern, it indicates that the pattern string is found.

The original shift-or algorithm runs fast with high space efficiency, but it leaves two limitations. First, it supports only a single string pattern. Second, although it consists of bit-level operations, the implementation cannot benefit from SIMD instructions except for very long patterns. We tackle these problems as below.

Multi-string shift-or matching. To support multi-string patterns, we update our data structures. First, we divide the set of string patterns into n distinct buckets where each bucket has an id from 0 to n-1. For now, assume that each string pattern belongs to one of n buckets as we will discuss how we divide the patterns later ('pattern grouping'). Second, we increase the size of sh-mask and st-mask by n times so that a group of n bits in sh-mask('x') record all the buckets that have 'x' in some of their patterns. More precisely, the k-th n bits of sh-mask('x') encode the ids of all buckets that hold at least one pattern which has 'x' at the k-th byte position. One difference from the original algorithm is that the byte position in a pattern is counted from the *rightmost* byte. This enables parallel execution of multiple CPU instructions per cycle as explained later ('SIMD acceleration'). For efficient implementation, we

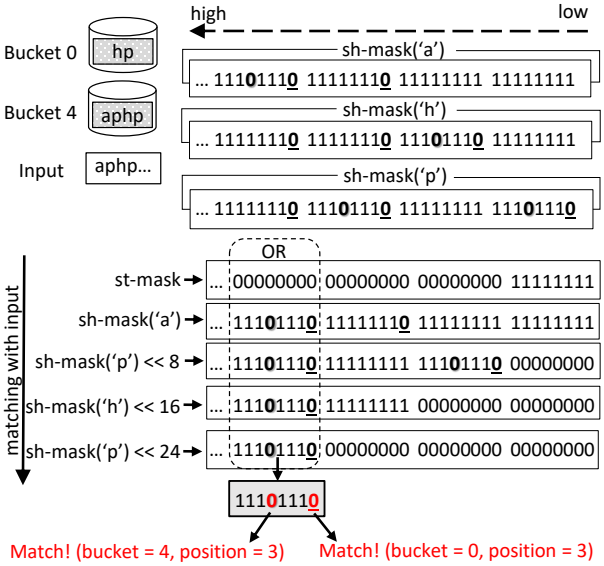


Figure 9: FDR's extended shift-or matching

set n to 8 so that the byte position in a sh-mask matches the same position in a pattern. This implies that the length of a sh-mask should be no smaller than the longest pattern.

Figure 8 shows an example. Bucket 0 has two string patterns, "ab" and "cd". Since 'a' appears at the second byte of only "ab" in bucket 0, sh-mask('a') zeros only the first bit (= bucket 0) of the second byte. The i-th bit within each byte of a sh-mask indicates the bucket id, i-1. If the byte position of a sh-mask exceeds the longest pattern of a certain bucket (called 'padding byte'), we encode the bucket id in the padding byte. This ensures matching correctness by carrying a match at a lower input byte along in the shift process. Examples of this are shown in the padding bytes in Figure 8, and in the sh-masks in Figure 9 that zero the first bit in the third and fourth bytes.

The pattern matching process is similar to the original algorithm except that sh-masks are shifted left instead of the st-mask. The st-mask is initially 0 except for the byte positions smaller than the shortest pattern. This avoids a false-positive match at a position smaller than the shortest pattern. Now, we proceed with input characters. The matcher keeps k, the number of characters processed so far modulo n. For an input character, 'x', st-mask |= (sh-mask('x') $\ll$ (k bytes)). The matcher repeats this for n input characters, and checks if the st-mask has any zero bits. Zero bits represent a possible match at the corresponding bucket. For example, Figure 9 shows that bucket 0 and 4 have a potential match at input byte position 3. The verification stage illustrated later checks whether they are a real match or a false positive.

Pattern grouping. The strategy for grouping patterns into each bucket affects matching performance. A good

strategy would distribute the patterns well such that most innocent traffic would pass with a low false positive rate. Towards the goal, we design our algorithm based on two guidelines. First, we group the patterns of a similar length into the same bucket. This is to minimize the information loss of longer patterns as the input characters match only up to the length of the shortest pattern in a bucket for matching correctness. Second, we avoid grouping too many short patterns into one bucket. In general, shorter patterns are more likely to increase false positives. To meet these requirements, we sort the patterns in the ascending order of their length, and assign an id of 0 to (s-1) to each pattern by the sorted order. Then, we run the following algorithm that calculates the minimum cost of grouping the patterns into n buckets using dynamic programming. The algorithm is summarized by the two equations below:

1. $t[i][j] = \min_{k=i+1}^{s-1}(cost_{ik} + t[k+1][j-1])$, where s is the number of patterns and t[i][j] stores the minimum cost of grouping the patterns i to (s-1) into (j+1) buckets.
2. $cost_{ik} = (k-i+1)^{\alpha}/length_i^{\beta}$, where cost_{ik} is the cost of grouping patterns i to k into one bucket, length_i is for pattern i, α and β are constant parameters.

t[i][j] is calculated as the minimum of the sum of the cost of grouping patterns i to k into one bucket (cost_{ik}) and the minimum cost of grouping remaining patterns (k+1) to (s-1) into j buckets (t[k+1][j-1]). cost_{ik} gets smaller as the bucket has a longer pattern, which allows more patterns in the bucket. It gets larger as the bucket has a shorter pattern, limiting the number of such patterns. Our implementation currently uses $\alpha = 1.05$ and $\beta = 3$ towards this goal, and computes t[0][7] to divide all string patterns into 8 buckets, and records the bucket id per each pattern in the process. In practice, we find that the algorithm works well, automatically reaching the sweetspot that minimizes the total cost.

Super Characters. One problem with the bucket-based matching is that it produces false positives with patterns in the same bucket. For example, if a bucket has /ab/ and /cd/, the algorithm not only matches the correct patterns but also matches false positives, /ad/ and /cb/. To suppress them, we use an m-bit (m>8) super character (instead of an 8-bit ASCII character) to build and index the sh-masks. An m-bit super character consists of a normal (8-bit) character in the lower 8 bits and low-order (m-8) bits of the next character in the upper bits. If it is the last character of a pattern (or in the input), we use a null character (0) as the next character. The key idea is to reflect some information of the next character in a pattern into building the sh-mask for the current character. Only if the same two characters appear in the input [6], we declare a match at that input byte position. This would significantly reduce false positives at the cost of a slightly large memory for sh-masks.

In practice, m should be between 9 and 15. Let's say m = 12 bits. For a pattern, /ab/, we see two 12-bit super characters, α = ((low-order 4 bits of 'b' $\ll$ 8) | 'a'), and β = 'b'. Then, we build sh-masks for α and β, respectively. When the input arrives, we construct a 12-bit super character based on the current input byte offset, and use it as an index to fetch its sh-mask. We advance the input one byte at a time as before. For example, if the input is 'ad', it first constructs γ = ((low-order 4 bits of 'd' $\ll$ 8) | 'a'), fetches sh-mask('γ'), and performs "shift" and "or" operations as before. Then, it advances to the next byte and constructs δ = 'd'. So, the input 'ad' will not match even if a bucket contains /ab/ and /cd/.

SIMD acceleration. Our implementation of FDR heavily exploits SIMD operations and instruction-level parallelism. First, it uses 128-bit sh-masks so that it employs 128-bit SIMD instructions (e.g., *pslldq* for "left shift" and *por* for "or" in the Intel x86-64 instruction set) to update the masks. As "shift" and "or" are the most frequent operations in FDR, it enjoys a substantial performance boost with the SIMD instructions. Second, it exploits parallel execution with multiple CPU execution ports. In the original shift-or matching, the execution of "shift" and "or" operations is completely serialized as they have dependency on the previous result. This under-utilizes modern CPU even if it can issue multiple instructions per CPU cycle. In contrast, FDR exploits instruction-level parallelism by pre-shifting the sh-masks with multiple input characters in parallel. Note that this is made possible as we count the byte position differently from the original version. The parallel execution effectively increases instructions per cycle (IPC) and significantly improves the performance. To accommodate the parallel shifting, we limit the length of a pattern to 8 bytes and extract the lower 8 bytes from any pattern longer than 8 bytes. Because it requires minimum 8-byte masks and up to 7 bytes of shifting, a 128-bit mask would not lose high bit information during left shift. In actual matching, FDR handles 8 bytes of input at a time. To guarantee a contiguous matching across 8-byte input boundaries, the st-mask of the previous iteration is shifted right by 8 bytes for the next iteration.

Verification. As our shift-or matching can still generate false positives, we need to verify if a candidate match is

[6] Of course, there is still a small chance of a false positive as we use partial bits of the next character, but the probability becomes fairly small as the pattern length grows.

Algorithm 1 FDR Multi-string Matcher

```
1:  function MATCH
2:      n := number of bits of a super character
3:      R := startMask
4:      for each 8–byte V ∈ input do
5:          for i ∈ 0...7 do
6:              index = V[i*8..i*8+n−1]
7:              M[i] := shiftOrMask[index]
8:              S[i] := LSHIFT(M[i], i)
9:          end for
10:         for i ∈ 0...7 do
11:             R := OR128(R, S[i])
12:         end for
13:         for zero bit b in low 8 bytes of R do
14:             j := the byte position containing bit b
15:             l := length of string in bucket b
16:             h := HASH(V[j−l+1..j])
17:             if h is valid then
18:                 Perform exact matching for each
19:                 string in hash bucket[h]
20:             end if
21:         end for
22:         R := RSHIFT(R, 8))
23:     end for
24: end function
```

an exact match. This phase consists of hashing and exact string comparison. To minimize hash collisions, we build a separate hash table for each bucket. Then, we leverage the byte position of a match and compare the input with each string in the hash bucket to confirm a match. In practice, we find hashing filters out a large portion of false positives.

4.2 Finite Automata Pattern Matching

Successful string matching often triggers FA component matching, which is essentially the same as general regex matching. Our strategy is to use a DFA whenever is possible, but if the number of DFA states exceeds a threshold [7], we fall back to NFA-based matching. As the state-of-the-art DFA already delivers high performance, we introduce fast NFA-based matching with SIMD operations here.

In NFA-based matching, there can be multiple current states (current set) that are active at any time. A state transition with an input character is performed on every active state in the current set in parallel, which produces a set of successor states (successor set). A match is successful if any state reaches one of the accept states.

We develop *bit-based NFA* where each bit represents a state. We choose the bit-based representation as it outperforms traditional NFA representations that use byte arrays to store transitions, and look up a transition table for each current state in a serialized manner. Also, bit-based NFA leverages SIMD instructions to perform vectorized bit operations to further accelerate the performance. Our scheme assigns an id to each state (i.e. each vertex in an n-node NFA graph) from 0 to n-1 by the topological order, and maintains a current set as a bit mask (called current-set mask) that sets a bit to 1 if its position matches a current state id. We define a *span of a state transition* as the id difference between the two states of the transition. Since state ids are sequentially assigned by the topological order, the span of a state transition is typically small. We exploit this fact to compactly represent the transitions below.

The bit-based NFA implements a state transition with an input character, 'c', in three steps. First, it calculates the successor set that can be transitioned to, from any state in the current set with *any* input character. Second, it computes the set of all states that can be transitioned to, from any state with 'c' (called reachable states by 'c'). Third, it computes the intersection of the two sets. This produces a correct successor set as a Glushkov NFA graph guarantees that one can enter a specific state only with the same character or with the same character-set.

The challenge is to efficiently calculate the successor set. One can pre-calculate a successor set for every combination of current states, and look up the successor set for the current-set mask. While this is fast, it requires storing 2^n successor sets, which becomes intractable except for a small n. An alternative is to keep a successor-set mask per each individual state, and to combine the successor set of every state in the current set. This is more space-efficient but it costs up to n memory lookups and $(n-1)$ "or" operations. We implement the latter, but optimize it by minimizing the number of successor-set masks, which would save memory lookups. To achieve this, we keep a set of *shift-k masks* shared by all relevant states. A shift-k mask records all states with a forward transition of span k, where a forward transition moves from a smaller state id to a larger one, and a backward transition does the opposite. Figure 10 shows some examples of shift-k masks. Shift-1 mask sets every bit to 1 except for bit 7 since all states except state 7 has a forward transition of span 1.

We divide each state transition into two types – typical or exceptional. A *typical* transition is the one whose span is smaller or equal to a pre-defined shift limit. Given a shift limit, we build shift-k masks for every k ($0 \leq k \leq$ limit) at initialization. These masks allow us to efficiently compute the successor set from the current set following typical transitions. If the current-set mask is S, then ((S & shift-k mask) $\ll k$) would represent all possible successor states with transitions of span k from S. If we combine successor sets for all k, we obtain the successor set reached by all typical transitions.

[7] We use 16,384 states as the threshold.

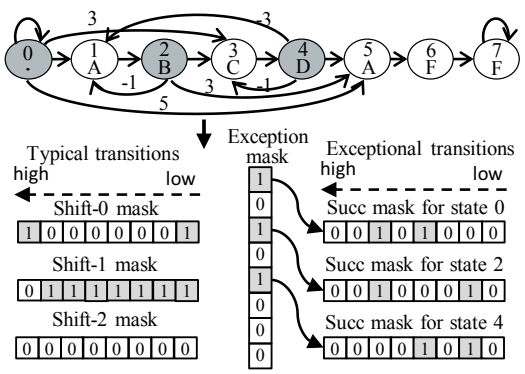


Figure 10: NFA representation for $(AB|CD)^*AFF^*$

We call all other transitions *exceptional*. These include forward transitions whose span exceeds the limit and any backward transitions. [8] Any state that has at least one exceptional transition keeps its own successor mask. The successor mask records all states reached by exceptional transitions of its owner state. All exceptional states are maintained in an exception mask.

As you can see, the choice of the shift limit affects the performance. If it is too large, we would have too many shift-k masks representing rare transitions, and if it is too small, we would have to handle many exceptional states. Our current implementation uses 7 after performance tuning with real-world regex patterns.

Figure 10 shows an NFA graph for $(AB|CD)^*AFF^*$. We set the shift limit to 2 and mark exceptional edges with the difference of ids. State 0, 2, and 4 are highlighted as they have exceptional out-edge transitions. The exception mask holds all exceptional states, and each state points to its own successor mask. For example, successor mask for state 2 sets bits 1 and 5 as its exceptional transitions point to states 1 and 5.

Algorithm 2 shows our bit-based NFA matching. It combines the successor masks possibly reached by typical transitions (*SUCC_TYP*) and exceptional transitions (*SUCC_EX*). Then, it fetches the reachable state set with the current input character, c, ($reach[c]$) and perform a bitwise "and" operation with the combined successor mask (*SUCC*). The result is the final successor set, and we report a match if the successor set includes any accept state. Otherwise, it proceeds with the next input character. For each character, it runs in O($l + e$) where l is the shift limit, and e is the number of "exception" states. Our implementation uses a 128-bit mask (and extends it up to a 512-bit mask with four variables if needed), and employs 128-bit SIMD instructions for fast bitwise operations. In practice, we find that 512 states are enough for representing the NFA graph of most regexes.

[8] In our implementation, forward transitions that cross the 64-bit boundary of the state id space (e.g., from an id smaller than 64 to an id larger than 64) are also treated as exceptional. This is related to a specific SIMD instruction that we use, so we omit the detail here.

Algorithm 2 Bit-based NFA Matching

```
1:  # SH_MSKS[i]     : shift-i masks for typical transitions
2:  # SUCC_MSKS[i]   : successor mask for state i
3:  # EX_MSK         : exception mask
4:  # reach[k]       : reachable state set for character k
5:  function RUNNFA(S: current active state set)
6:      SUCC_TYP := 0,   SUCC_EX := 0
7:      for c ∈ input do
8:          if any state is active in S then
9:              for i := 0 to shiftlimit do
10:                 R0 := AND(S, SH_MSKS[i])
11:                 R1 := LSHIFT(R0, i)
12:                 SUCC_TYP := OR(SUCC_TYP, R1)
13:             end for
14:             S_EX := AND(S, EX_MSK)
15:             for active state s in S_EX do
16:                 SUCC_EX :=
17:                   OR(SUCC_EX, SUCC_MSKS[s])
18:             end for
19:             SUCC := OR(SUCC_TYP, SUCC_EX)
20:             S := AND(SUCC, reach[c])
21:             Report accept states in S
22:         end if
23:     end for
24: end function
```

Total Size	1 GBytes
Number of Packets	818,682
Number of TCP Packets	818,520
Percent of TCP Bytes	97.2%
Percent of HTTP Bytes	92.9%
Average Packet Size	1265 Bytes

Table 2: HTTP traffic trace from a cloud service provider.

5 Implementation

Hyperscan consists of compile and run time functions. Compile-time functions include regex-to-NFA graph conversion, graph decomposition, and matching components generation. The run time executes regex matching on the input stream. While we cover the core functions in Section 3 and 4, Hyperscan has a number of other subsystems and optimizations:

- Small string-set (<80) matching. This subsystem implements a shift-or algorithm using the SSSE3 "PSHUFB" instruction applied over a small number (2-8) of 4-bit regions in the suffix of each string.
- NFA and DFA cyclic state acceleration. Where a state (in the case of the DFA) or a set of states (in the case of the NFA) can be shown to recur until some input is seen, we consider these cyclic states. In case where

# of regexes	Prefilter	Hyperscan	Reduction
500	2,971,652	645,326	4.6x
1000	93,595,304	714,582	131.0x
1500	110,122,972	791,017	139.2x
2000	139,804,519	780,665	179.1x
2500	156,332,187	857,100	182.4x

Table 3: Regex invocations of Snort's ET-Open ruleset

# of regexes	Prefilter	Hyperscan	Reduction
700	699,622	18,164	38.5x
850	7,516,464	19,214	391.2x
1000	17,063,344	32,533	524.5x
1150	17,737,814	34,075	520.6x
1300	25,143,574	36,040	697.7x

Table 4: Regex invocations for Snort's Talos ruleset

current states in NFA or DFA are all cyclic states with a large reachable symbol set, there is a high probability of staying at current state(s) for many input characters. We have SIMD acceleration for searching the first exceptional input sequence (1-2 bytes) that leads to one or more transitions out of current states or switches off one of the current states.

- Small-size DFA matching. We design a SIMD-based algorithm for a small-size DFA (< 16 states) that outperforms the state-of-the-art DFA by utilizing the shuffle instruction for fast state transitions.
- Anchored pattern matching. When an anchored pattern consists of comparatively short acyclic sequences (i.e. no loops), the automata corresponding to them are both simple and short-lived. They are thus cheap to scan and scale well. We run DFA-based subsystems specialized to attempt to discover when anchored patterns are matched or rejected.
- Suppression of futile FA matching. We design a fast lookaround approach that peeks at inputs that are near the triggers for an FA before running it. This often allows us to discover that the FA either does not need to run at all or will have reached a dormant state before the triggers arrive. These checks are implemented as comparatively simple SIMD checks and can be done in parallel over a range of input characters and character classes. For example, in the regex fragment $/R\backslash d\backslash s\{4,5\}foo/$, where R is a complex regex, we can first detect that the digit and space character classes have matched with SIMD checks, and, if not, avoid or defer running a potentially expensive FA associated with R.

6 Evaluation

In this section, we evaluate the performance of Hyperscan to answer the following questions. (1) Does regex decomposition extract better strings than those by manual choice? (2) How well do multi-string matching and regex matching perform in comparison with the existing state-of-the-art? (3) How much performance improvement does Hyperscan bring to a real DPI application?

6.1 Experiment Setup

We use a server machine with Intel Xeon Platinum 8180 CPU @ 2.50GHz and 48 GB of memory, and compile the code with GCC 5.4. To separate the impact by network I/O, we evaluate the performance with a single CPU core by feeding the packets from memory. We test with packets of random content as well as a real-world Web traffic trace obtained at a cloud service provider as shown in Table 2. For all evaluation, we use the latest version of Hyperscan (v5.0) [2].

6.2 Effectiveness of Regex Decomposition

The primary goal of regex decomposition is to minimize unnecessary FA matching by extracting "good" strings from a set of regexes with rigorous graph analyses. To evaluate this point, we compare the number of regex matching invocations triggered by a prefilter-based DPI and by Hyperscan. We extract the content options and their associated regex from Snort rulesets, and count the number of regex matching invocations for a prefilter-based DPI. And then, we measure the same number for Hyperscan where Hyperscan automatically extracts the strings from regexes rather than using the keywords from the content option. For the ruleset, we use ET-Open 2.9.0 [8] and Talos 2.9.11.1 [11] against the real traffic trace, and confirm the correctness – both versions produce the identical output for the traffic.

Tables 3 and 4 show that Hyperscan dramatically reduces the number of regex matching invocations by over two orders of magnitude! As the number of regex rules increases, the reduction by Hyperscan grows, affirming that regex decomposition is the key contributor to efficient pattern matching. Close examination reveals that there are many single-character strings in the content option of the Snort rulesets, which invokes redundant regex matching too frequently. In practice, other rule options in Snort may mitigate the excessive regex invocations, but frequent string matching alone poses a severe overhead. In contrast, Hyperscan completely avoids this problem by triggering regex matching only if it is necessary.

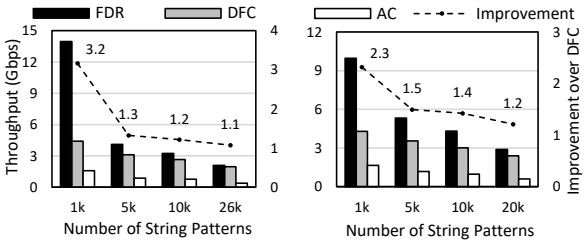
6.3 Microbenchmarks

We evaluate the performance of FDR, our multi-string matcher, as well as that of regex matching of Hyperscan.

Multi-string pattern matching. We compare the performance of FDR with that of DFC and AC. We measure the performance over different numbers of string patterns ex-

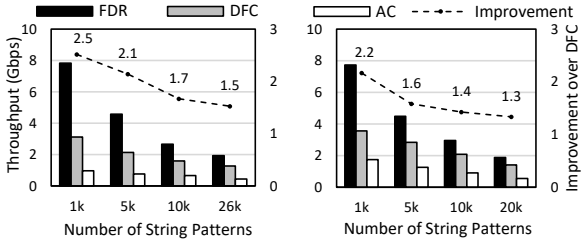
Ruleset	PCRE	PCRE2	RE2-s	Hyperscan-s	RE2-m	Hyperscan-m
Talos	6,942	394	1,777	173	29	2.15
ET-Open	12,800	913	4,696	516	1,116	133

Table 5: Performance comparison with PCRE, PCRE2, RE2 and Hyperscan for Snort Talos (1,300 regexes) and Suricata (2,800 regexes) rulesets with the real Web traffic trace. Numbers are in seconds.



(a) ET-Open ruleset **(b)** Talos ruleset

Figure 11: String matching performance with random packets



(a) ET-Open ruleset **(b)** Talos ruleset

Figure 12: String matching performance with a real traffic trace

tracted from Snort ET-Open and Talos rulesets. Figure 11 and Figure 12 show that FDR outperforms the state-of-the-art matcher, DFC, by 1.1x to 3.2x (and AC by 4.2x to 8.8x) for packets of random content and by 1.3x to 2.5x (and AC by 3.2x to 8.2x) for the real traffic trace. We also evaluate it with the Suricata ruleset but we omit the result here since it exhibits the similar performance trend. When the number of string patterns is small, Hyperscan benefits from small CPU cache footprint and SIMD acceleration, but when the number of patterns grows, the performance becomes compatible with DFC due to increased cache usage, but it is still much better than AC.

Regex matching. We now evaluate the performance of regex matching of Hyperscan. We compare the performance with PCRE (v8.41) [6] as it is most widely used in network DPI applications, and PCRE2 (v10.32) [7], a more recent fork from PCRE with a new API. We enable the JIT option for both PCRE and PCRE2. We also compare with RE2 (v2018-09-01) [1], a fast, small memory-footprint regex matcher, developed by Google. Both Hyperscan and RE2 support multi-regex matching in parallel while PCRE matches one regex at a time. For fair comparison with PCRE and PCRE2, we measure the total time for matching all regexes in a serial manner (e.g., match against one regex at a time), which would require passing the entire input for each regex. For Hyperscan and RE2, we measure two numbers – one for matching one regex at a time (RE2-s, Hyperscan-s), and the other for matching all regexes in parallel (RE2-m, Hyperscan-m). For testing, we use 1,300 regexes from the Snort Talos ruleset and 2,800 regexes from the Suricata ruleset.

Table 5 shows the result. For the Snort Talos ruleset, Hyperscan-s outperforms PCRE, RE2-s, and PCRE2 by 40.1x, 10.3x, and 2.3x, respectively. Hyperscan-m is 13.5x faster than RE2-m while it reveals 183.3x performance improvement over PCRE2! For the Suricata ruleset, Hyperscan-s shows 24.8x, 9.1x, and 1.8x speedups over PCRE, RE2-s, and PCRE2. Hyperscan-m outperforms RE2-m and PCRE2 by 8.4x and 6.9x, respectively. We do not report DFA-based PCRE performance as we find it much slower than the default operation with NFA-based matching [5]. RE2-s uses a DFA for regex matching (and fails on a large regex that requires an NFA), but it needs to translate the whole regex into a single DFA. In contrast, Hyperscan splits a regex into strings and FAs, and benefits from fast string matching as well as smaller DFA matching of the FA components, which explains the performance boost.

6.4 Real-world DPI Application

We now evaluate how much performance improvement Hyperscan brings to a popular IDS like Snort. We compare the performance of stock Snort (ST-Snort) and Hyperscan-ported Snort (HS-Snort) that performs pattern matching with Hyperscan, both with a single CPU core. ST-Snort employs AC and PCRE for multi-string matching and regex matching, respectively. HS-Snort keeps the basic design of Snort but it replaces AC and PCRE with the multi-string and single-regex matchers of Hyperscan. It also replaces the Boyer-Moore algorithm in Snort with a fast single-literal matcher of Hyperscan. With the Snort Talos ruleset, ST-Snort achieves 113 Mbps on our real Web traffic. In contrast, HS-Snort produces 986 Mbps on the same traffic, a factor of 8.73 performance improvement. We find that the main contributor for performance improvement is the highly-efficient multi-string matcher of Hyperscan as shown in Figure 11. In practice, we expect a much higher performance improvement if we restructure Snort to use multi-regex matching in parallel.

7 Evolution, Experience, and Lessons

Hyperscan has been developed since 2008, and was first open-sourced in 2013. Hyperscan has been successfully adopted by over 40 commercial projects globally, and it is in production use by tens of thousands of cloud servers in data centers. In addition, Hyperscan has been integrated into 37 open-source projects, and it supports various operating systems such as Linux, FreeBSD, and Windows. Hyperscan APIs are initially developed in C, but there are public projects that provide bindings for other programming languages such as Java, Python, Go, Node.js, Ruby, Lua, and Rust. In this section, we briefly share our experience with developing Hyperscan and lay out its future direction.

7.1 Evolution of Hyperscan

Hyperscan was developed at a start-up company, Sensory Networks, after a move away from hardware matching, which was expensive in terms of material costs and development time. We investigated GPU-based regex matching, but it imposed unacceptable latency and system complexity. As CPU technology advances, we settled at CPU-based regex matching, which not only became cost-effective with high performance, but also made it simple to be employed by applications.

Version 1.0. The initial version was released in 2008, with the intent of providing a simple block-mode regex package that could handle large numbers of regexes. Like other popular solutions at that time, it used string-based prefiltering to avoid expensive regex matching. However, the initial version was algorithmically primitive and lacked streaming capability (e.g., pattern matching over streamed data). Also, it suffered from quadratic matching time as it had to re-scan the input from a matched literal for each match.

Version 1.0 did include a large-scale string matcher (a hash-based matcher called "hlm", akin to Rabin-Karp [29] with multiple hash tables for different length strings) as well as a bit-based implementation of Glushkov NFAs. The NFA implementation allowed support of a broad range of regexes that would suffer from combinatorial explosions if the DFA construction algorithm was used. The Glushkov construction mapped well to Intel SIMD instructions, allowing NFA states to be held in one or more SIMD registers.

Version 2.0. The algorithmic issues and the absence of a streaming capability led to major changes to version 1.0, which became Hyperscan version 2.0. First, it moved towards a Glushkov NFA-based internal representation (the "NFA Graph") that all transformations operated over, departing from ad-hoc optimizations on the regex syntax tree. Second, it supported 'streaming' – scanning multiple blocks without retaining old data and with a fixed-at-pattern-compile-time amount of stream state. Support for efficient streaming was especially desirable for network traffic monitoring as patterns may spread over multiple packets. Third, it scanned patterns that used one or more strings, detected by a string matching pre-pass, followed by a series of NFA or DFA engines running over the input only when the required strings are found. This approach avoids the high risk of potential quadratic behavior of version 1.0, with the tradeoff of potentially bypassing some comparatively lesser optimizations if a regex could be quickly falsified at each string matching site.

Unfortunately, version 2.0 still had a number of limitations. First, we observed the adverse performance impact of prefiltering. Prefiltering did not reduce the size of the NFA or DFA engines even if a string factor completely separated a pattern into two smaller ones. This exacerbated the problem of a large regex that often needed to be converted into an NFA. As the system had a hard limit of 512 NFA states (dictated by the practicalities of a data-parallel SIMD implementation of the Glushkov NFA; more than 512 states resulted in extremely slow code), it often did not accommodate user-provided regexes when they were too large. Further, if prefiltering failed (i.e., when the string factors were all present), it ended up consuming more CPU cycles than naïvely performing the NFA engines over all the input.

Another serious limitation was that matches emerged from the system in an undefined order. Since the NFAs were run after string matching had finished, the matches from these NFAs would emerge based on the order of which NFAs were run first and no rigorous order was defined for when these matches would appear. Further confusing matters, the string matcher was capable of producing matches of plain strings ahead of the NFA executions. In fact, due to potential optimizations where NFA graphs might be split (for example, splitting into connected components to allow an unimplementably large NFA to be run as smaller components), it was even possible to receive duplicate matches for a given pattern at a given location. After an NFA is split into connected components and run in separate engines, no mechanism existed to detect whether these different components (which would be running at different times) might be sometimes producing matches with the same match id and location.

For example, a regex workload consisting of patterns $/foo/$, $/abc[xyz]/$ and $/abc[xy].*def|abc.z.*de[fg]/$ might first produce matches for the simple literal $/foo/$, then provide all the matches for the components of the pattern $/abc[xyz]/$, then provide matches for the two parts

of the alternation $abc[xy].*def$ and $abc.z.*de[fg]$ without removing duplicate matches for inputs that happened to match both parts of the pattern on the same offset (e.g. the input "$abcxxxxzdef$").

Versions 2.1 and 3.0. (The version 2.1 release series of Hyperscan saw considerable development and in retrospect should have merited a full version number increment) The limitation of prefiltering spurred the development of an alternate matching subsystem called 'Rose'. 'Rose' allowed both ordered matching, duplicate match avoidance, and pattern decomposition. This subsystem was maintained in parallel to the prefiltering system inherited from the original 2.0 design. Whenever it was possible to decompose patterns, the patterns were matched with the 'Rose' subsystem, which initially was not capable of handling all regular expressions.

FDR was developed during in the version 2.1 release series; it replaced the hash-based literal matcher ("hlm") with considerably performance improvements and reduction in memory footprint.

Eventually, by version 3.0, the old prefiltering system was entirely removed, as the Rose path was made fully general. Version 3.0 also marked an organizational change in that Intel Corporation had acquired Sensory Networks.

Version 4.0. Version 4.0 was released in October 2013 under an open-source BSD license to further increase the usage of Hyperscan, by removing barriers of cost and allowing customization. Many elements of Hyperscan's design continued to evolve. For example, the initial Rose subsystem had a host of special-purpose matching mechanisms that identify the strings separated by various common regex constructs such as $.*$ or $X+$ for some character class X. For example, it is frequently the case that strings in regexes might be separated by the .* construct (i.e. $/foo.*bar/s$). This is usually implementable by requiring only that "foo" is seen before "bar" (usually, but not always: consider the expression $/foo.*oar/s$). The original version of Rose had many special-purpose engines to handle these type of subcases; during the evolution of the system, this special-purpose code was almost entirely replaced with generalized NFA and DFA mechanisms, amenable to analysis and optimization, and were needed for the general case of regex matching in any case.

Version 5.0. Version 5.0, which is the latest version as of writing this paper, mainly focused on enhancing the usability of Hyperscan. Two key added features are support for logical combinations of patterns and *Chimera*, a hybrid regex engine of Hyperscan and PCRE [6]. As the detection technology of malicious network traffic matures, it often requires evaluating a logical combination of a group of patterns beyond matching a single pattern. To support this, the system now allows user-defined AND, OR, NOT along their patterns. For example, an AND operation between patterns /foobar/ and /teakettle/ required that both patterns are matched for input before reporting a match. Version 5.0 added Chimera, a hybrid matcher of Hyperscan and PCRE, brings the benefit of both worlds – support for full PCRE syntax while enjoying the high performance of Hyperscan. Lack of support of Hyperscan for full PCRE syntax (such as capturing and back-references) made it difficult to completely replace PCRE in adopted solutions. Chimera employs Hyperscan as a fast filter for input, and triggers the PCRE library functions to confirm a real match only when Hyperscan reports a match on a pattern that has features not supported by Hyperscan.

7.2 Lessons Learned

We summarize a few lessons that we learned in the course of development and commercialization of Hyperscan.

Release quickly and iterate, even with partial feature support. The difficulty of generalized regex matching often led Hyperscan to focus on the delivery of some capability in partial form. From a theoretical standpoint, it is unsatisfactory that Hyperscan still cannot support all regexes (even from the subset of 'true' regexes) and that the support of regexes with the 'start of match' flag turned on is even smaller. However, customers can still find the system practically useful despite these limitations. Despite the limited pattern support of version 2.0 and the problems of ordering and duplicated matches, there was immediate commercial use of the product even in that early form (use which made subsequent development possible). This lends support to the idea of releasing a "Minimal Viable Product" early, rather than developing a product with a long list of features that customers may or may not want.

Evolve new features over several versions, at the expense of maintaining multiple code paths. Academic systems are usually built for elegance and to illustrate a particular methodology. However, a commercial system must stay viable as a product while a new subsystem is built. For example, Hyperscan maintained both the 'prefilter' arrangement and the new 'Rose-based' decomposition arrangement in the same code base, resulting in considerable extra complexity. However, the benefits of having the new 'ordered' semantics (with additional powers of support for large patterns due to decomposition) outweighed the complexity cost. It was almost impossible that a small start-up could have managed to transition from one system to the other in a span of a single release, or to have simply not meaningfully updated the project for an extended time while working on a substantial update.

Commercial products may need to emphasize less interesting subcases of a task, or unusual corner cases. There was also considerable commercial pressure to be the best option at some comparatively degenerate subset of regex matching, or some relatively 'hard case'. Customers often wanted Hyperscan to function as a string matcher - sometimes even a single-string-at-a-time matcher! Other customers wanted high performance despite very high regex match rates (for example, more than 1 match per character). Such demands often force special optimizations that lack deep 'algorithmic interest', but are necessary for commercial success.

Be cautious of cross-cutting complexity resulting from customer API requests. One illuminating experience in the delivery of a commercially viable regex matcher was that customer feature requests for new 'modes' or unusual calls at the API level resulted in cross-cutting complexity that made the code base considerably more complicated (due to a combinatorial explosion of interactions between features) while rarely being reused by other customers. Features added in the 2.0 or 3.0 release series over time were not carried forward to the 4.0 series; we found that frequently such features were only used by a single customer (despite being made available to all).

Examples of two such features were "precise alive" (the ability to tell at any given stream write boundary whether a pattern might still be able to match) and an ad-hoc stream state compression scheme that allowed some stream states to be discarded if no NFA engines had started running. These features were complicated and suppressed potential optimizations as well as interacting poorly with other parts of the system.

7.3 Future Directions

Hyperscan is performance-oriented; future development in Hyperscan will still focus on delivering the best possible performance, especially on upcoming Intel Architecture cores featuring new instruction set extensions such as Vector Byte Manipulation Instructions (VBMI). Improvement of scanning performance as well as reduction of overheads such as the size of bytecodes, size of stream states and time to compile the pattern matching bytecode are obvious next steps.

Beyond this, adding richer functionality, including support for currently unsupported constructs such as generalized "lookaround" asserts and possible some level of support for back-references would aid some users. There is a considerable amount of usage of the 'capturing' functionality of regexes, which Hyperscan does not support at all (an experimental subbranch of Hyperscan, not widely released, supported capturing functionality for a limited set of expressions). Hyperscan could be extended to have enriched semantics to support capturing, which would allow portions of the regexes to 'capture' parts of the input that matched particular parts of the regular expression.

8 Conclusion

In this paper, we have presented Hyperscan, a high-performance regex matching system that suggests a new strategy for efficient pattern matching. We have shown that the existing prefilter-based DPI suffers from frequent executions of unnecessary regex matching. Even though Hyperscan started with the similar approach, it has evolved to address the limitation over time with novel regex decomposition based on rigorous graph analyses. Its performance advantage is further boosted by efficient multi-string matching and bit-based NFA implementation that effectively harnesses the capacity of modern CPU. Hyperscan is open sourced for wider use, and it is generally recognized as the state-of-the-art regex matcher adopted by many commercial systems around the world.

Acknowledgment

We appreciate valuable feedback by anonymous reviewers of USENIX NSDI'19 as well as our shepherd, Vyas Sekar. We acknowledge the code contributions over the years by the Sensory Networks team: Matt Barr, Alex Coyte and Justin Viiret. This work is in part supported by the ICT Research and Development Program of MSIP/IITP, South Korea, under projects [2018-0-00693, Development of an ultra-low latency user-level transfer protocol] and [2016-0-00563, Research on Adaptive Machine Learning Technology Development for Intelligent Autonomous Digital Companion].

References

[1] Google RE2. https://github.com/google/re2/.

[2] Hyperscan GitHub Repository. https://github.com/intel/hyperscan.

[3] ModSecurity. https://www.modsecurity.org/.

[4] nDPI. https://www.ntop.org/products/deep-packet-inspection/ndpi/.

[5] PCRE Manual Pages. https:/pcre.org/pcre.txt.

[6] PCRE: Perl Compatible Regular Expressions. https://www.pcre.org/.

[7] Perl-compatible Regular Expressions (revised API: PCRE2). https://www.pcre.org/current/doc/html/index.html.

[8] Snort Emerging Threats Rules 2.9.0. https://rules.emergingthreats.net/open/snort-2.9.0/rules/.

[9] Snort Intrusion Detection System. https://snort.org.

[10] Suricata: Open Source IDS. http://suricata-ids.org/.

[11] Talos Ruleset. https://www.snort.org/talos,.

[12] Aho, Alfred V. and Corasick, Margaret J. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975.

[13] Ricardo A. Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM (CACM)*, 35(10):74–82, 1992.

[14] Zachary K. Baker and Viktor K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2004.

[15] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2007.

[16] M. Becchi and P. Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proceedings of the ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2007.

[17] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2007.

[18] M. Becchi and P. Crowley. A-DFA: A Time- and Space-Efficient DFA Compression Algorithm for Fast Regular Expression Evaluation. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(1), April 2013.

[19] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral. Deep Packet Inspection as a Service. In *Proceedings of the ACM International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2014.

[20] Taylor-D. E. Brodie, B. and R. K. Cytron. A scalable architecture for high-throughput regular expression pattern matching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2006.

[21] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han. DFC: Accelerating string pattern matching for network applications. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[22] Chris Clark, Wenke Lee, David Schimmel, Didier Contis, Mohamed Koné, and Ashley Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Proceedings of the Workshop on Network Processors Applications (NP3)*, 2004.

[23] Christopher R. Clark and David E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, 2003.

[24] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the Colloquium, on Automata, Languages and Programming*, 1979.

[25] Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[26] Giordano-S. Procissi G. Vitucci F. Antichi G. Ficara, D. and A. Di Petro. An improved DFA for fast regular expression matching. *ACM SIGCOMM Computer Communication Review (CCR)*, 38(5), 2008.

[27] V.M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[28] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 317–328, 2012.

[29] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[30] T. Kojm. ClamAV. http://www.clamav.net/.

[31] Smith-R. Kong, S. and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2008.

[32] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and

J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the ACM SIGCOMM on Data communication (SIGCOMM)*, 2006.

[33] Turner-J. Kumar, S. and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, 2006.

[34] Janghaeng Lee, Sung Ho Hwang, Neungsoo Park, Seong-Won Lee, Sunglk Jun, and Young Soo Kim. A High Performance NIDS using FPGA-based Regular Expression Matching. In *Proceedings of the ACM symposium on Applied computing*, 2007.

[35] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating Snort IDS. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2007.

[36] J. Jang J. Truelove D. G. Andersen S. K. Cha, I. Moraru and D. Brumley. SplitScreen: Enabling efficient, distributed malware detection. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[37] E. Sitaridi, O. Polychroniou, and K. A. Ross. SIMD-accelerated regular expression matching. In *Proceedings of the Workshop on Data Management on New Hardware (DaMoN)*, 2016.

[38] R. Smith, C. Estan, and S. Jha. XFA: Faster Signature Matching with Extended Automata. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[39] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proceedings of the ACM SIGCOMM on Data communication (SIGCOMM)*, 2008.

[40] Y. E. Yang, W. Jiang, and V. K. Prasanna. Compact architecture for high-throughput regular expression matching on fpga. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2008.

[41] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman, and R. H. Kats. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2014.

[42] Xiaodong Yu, Bill Lin, , and Michela Becchi. Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence. *IEEE Journal on Selected Areas in Communications*, 32(10), October 2014.

Appendix

Algorithm 3 Dominant Path Analysis

```
Require: Graph G=(E,V)
 1: function DominantPathAnalysis(G)
 2:     dpath := {}
 3:     for v ∈ accepts do
 4:         calculate dominant path p[v] for v
 5:         if dpath = {} then
 6:             dpath := p[v]
 7:         else
 8:             dpath := common_prefix(dpath, p[v])
 9:             if dpath = {} then
10:                 return null_string
11:             end if
12:         end if
13:         strings := expand_and_extract(dpath)
14:     end for
15:     return strings
16: end function
```

The dominant path analysis algorithm finds the dominant path ($p[v]$) for every accept state (v), and find the common path of all dominant paths. The function, expand_and_extract(), expands small character-sets in the path, and extracts the string on the path.

Algorithm 4 Dominant Region Analysis

```
Require: Graph G=(E,V)
 1: function DominantRegionAnalysis(G)
 2:     acyclic_g := build_acyclic(G)
 3:     Gt := build_topology_order(acyclic_g)
 4:     candidate := q0
 5:     it = begin(Gt)
 6:     while it != end(Gt) do
 7:         if isValidCut(candidate) then
 8:             setRegion(candidate)
 9:             initializeCandidate(candidate)
10:         else
11:             addToCandidate(it)
12:             it := it + 1
13:         end if
14:     end while
15:     setRegion(candidate)
16:     Merge regions connected with back edge
17:     strings := expand_and_extract(regions)
18:     return strings
19: end function
```

The dominant region analysis builds an acyclic graph and sorts the vertices by the topological order. Then, it adds each vertex of the graph into a candidate vertex set,

and sees if the candidate vertex set forms a valid cut. If so, it creates a region. It continues to create regions by iterating all vertices. Finally, it merges the regions by back edges, and extracts strings from the merged region.

Algorithm 5 Network Flow Analysis

Require: Graph G=(E,V)
1: **function** NETWORKFLOWANALYSIS(G)
2: **for** $edge \in E$ **do**
3: $strings := find\_strings(edge)$
4: $scoreEdge(edge, strings)$
5: **end for**
6: $cuts := MinCut(G)$
7: $strings :=$ *extract and expand strings from cuts*
8: **return** $strings$
9: **end function**

The network flow analysis assigns a score to every edge and runs the "max-flow min-cut" algorithm. An edge is assigned a score inverse proportional to the length of a string that ends at the edge. So, the longer the string is, the smaller the score gets. Then, the max-flow min-cut algorithm finds a cut whose edge has the longest strings.