DFC: Accelerating String Pattern Matching for Network Applications

Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, Kyoungsoo Park, and Dongsu Han, *Korea Advanced Institute of Science and Technology (KAIST)*

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/choi

This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).

March 16–18, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-29-4

DFC: Accelerating String Pattern Matching for Network Applications

Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, Dongsu Han

KAIST

Abstract

Middlebox services that inspect packet payloads have become commonplace. Today, anyone can sign up for cloud-based Web application firewall with a single click. These services typically look for known patterns that might appear anywhere in the payload. The key challenge is that existing solutions for pattern matching have become a bottleneck because software packet processing technologies have advanced. The popularization of cloud-based services has made the problem even more critical.

This paper presents an efficient multi-pattern string matching algorithm, called DFC. DFC significantly reduces the number of memory accesses and cache misses by using small and cache-friendly data structures and avoids instruction pipeline stalls by minimizing sequential data dependency. Our evaluation shows that DFC improves performance by 2.0 to 3.6 times compared to state-of-the-art on real traffic workload obtained from a commercial network. It also outperforms other algorithms even in the worst case. When applied to middlebox applications, such as network intrusion detection, anti-virus, and Web application firewalls, DFC delivers 57-160% improvement in performance.

1 Introduction

Multi-pattern string matching is a performance-critical task for many middlebox applications that perform deep packet inspection (DPI), such as network intrusion detection systems (IDS) [18, 21], Web application firewalls [13, 17], traffic classification [7], and network censorship/surveillance [9, 10, 22]. These applications commonly apply pattern matching to select flows or packets of interest that are subjected to further extensive processing. The common practice for Web application firewalls and IDS is to initially perform pattern matching on the traffic and apply regular expression matching to a relatively small number of packets that contain the string patterns [18, 21, 39, 66]. Many studies have shown that string pattern matching is one of the primary performance bottlenecks for these systems [25, 39, 48, 66, 67].

The key challenge to multi-pattern string matching is that its performance requirement has increased dramatically (e.g., from multi-Gbps [35] to multi-10s of Gbps [6]), outpacing the performance of existing solutions [39, 66]. The popularization of cloud-based third-party middlebox services [58], such as CloudFlare and Akamai's Web application firewall (WAF), requires that they handle large amounts of traffic efficiently [3, 5, 66].

Multi-pattern matching algorithms for network applications must satisfy three requirements: 1) they must support exact string matching to ensure correctness, while identifying the patterns matched, 2) they must support small and variable size patterns, ranging from a single byte to hundreds of bytes [8, 19], and 3) it must provide efficient online lookup against a stream of data (e.g., network traffic) as opposed to batched processing.

The classic Aho-Corasick (AC) algorithm [24] satisfies these requirements, and therefore is used by many applications, including intrusion detection systems, such as those of Snort [18] and Suricata [21], and Web application firewalls, such as ModSecurity [13] and CloudFlare's WAF [66]. AC constructs a deterministic finite automaton (DFA) based on the pattern set and is known to deliver asymptotically optimal performance. However, the main problem is that it references memory frequently and causes a large number of cache misses, resulting in poor performance. In particular, the size of DFA it constructs increases linearly with the number of states and causes severe performance degradation [53]. Although many efforts have been attempted to reduce the memory footprint, they come with undesirable performance trade-offs because they often require additional computation and/or memory lookups [20].

This paper presents DFC, a memory-efficient and cache-friendly data structure designed to deliver high performance. Our central tenet is that to obtain high performance we must minimize CPU stalls and maximize instruction level parallelism, while reducing the amount of per-byte operations and memory lookups. However, achieving this while satisfying the three requirements is not trivial. In particular, it is especially difficult to support exact matching with *short* and *variable* size patterns. Furthermore, its worst case performance must also be better than that of other algorithms.

To achieve our goal, DFC combines a number of small, efficient data structures by leveraging two key ideas. First, at its heart is a small data structure, called direct filter (DF) that, using a small sliding window, is designed to quickly reject parts of text that will not generate a match. It increases instruction-level parallelism by avoiding data

	1-4 byte	5-8 byte	>9 byte	Total
ET-PRO May 2015	7.3K (28%)	6.7K (26%)	12.1K (46%)	26.2K
ET-Open May 2015	5.3K (30%)	4.6K (26%)	7.8K (44%)	17.7K
Snort VRT 2.9.7.0	1.2K (21%)	1.0K (17%)	3.6K (62%)	5.9K
ModSecurity 2.2.9	0.7K (13%)	1.7K (32%)	2.9K (55%)	5.2K

Table 1: String length distribution in the pattern set

dependencies in the critical path and occupies a small memory footprint (8 KB) for cache efficiency. Moreover, its lookup does not require hash computation. Second, to support exact matching, we use multiple layers of direct filters designed to classify patterns based on their lengths and to filter out the window in a progressive fashion. The key premise is that the longer the pattern is the more one has to inspect to reduce false positives.

Our evaluation shows that DFC outperforms existing solutions by a large margin. In particular, DFC delivers 2.0-2.5 times the performance of AC with 1-26K variable size patterns on real traffic workloads. DFC references memory 1.8 times less frequently and incurs 3.8 times fewer L1 data cache misses. Its memory footprint is four times smaller than that of AC with 26K patterns. Even under malicious and adversarial workloads, DFC outperforms AC by a factor of 1.7 to 2.0. Finally, we implement four applications that use DFC—IDS, Web application firewalls, traffic classification, and anti-virus—and demonstrate that DFC improves their performance by 57 to 160%.

In summary, we make the following contributions:

1. **New algorithm:** We present a new algorithm for exact multi-pattern string matching that gracefully handles short and variable size patterns, and also works well with a stream of network traffic.
2. **Prototype and evaluation:** We evaluate DFC using a variety of deep packet inspection (DPI) patterns from IDS, anti-virus software, and Web application firewalls under real and synthetic workloads.
3. **Real-world applications:** We show four applications of DFC. When applied to intrusion detection, DFC delivers up to 2.3X the performance of the next best solution, in real traffic from a commercial ISP. When applied to Web application firewalls, DFC delivers 1.9X the performance. Moreover, DFC improves the performance of traffic classification and anti-virus by 60% and 75%, respectively.

2 Motivation

Why is it important? Network applications, such as network intrusion detection, traffic classification, and Web application firewalls, specify pattern signatures using regular expression and strings [48, 53]. While regular expression is much more *expressive*, matching multiple regular expressions [60] is much more *expensive*. Matching a *single* regular expression (regex) against network traffic is 4X slower than multi-string matching with 2.4K patterns [39][§4.5]. Matching multiple regex patterns by constructing a single extended automaton (XFA [60, 61]) is at least two orders of magnitude slower than multi-string matching.[1] Due to the distinct tradeoff between performance and expressiveness of the two, string pattern matching is commonly used to accelerate regex-matching. String matching can filter out the vast majority of input early on and specifies a small set of candidate regex patterns to inspect [13, 18, 21, 31]. Typically, regex patterns contain at least a few bytes of string [60][Table 2]. SplitScreen [31] leverages this to extract strings from regex patterns and uses them as a pre-filter to improve the performance. For the same reason, it is strongly recommended that IDS signatures contain string fields [23].

String matching is also used for protocol classification/identification and for testing the presence of keywords. Because string-pattern matching is a critical building block for network applications that inspect payloads, it is reported that string matching is one of the most expensive operations in Web application firewalls [66] and accounts for 70 to 80% of CPU cycles for IDS [25, 39].

Today, middlebox services offered over the cloud (e.g., CloudFlare's WAF [66]) allows customers to easily sign up for these services. One of the reasons for the widespread use of WAF is that it is one of the two ways to meet Payment Card Industry (PCI) Data Security Standards (DSS) requirement 6.6 for Web sites that take credit cards [1, 16]. [2] Decrypted traffic is inspected by Note, a Web application firewall (WAF), to protect the Web service. For this, some WAFs [13] run as part of the Web server, whereas cloud-based WAF services integrate itself with existing SSL acceleration service. We believe that popularization of cloud-based middlebox services and the security regulations/best practices make high-performance multi-pattern matching more important.

Workload: Three key characteristics define our target workload. First, patterns can appear *anywhere* in the text. Second, the number of patterns is large, typically on the order of 10K patterns, and accumulates over time as new patterns are discovered. Finally, patterns are typically *short* and of *variable* size.

Table 1 shows the pattern-length distribution from four popular rulesets for IDS and Web firewalls. Over 54% of patterns are 8 bytes or shorter in the ET-Pro®ruleset, a popular commercial pattern set for IDSs [8].

Why is it difficult? We categorize existing approaches into three classes and show that the characteristics of our workload make each of these algorithms rather ineffective.

[1]Note, the main benefit of XFA is that it solves the state explosion problem of DFAs. But its matching performance is still lower than that of DFA. Although it is an indirect comparison, XFA's multi-regex matching takes hundreds of CPU cycles per byte of input [60, 61], whereas multi-string matching takes a few cycles.

[2]The other option is to conduct application vulnerability security reviews of all Web applications in use.

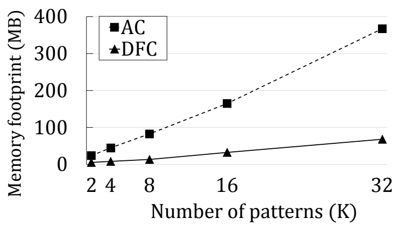


Figure 1: Memory footprint of AC and DFC

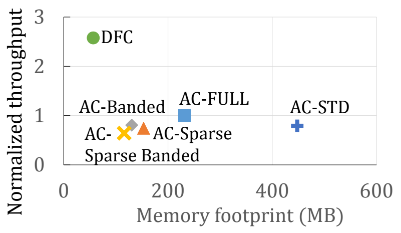


Figure 2: Throughput and memory footprint

	DF	DFC	AC
# L1-D cache load	2.0	4.6	8.4
# L1-D cache misses	0.003	0.282	1.070
# L2 cache misses	0.002	0.036	0.190
# L3 cache misses	0.0019	0.011	0.017

Table 2: Number of memory accesses and cache misses per byte of input text

Aho-Corasick-based algorithms construct finite state automata (based on the pattern strings) that consume each character of the input text sequentially. AC is by far the most popular multi-string matching algorithm used for deep packet inspection. While it achieves asymptotically optimal performance, in practice, its performance is far from ideal because it generates frequent cache misses due to its large memory footprint [53]. As shown in Figure 1, its memory footprint increases dramatically as the number of patterns increases. [3] For a reasonably large pattern set, the data structure does not fit in a CPU cache, resulting in a severe performance penalty.

Variants of AC [20, 55] that try to compress the state transition tables come at the cost of decreased performance. Figure 2 illustrates the memory footprint and performance trade-offs of five commonly used AC variants in Snort [20, 55], in contrast with those of DFC. For AC implementations, the memory footprint and the throughput are highly correlated. As a result, Snort defaults to AC-FULL [20], which achieves the highest throughput at the cost of the largest memory footprint.

The AC algorithm also frequently accesses memory because it examines the state table for every byte of text. Table 2 shows the average number of data accesses (measured using # of L1 data cache accesses) and the number of L1 data cache misses per byte of input text, while matching randomly generated text with 26K patterns from the ET-Pro ruleset.[4] AC accesses the memory at least five times on average per byte of input text, consisting of access to: (1) the state machine, (2) the input text, (3) the case translation table (for case insensitive lookup), (4) the state transition table for looking up the next state, and (5) a field that indicates whether the next state indicates a valid match. Note, DFC reduces the memory access frequency by a factor of 1.8 as shown in §5.4.

Heuristic-based algorithms try to achieve sub-linear time complexity by advancing the sliding window by multiple characters using the "bad character" and "good suffix" heuristics [30]. For example, the Wu-Manber algorithm [71] leverages the "bad character" heuristics. After examining a block of characters, it looks up a shift table that indicates by how many bytes it can shift the sliding window. The shift table is constructed so that when a block of characters never appear in any of the patterns, the algorithm skips the entire block and advances the sliding window by multiple characters. However, this has two main limitations. First, heuristics works well when the pattern is long and its size is fixed, but is not effective with *short patterns* [48, 71]. With short patterns, the shift distance becomes small because the block size has to be less than or equal to the shortest pattern, which is one byte in our workload. Second, the algorithm inherently has data dependencies that make it difficult to leverage the performance characteristics of modern CPUs. Until it retrieves the shift value from memory, it cannot determine the next window to examine. This limits the instruction-level parallelism, results in frequent instruction pipeline stalls, and makes prefetching very difficult. As a result, heuristics like Wu-Manber introduce severe performance penalty in practice, as we show in §5.4. [5]

Hashing-based algorithms compare the hash of a text block with the hash of the pattern. They are designed to quickly filter out non-matching text using its hash values, but have several practical limitations. First, they introduce false positives and thus require additional processing to ensure exact pattern matching [53]. Second, when the patterns are of variable size, the text block to hash must be shorter than or equal to the *shortest* pattern to avoid false negatives, which makes the algorithm ineffective under our workload. A common solution is to use hashing for long patterns and to fall back to traditional approaches (e.g., Aho-Corasick) for short patterns. For example, an exact matching algorithm based on feed-forward Bloom filters (FFBF) [53] applies the hash-based approach for patterns of size greater than 19 bytes, but uses AC for the remaining patterns. Finally, they require multiple expensive hash computation for *every sliding window*. Bloom filters typically use multiple hash functions applied on every sliding window. Even if a rolling hash function [53] is to incrementally calculate the hash values, it is far more expensive (than a simple lookup) as we show in §5.4.

[3] The patterns were taken from Snort and ET-Pro rulesets [8, 18].

[4] The performance statistics are obtained using *perf* and Intel Performance Counter Monitor [12].

[5] Note earlier versions of Snort once used a modified Wu-Manber algorithm.

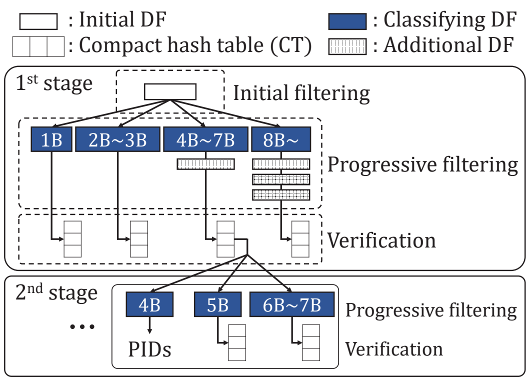


Figure 3: DFC design overview with an example configuation

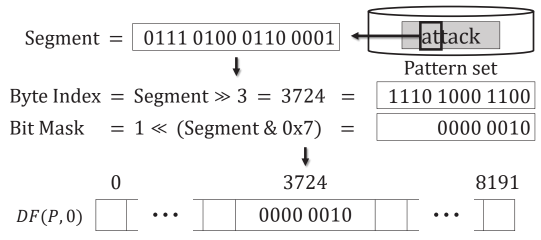

Segment = 0111 0100 0110 0001
attack
Pattern set
Byte Index = Segment ≫ 3 = 3724 = 1110 1000 1100
Bit Mask = 1 ≪ (Segment & 0x7) = 0000 0010
0
3724
8191
DF(P, 0)
0000 0010

Figure 4: An initialization example of Direct Filter

3 Design

To overcome the limitations of existing approaches, our main approach is to develop an efficient and simple primitive that we use as a key building block. Making the basic component efficient is critical because the nature of the workload—matching a large number of small and variable sized patterns—forces us to examine *every byte* of input text. We show that one can design an efficient string matching algorithm out of a simple primitive. We also show that our algorithm not only delivers the best average case performance but also is robust across many workloads, including the worst case.

Figure 3 shows the overall design of DFC that consists of three logical parts:

- **Initial filtering** phase employs a simple filter that eliminates windows of input text that fail to match any string patterns. Only the input text window that is not filtered advances to the next phase. We describe the filter design in §3.1.
- **Progressive filtering** phase first determines the approximate lengths of potentially matching patterns. We group patterns into pattern size groups (e.g., 1B, 2-3B, 4-7B, 8+B) and use *classifying filters* to determine the groups that the current window belongs to. We then apply additional filtering proportional to their lengths using *additional filters*. We use the same type of filters for both purposes. We describe the design of progressive filtering in §3.2.
- **Verification** phase verifies whether the input generates an exact match by comparing the text with actual patterns. This is required because the previous phases do not completely eliminate false positives. For example, if there are patterns 'AABB' and 'CCDD', the initial DF can be set for 'AA' and 'CC' and an additional DF can be set for 'BB' and 'DD'. In this case, an input sequence 'AADD' will pass both filters even though it is not a match. For verification, we use a compact hash tables for each pattern size class. Depending on the pattern size group determined by the previous phase, it inspects a different table. Its lookup is efficient because the progressive filtering phase significantly reduces both the false positives and the set of potentially matching patterns to inspect. We describe the design in §3.3.

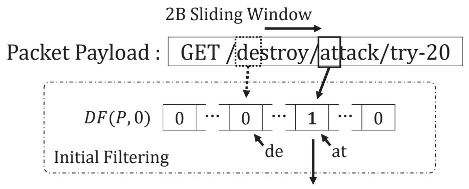
Two-stage hierarchical design: Our algorithm uses progressive filtering and verification in two stages as shown in Figure 3. The first stage of progressive filtering classifies patterns by their lengths in a coarse grained manner. After roughly determining the pattern lengths, it looks up a hash table for the pattern class in the verification phase. However, this may be inefficient if there are many hash collisions. Some buckets may contain many patterns due to the skewed popularity of patterns. Normally, hash collisions can be controlled either by using a more uniform hash function or adjusting the size of hash table. However, we find that hash collisions are caused because real-world pattern sets often contain common string prefix or segments, and some prefixes (e.g., HOST) appear very frequently (e.g., more than 100 times). For example, consider a hash table indexed by 4 bytes and holds patterns of size 4–7B. If many of them start with the same prefix (e.g., HOST), then all of them will go to the same bucket. This dramatically worsens the worst case performance because we must perform verification. However, in this case, increasing the hash table size or using a better hash function does not help, but only increases the overhead.

To remedy this, we selectively employ 2nd-stage progressive filtering on a per-bucket basis. When a bucket contains a large number of patterns, we apply a 2nd stage of progressive filtering for finer-grained classification as shown in Figure 3. For buckets with many collisions, the 1st stage verifies exact matching with the prefix (minimum pattern length in the group) and the 2nd stage only examines the string that follows the prefix.

3.1 Filter Design

DFC design relies on an efficient filter primitive, called direct filter (DF), for filtering and classification.

Direct filter is a bitmap indexed by several bytes of input text. For a 2B DF, a 2B sliding window from the input is treated as a 16-bit unsigned integer and used as a bit index to DF. Each bit tells whether the string containing the 2B window can generate a match with any patterns. For most cases, we use 2B windows so that the DF fits in lower levels of the CPU cache. A 2B DF is initialized by taking a


2B Sliding Window
Packet Payload : GET /destroy/attack/try-20
DF(P, 0) 0 ··· 0 ··· 1 ··· 0
de
at
Initial Filtering
Figure 5: Initial filtering example

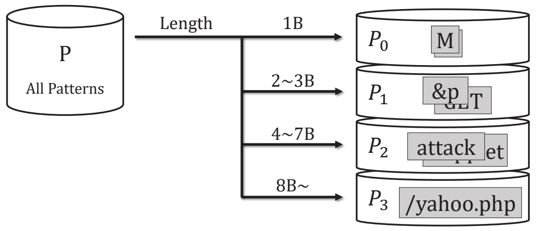


Figure 6: A classification example of patterns by lengths

single two-byte fragment (e.g., first two bytes) from each string pattern and setting the bit indexed by the two bytes, while the rest of the bits are set to zero. Figure 4 shows an initialization example of DF for a pattern 'attack'.

During the initial filtering phase, we take every two-byte window of the input text and look up whether the corresponding bit is set in the DF as shown in Figure 5. A DF lookup performs three bitwise operations (two SHIFTs and one AND) and a single memory reference. If the corresponding DF bit is not set, it indicates that the part of input text that contains the two-byte window cannot generate a match. Thus, no pattern of interest can appear at this location of the input text. We then advance the window by one byte to examine the next window. If the corresponding bit is set, it indicates that the location of text potentially contains patterns of interest. Only then, we inspect in more detail the sequence starting from the current window.

The initial filtering phase uses a single DF to filter out most of the innocent input text. In our experiment with real traffic and 26K patterns from the ET-Pro ruleset, 94% of windows are filtered out in this phase.

Size of initial DF: Similar to other filters (e.g., Bloom filter [29]), the size of DF determines the tradeoff between the number of cache misses and false positives; DF is actually a special case of Bloom filter that uses a single identity function as the hash function. For most cases, we use a two-byte indexed 8KB DF (2B DF) to achieve a balance. If we use one-byte indexed DF (1B DF), the size of DF reduces to 256 bits, but the rate of false positives will be 256 times higher on average. This would, for example, cause 34.8% of windows with 26K ET-Pro rules (compared to 3.6% for 8 KB DF assuming a uniformly random input) to advance to the next phase. In contrast, using a three-byte indexed DF (3B DF) further reduces the false positives, but the size of DF increases to 2 MB. Because initial DF lookup is performed very frequently, we would like to minimize the cost by making DF fit inside lower levels of the CPU cache (2 MB easily exceeds the typical size of L2 caches).

Our evaluation shows 2B DF delivers better performance for workload that contains up to a few tens of thousands of patterns; using a 3B DF is actually up to 18% slower as we show in §5.4. This is because L3 cache latency is four to seven times higher than that of L2 cache [47]. Thus, we use 2B DF in most cases, but use 3B DF when there are millions of patterns (e.g., ClamAV ruleset). We quantify this tradeoff in more detail in §5.4.

Initialization: To make initial filtering more effective, we try to minimize the number of bits set. A negative correlation exists between the number of bits set in the initial DF and its performance because the more the bits are set, the more windows pass through. We observe that minimizing the number of bits set in the initial DF delivers up to 10% performance benefit than a native solution (i.e., using the first two byte) in the real traffic workload. We use the following heuristics to achieve this. First, we classify patterns into their lengths as we do for the progressive filtering phase. We use the same offset for each class to avoid looking up additional state in progressive filtering. In particular, we examine the pattern class size in the increasing order (i.e., from short patterns to long patterns) and, for each class, we choose the offset of the two-byte segment from each class so that it minimizes the number of additional bits set.

If 1-byte patterns exist, we enumerate all two-byte segments that start with the single byte in the pattern. Thus, a 1-byte pattern sets 256 bits in the DF. Each pattern of size two bytes or more sets a single bit if they are case sensitive. If they are case insensitive, we enumerate all cases, setting at most four bits per string pattern.

3.2 Progressive Filtering

The main idea of this phase is to progressively eliminate false positives to take small steps towards exact pattern matching using multiple layers of DFs. The key insight we leverage is that the longer the pattern is, the more one has to compare it against the input text to reduce false positives. For example, when the text is filtered using a single DF, a two-byte pattern does not produce any false positive, but four-byte patterns can produce as many as 2^{16} false positive patterns.

Thus, we first determine the pattern lengths that the current window of input might match and apply different amounts of additional filtering proportional to the lengths For example in Figure 3, we use three additional DFs for 8+ bytes but only use a single additional DF for 4-7 bytes in the 1st stage progressive filtering. As the algorithm progresses, the set of the potentially matching patterns also reduces, and the likelihood of generating a match increases exponentially. When the scope is sufficiently

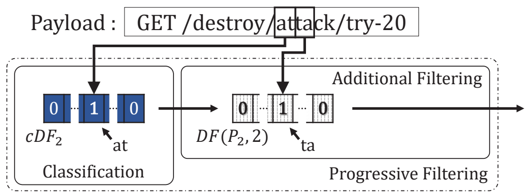


Figure 7: Progressive filtering example

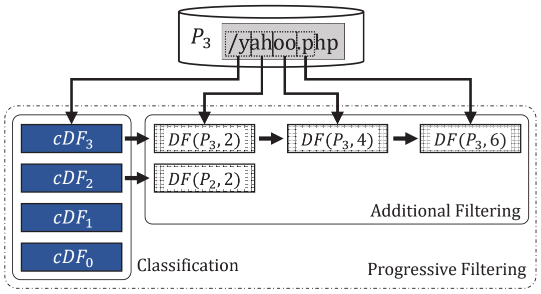


Figure 8: An initialization example of cDFs and DFs in Progressive Filtering

reduced, we perform exact comparison in the verification phase. Progressive filtering consists of two steps: pattern length classification and additional filtering.

Classification: During initialization of DFC, we classify the string pattern set P into pattern classes (P_i's) according to their lengths—P_0 contains the shortest patterns and $P_1,...,P_n$ contains longer patterns in increasing order. Figure 6 shows an example with four classes (e.g., P_0 contains 1-byte patterns, and P_3 contains patterns of 8 bytes or longer). For each class, P_i, we create a classifying direct filter, cDF_i, whose bits are set using the patterns in P_i. For patterns in P_i, to set the cDF_i, we use the same two-byte segment that was used to set the initial DF.

At run time, we look up the classifying DFs (cDFs). Each cDF determines whether the window might match the patterns in the class. For example if the corresponding bit of cDF_i is set, we know that we must inspect the pattern class P_i. Note the classification is *not* mutually exclusive; multiple pattern classes can match because patterns from two different classes may share the same two-byte segment.

Additional filtering: When the corresponding bit in cDF_i is set, we further filter the input sequence that follows the current two-byte window with additional DFs. These additional DFs are designed to inspect different two-byte text segments in the pattern. We use additional DFs, $DF(P_i,O_k)$'s, to test whether the input text might match patterns in P_i, where O_k denotes the offset from the initial window—e.g., the initial DF that inspects the first two byte can be represented as $DF(P,0)$. $DF(P_2,O=2)$ inspects the third and fourth bytes (two bytes at offset two) against patterns in P_2 as depicted in Figure 7; The following two-byte text segment is inspected using additional DF ($DF(P_2,2)$) after passing through classifying

Algorithm 1: Progressive Filtering and Verification

```
# @param  cDF  Array of Classifying Direct Filters
# @param  buf  Input text
# @param  pos  Position of window
def Progressive_Filtering(cDF, buf, pos):
    for $i = 0$ to $\#of\ cDF$:
        if Corresponding bit of buf at pos in cDF[i] is set:
            if Pass through additional DFs:
                CT ← corresponding hash table
                pos ← corresponding position
                Verification(CT, buf, pos)

# @param  CT   (Compact) Hash Table
# @param  buf  Input text
# @param  pos  Position of window
def Verification(CT, buf, pos):
    H ← hash value of buf at pos
    Bucket ← CT[H]
    Second ← Second stage flag in Bucket
    if Second is true:
        cDF ← array of cDFs in Bucket
        pos ← corresponding position
        Progressive_Filtering(cDF, buf, pos)
    else:
        for PID ∈ Bucket:
            Perform exact matching for PID with buf at pos
```

DF (cDF_2). $DF(P_2,O=2)$ is thus created from using two bytes from P_2 that correspond to offset O. Figure 8 depicts how cDF and additional DFs are initialized for a pattern from the pattern class P_3. Each DF performs filtering and we advance to the next phase only if the window passes through all DFs in the sequence. Algorithm 1 shows the pseudo code for progressive filtering and verification.

Additional filtering is an optimization to avoid the verification phase where we perform hash table lookup and exact string comparison. It is only beneficial, if the benefit outweighs the cost of additional DF lookups. In general, the longer the pattern size, we inspect longer segments that follow the current window. However, because each additional lookup adds decreasing marginal benefit, we use a small number of additional filters. We discuss the configuration issue at the end of the subsection. As we show in §5, each additional filtering is effective, and once the input text passes through this phase, it is much more likely to generate a match.

Optimizations: To minimize the average memory lookup, we perform two optimizations. First, similar to the initial DF, we carefully choose the offsets of each additional DFs to filter text as much as possible. For this, we choose an offset that minimizes the number of bits set for each DF. This is done when the DFs are initialized. To reduce the search space, we first identify non-overlapping two-byte segments and greedily select the offset. Second, we change the order of DFs we inspect so that the most effective filter comes early. Note that each sequence of DFs that do not share a parent (e.g., cDF(P_3,0) $\rightarrow$ DF(P_3,2) and cDF(P_2,0) $\rightarrow$ DF(P_2,2)) is independent, and the ordering of DFs within a sequence does not affect the correctness of the algorithm. However, if we first look up the one with the smallest number of bits set, the number of average memory lookups can decrease. Thus, we

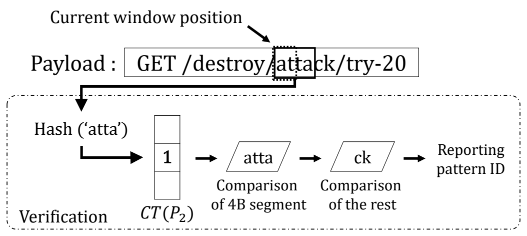


Figure 9: Verification example

order DFs in a non-decreasing order of the number of bits set.

Configuration: The number of pattern classes and their ranges are parameters that can be configured. In our evaluation with DPI patterns in Table 1, we classify the length in multiples of two—P_i holds patterns of length 2^i to $2^{(i+1)} - 1$, and the last class holds the rest of the patterns. Figure 6 illustrates our configuration. P_0 holds patterns of one byte and P_1 holds patterns of two to three bytes. We classify patterns into four classes by their lengths: one-byte (P_0), two to three bytes (P_1), four to seven byte (P_2), and 8+-byte (P_3) string patterns. In our implementation, we use an additional filter for P_2 and three additional DFs for longer patterns (≥ 8).

The classification we use in our implementation is determined empirically and can be tuned depending on the pattern set; e.g., when there is no pattern of size [2,3], having such a classification is wasteful. Tuning the classification parameters involves in one time cost that can be done offline when the pattern size distribution has changed significantly since the last update. On the one hand, having a fine-grained classification or a large number of classes helps in significantly reducing the false positives in the progressive filtering phase and the number of patterns to inspect in the verification phase. This is because we can only filter up to the minimum length in each pattern set to ensure that there is no false negatives. However, fine-grained classification increases overhead because each DF requires an additional memory lookup. Thus, we must strike a balance in choosing the granularity of the pattern classes. We evaluate the tradeoffs of having a finer or coarser classification in §5.

3.3 Verification

The final verification phase ensures exact matching by comparing the text with the actual patterns in the pattern class. To perform exact matching, we create a (compact) hash table, $CT(P_i)$, for each pattern size class, P_i. These tables are indexed by a hash of text fragments whose lengths correspond to the shortest pattern in P_i; e.g., for pattern class P_2 of size 4 to 7, a hash table $CT(P_2)$ is indexed by a hash of four-byte segment. Each bucket in the hash table holds a list of a pattern ID (PID), a text segment of the pattern, and a pointer to the rest of the pattern text.

Rulesets	ET-Pro (May 2015) Snort VRT 2.9.7.0	ET-Open (May 2015) ModSecurity CRS 2.2.9
Input workload	(1) Random payload (2)Real traffic traces from a commercial ISP	

Table 3: Patterns and inputs

Total volume	68 GB
Number of packets	89,043,284
Number of HTTP packets	77,582,806
Number of TCP sessions	2,213,975
Number of HTTP sessions	1,869,208
Capture duration	57 min 16 sec
Average packet size	757 B
Average flow size	30 KB

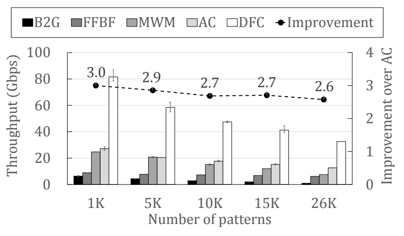
Table 4: Statistics of traffic traces from a commercial ISP

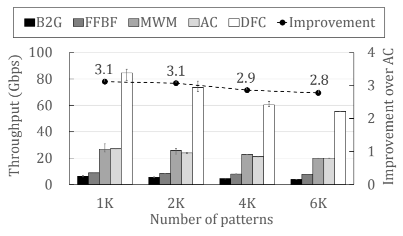
During verification, we compare the text segment with all the pattern segments in the bucket. If a match is found, we compare with the rest of the string segments to ensure an exact match. If exact matches are found, we report the PID and the offset within the input text that generated the matches; Figure 9 illustrates verification process for 'attack' pattern from pattern class P_2. A hash table $CT(P_2)$ is indexed by a hash of four byte segment 'atta' because a length of the shortest pattern in P_2 is four. Then, the four byte segment is compared to four byte segment in the bucket. Because they are same, the rest of the pattern 'ck' is compared to the following two-byte segment from the payload and PID of the pattern is reported. On the one hand, using a simple hash function is good enough because the number of possible patterns that can actually reach this phase is significantly reduced due to progressive filtering.

We adjust the size of the table so that on average a small number of PIDs are present (e.g., < 0.1 PIDs). However, because pattern strings commonly share some popular string segment hash collisions may be high for some buckets as explained in §3. If the number of PIDs in a bucket exceeds a threshold, we apply the second stage on a per-bucket basis as represented in Algorithm 1. In this case, the 1st stage examines up to the minimum length pattern in its size classification, and the 2nd stage verifies the rest. In the 2nd stage progressive filtering, we perform a more fine-grained classification and verify the rest of the text in the second stage of verification (Figure 3). For example, for P_2 that holds patterns of size [4,7], the 2nd stage progressive filtering divides it into three sub-classes. The 1st stage examines up to four bytes and the 2nd stage examines the rest.

4 Implementation

DFC is implemented in 2.4K lines of C code. For comparison, we use AC and modified Wu-Manber (MWM) algorithm implementations in Snort, b2g algorithm (2-gram implementation of the SBNDMq) [36] implementation


(a) ET-Pro rulset (May 2015)


(b) Snort VRT 2.9.7.0 ruleset

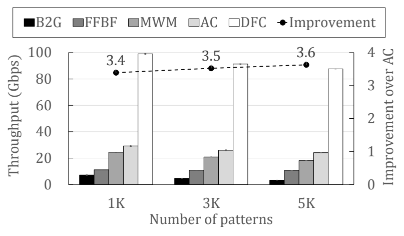
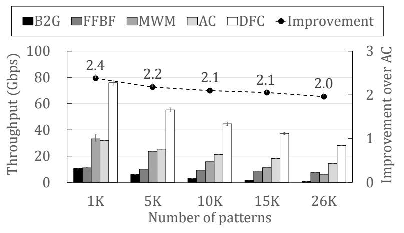
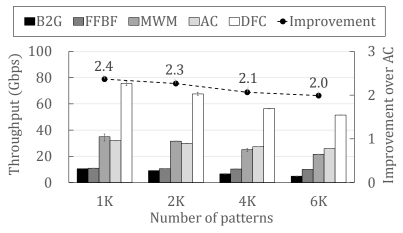

(c) ModSecurity 2.2.9 ruleset

Figure 10: Standalone performance benchmark of AC and DFC under random packet workload.


(a) ET-Pro rulset (May 2015)


(b) Snort VRT 2.9.7.0 ruleset

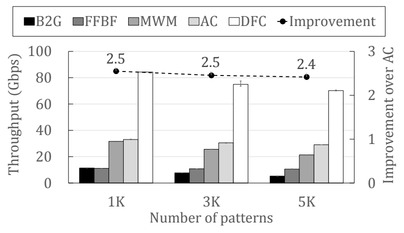

(c) ModSecurity 2.2.9 ruleset

Figure 11: Standalone performance benchmark of AC and DFC using packet contents from real traffic.

from Suricata, and FFBF [53]. In FFBF, short patterns are processed by AC and long patterns are processed by FFBF. FFBF works as a filter and outputs a subset of patterns and input set that can potentially match. Then, it uses AC to remove false positive [53]. To demonstrate the real-world benefit, we apply DFC to four different applications: intrusion detection system, Web application firewall, HTTP traffic classification, and anti-virus. For IDS, we choose an optimized version of Snort, Kargus-CPU, that uses high performance packet I/O and lightweight data structures [39]. Note, we do not use its GPU module, but only use its CPU version. Its pattern matching algorithm is almost identical to that of Snort. We replace AC with DFC by modifying 82 lines of code. For Web application firewalls, we apply DFC to ModSecurity, a popular open source implementation. ModSecurity scans through HTTP requests and responses to match against malicious patterns. For HTTP traffic classification, we use nDPI [7]-like traffic classification for HTTP traffic that looks for popular domain names in the HOST field of the request header. This classifies which application (e.g., Netflix, YouTube) generated the traffic. Note, HTTP consists of 75% of all downstream bytes in modern cellular networks [70]. For these three implementations, we thoroughly conduct correctness tests by comparing the result from DFC with result from AC using two types of input: randomly generated input and a 68GB traffic trace from a commercial cellular ISP. We find that the DFC produces output identical to AC. Finally, for anti-virus, we modify ClamAV [43] version 0.98.7.

5 Evaluation

We answer four questions about DFC in this section:

- How does it compare with existing algorithms under a variety of workloads?
- How much gain does it provide when applied to middlebox applications?
- How does each component affect performance, and what are the trade-offs involved in parameter settings?
- How does its direct filter compare against other primitives?

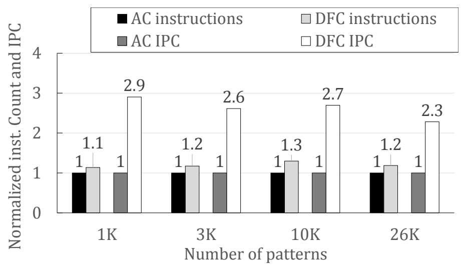


Figure 12: Instruction count and IPC from the benchmark for real traffic with ET-Pro ruleset

5.1 DFC Performance Benchmark

We compare the performance of DFC with that of AC, modified Wu-Manber (MWM), FFBF, and b2g across various workloads and configurations. We empirically choose a window size of FFBF to 32B and a size of bloom filter to 1MB and use 4 hash functions, showing the best performance in the workloads and configurations. B2g does not detect one byte patterns because it processes 2 characters as a single character.

To evaluate the performance on a pure algorithmic basis, we disable network I/O and feed input directly from memory. We use four different DPI patterns and two kinds of input workloads described in Table 3. The DPI patterns are taken from popular IDSs and a Web firewall, and the input workloads are either randomly-

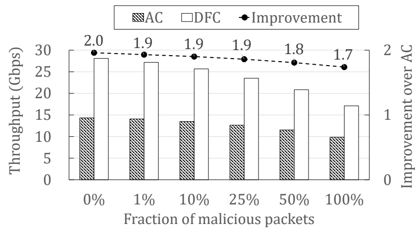


Figure 13: Performance with malicious traffic (ET-Pro 26K rules)

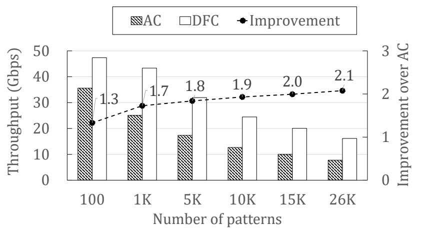


Figure 14: Pathological workload: Input purely consists of all two-byte segments used to set initial DF

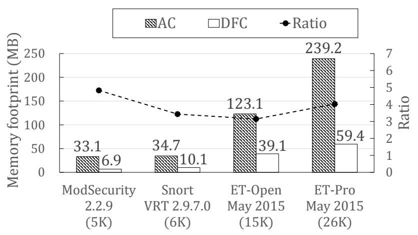


Figure 15: Memory footprint comparison

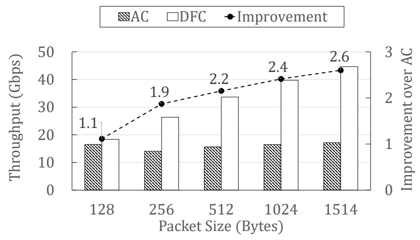


Figure 16: End-to-end IDS performances under synthetic traffic (Snort VRT 6K rules)

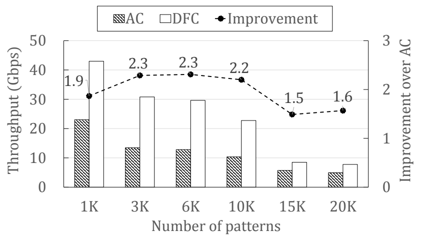


Figure 17: End-to-end IDS performances with traffic traces from a commercial ISP

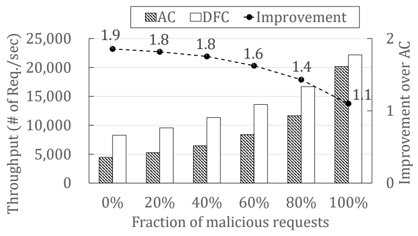


Figure 18: Performance of Apache with ModSecurity with malicious HTTP requests

generated or taken from a real traffic dump from a 10 Gbps LTE backhaul link at a commercial cellular ISP. Table 4 summarizes the statistics of the real traffic trace. Unless specifically noted, the results are measured on an Intel E5-2690 (Sandy Bridge) machine with 16 cores and 126 GB of memory.[6] We use Intel®Compilers (icc) for all experiments. We inspect the input against all patterns in the pattern set. Note some applications, such as Web firewall and anti-virus, match the input against all patterns, while other applications, such as IDS, specify the flow group in 5-tuple with the patterns and only the packets that belong to the group are matched against the patterns.

Figure 10 and 11 show performance comparisons for each input data, while varying the number of randomly sampled patterns. We report the average of ten independent runs. The error bars in figure show min and max of throughput. Dotted line shows DFC's performance improvement over AC. [7]

With the random traffic workload, DFC outperforms AC by a factor of 2.6 to 3.6. B2g performs worse than AC even though it does not detect one byte patterns. FFBF also shows worse performance than AC due to short patterns in the ruleset (Table 1) that requires traditional Aho-Corasick based matching. MWM also performs slightly worse than AC. The existence of short patterns contributes to the loss in performance. With the real traffic workload, DFC outperforms AC by a factor of 2.0 to 2.5. DFC's relative improvement slightly decreases as the number of patterns increases. This is because the text generates more matches. The difference between the real traffic and random payload is also due to the same reason. The real traffic contains a relatively larger fraction of string segments in the pattern set because some string patterns contain very generic keywords (e.g., GET). Note the rules that contain these generic patterns also have other options to evaluate (e.g, traffic direction and user-agent type). In a real IDS setting, these packets will be filtered by such options. Our microbenchmark is a conservative one that purely focuses on the string matching. Nevertheless, DFC consistently delivers significant performance improvement in all cases.

Figure 12 shows the number of instructions executed and the instructions per cycle (IPC) of DFC relative to those of AC. We observe that DFC executes 10 to 30% more instructions, but the instruction throughput improves by 130 to 190%; as shown in Table 2, DFC incurs L1 cache misses 3.8 times less frequently and memory accesses 1.8 times less frequently compared with AC.

Malicious traffic and pathological cases: We now evaluate DFC under malicious input traffic by varying the fraction of packets that contain a pattern string from the ruleset. We insert a randomly selected malicious pattern for each packet both for the real traffic workload and random traffic. Figure 13 shows the result with 26K patterns in the ET-Pro ruleset with real traffic. Both algorithms' performance degrades by approximately 30% as we increase the fraction of malicious packets from 0 to 100%. DFC consistently outperforms AC by a factor of 1.7 to 2.0 because AC's performance also degrades due to increased cache miss rates and the additional cost of book-keeping the position of matched input and the PID. Even when all packets contain a malicious pattern, it achieves 1.7 times the performance of AC. We observe a similar behavior

[6]We use two CPUs with 16 cores. The actual memory usage depends on the algorithm (see Figure 15).

[7]We omit ET-Open result because it is similar to ET-Pro's.

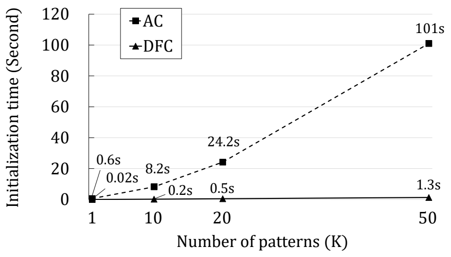


Figure 19: Time to initialize data structures

in the random traffic workload (not shown in the figure) with performance improvement by a factor of 2.2 to 2.6.

To study the behavior of DFC with more adversarial input, we generate a pathological case where the input consists only of two-byte segments used to set the initial DF. We randomly order the two-byte segments to generate the input. This forces DFC to pass through the initial filter on every two bytes of input. Figure 14 shows the performance of AC and DFC by varying the number of patterns. Even in this case, DFC performs significantly better than AC regardless of the number of patterns.

Finally, we examine the two cases where *every byte* of input is malicious and nearly malicious. For the first case, we construct the payload by concatenating randomly selected the strings from the pattern set. For the second case, we concatenate all but the last byte of randomly selected patterns, forcing DFC to trigger the verification phase very often. In these two worst cases, DFC respectively delivers 1.4X and 1.6X the performance of AC at 1 Gbps. The reasons is that, even in those cases, DFC shows 1.6X less L1-D cache misses compared to AC. Note, in the first case, they also frequently write to the memory to note the pattern IDs matched. However, these are very unlikely cases because often an upstream firewall can easily cut the flow based on the 5-tuple information for flows containing many malicious patterns or that try to attack the pattern matching system. Our results show that DFC outperforms AC both in the average and worst case scenarios.

Memory footprint: Figure 15 shows the memory footprint of AC and DFC including the pattern strings. We see that DFC occupies 3.1 to 4.8 times less memory compared to AC. AC requires 239.2 MB for 26K patterns from the ET-Pro ruleset. In contrast, DFC takes up only 59.4 MB memory for the same patterns. The memory footprint of the initial DF is negligible. The total footprint is actually dominated by compact tables that account for 95.0%. Progressive filtering takes up only 0.2%, and the string patterns take up 5.0% with 26k ET-Pro patterns.

Throughput with smaller CPU cache: An application performing multi-pattern matching may share CPU cache with other middlebox applications in some environments, such as network function virtualization (NFV). To evaluate whether the benefits of DFC still remain in such circumstances, we launch an additional thread that consumes half (10MB) of the L3 cache. Even in this case, DFC still outperforms AC by a factor of 1.9 with packet contents from real traffic and ET-Pro ruleset.

Initialization Time: Due to policy change and newly discovered attack, patterns used in middlebox applications are continuously kept updated. For example, new patterns for IDS are typically released each day [15]. To evaluate the overhead for updating data structures, we measured how long it takes to construct data structures for AC and DFC with ET-Pro ruleset.

As depicted in Figure 19, the gap between DFC and AC becomes larger; As the number of pattern increases from 1K to 50K, the speedup improves from 30X to 78X. It is because the initialization time of AC increases super-linearly while that of DFC increases linearly.

5.2 Applications of DFC

Intrusion detection: We apply DFC to an intrusion detection system. In particular, we use Kargus-CPU [39] which is a software IDS that is functionally compatible with Snort and provides multi-threading support and high-performance packet I/O. We measure the end-to-end performance of the IDS using synthetic/real traffic workloads and ET-Pro ruleset except rules that do not contain string fields. Note, this is strongly recommended practice [23]. For synthetic traffic, we use randomly generated the packet payload. For real traffic workload, we replay the packets captured from a commercial cellular ISP as fast as possible to attain a peak transmission rate of up to 70 Gbps. Up to 2 machines are used to generate the traffic. Because the workload consists of real-life flows, the flow management module takes up additional CPU cycles to update per-flow state. The patterns are also grouped based on the 5-tuple flow information. The IDS performs flow reassembly and feeds in the stream for pattern matching. This is representative of how IDSs actual work in a real environment.

Figure 16 shows the performance with Snort VRT ruleset for synthetic traffic by varying packet sizes. For large packets ($\geq$ 512B) DFC improves the performance by a factor of up to 2.6. For 128B packets, the difference is small because packet processing overhead is far greater than that of pattern matching since the amount of payload is only 74B per packet. Figure 17 shows the result while varying the number of patterns from 1K to 20K from the ET-Pro ruleset. DFC shows 120% improvement in performance over AC with 10K patterns. Even though some fraction of the CPU cycles are being spent in flow reassembly, DFC improves performance by 50 to 130%. Note that Kargus-GPU delivers a factor of 1.6 to 2.3 improvement over Kargus-CPU using two GPU cards.

Web application firewall: We apply DFC to an L7 Web firewall, ModSecurity. We benchmark the performance using the OWASP ModSecurity Core Ruleset [14]. Using

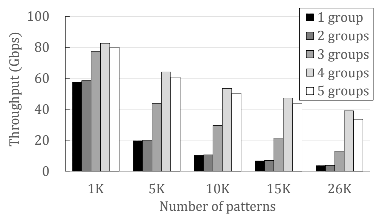


Figure 20: Performance for various number of classification

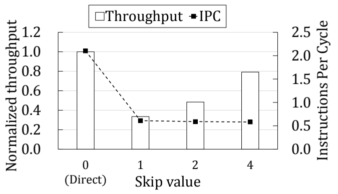


Figure 21: Filtering throughput and instruction throughput (IPC) of DF and "skip" heuristics

	DF	Bloom
Filtering throughput	1	0.42
Instructions/byte	1	2.5

Table 5: Normalized throughput and # of instructions per byte of text of DF and Bloom Filter using two light-weight rolling hashes [53]

ab [2], we request random files of size 10 KB. We choose 10 KB because it is the most frequently found Web object size according to HTTP Archive [11]. We vary the fraction of malicious requests from 0 to 100%. Note, ModSecurity inspects both the request and the response. When a malicious pattern is detected in the request, it generates a 403 forbidden response.

Figure 18 shows the transaction throughput per second. We report the performance result. DFC improves the performance by 90% with innocent request/responses. When the fraction of malicious requests increases, the transaction throughput goes up because response body is not inspected. For 100% malicious requests, pattern matching is only applied to requests, and the performance is dominated by the TCP processing overhead.

We also measure the performance with HTTP request/responses extracted from our real traffic trace where most of the traffic is HTTP. We modify ab and Apache to generate the same request and responses. The result shows that DFC delivers 57% improvement over AC.

Traffic classification: We apply DFC to HTTP traffic classification using nDPI [7], an open source DPI library. nDPI performs flow reassembly to gather fragmented HTTP headers, feeds them in pattern matching module that uses Aho-Corasick, and classifies which application generated the traffic by pattern-matching domain names in the header. We replace AC with DFC. We measure the performance of the traffic classification using the packets captured from a commercial cellular ISP and the top 100K domain names on Alexa [4]. Our result shows that DFC improves the performance by 60% (from 4.2 Gbps to 6.7 Gbps).

Anti-virus: We apply DFC to an anti-virus application, ClamAV [43]. ClamAV uses string and regular expression-like signatures that contain wildcard characters. The AC algorithm processes signatures containing wildcard characters and Wu-Manber (WM) handles fixed string signatures. We replace WM to use DFC and compare against SplitScreen that uses feed-forward Bloom filters [31] to improve ClamAV.

We use 1.5 million ClamAV signatures and scan a clean install of Microsoft Office 2015. For SplitScreen, we run both server and client on the same machine, while using the parameters from the paper [31]. Our result shows that

# of patterns	1K	5K	10K	15K	26K
1B DF	41 Gbps	33 Gbps	28 Gbps	25 Gbps	21 Gbps
2B DF	**76 Gbps**	**55 Gbps**	**45 Gbps**	**37 Gbps**	**28 Gbps**
3B DF	66 Gbps	47 Gbps	37 Gbps	32 Gbps	25 Gbps

Table 6: Performance sensitivity to DF size (ET-Pro ruleset)

# of patterns	1K	50K	1000K	1500K	2000K
1B DF	14 Gbps	6 Gbps	5 Gbps	5 Gbps	5 Gbps
2B DF	**84 Gbps**	21 Gbps	18 Gbps	18 Gbps	18 Gbps
3B DF	53 Gbps	**49 Gbps**	**28 Gbps**	**25 Gbps**	**23 Gbps**

Table 7: Performance sensitivity to DF size (ClamAV ruleset)

DFC brings 75% the performance improvement (from 14.3 MB/s to 25.2 MB/s). When the files are malicious, the gap increases because SplitScreen must perform exact matching with AC using the output of FFBF, which is expensive. We also perform a standalone microbenchmark of FFBF and DFC using 1.5 million string patterns. The result shows that DFC's exact matching outperforms FFBF's inexact matching by a factor of 1.9. We see that DFC accesses memory (measured in terms of L1-data cache loads) 1.3 times less frequently, incurs 1.6 times less L1 cache misses, and uses 2.1X less instructions compared to FFBF.

5.3 Performance Contributions and Parameters

Performance contribution: Initial filtering and progressive filtering are very effective in screening innocent traffic. We measure the fraction of windows that actually reach the verification phase. We use real traffic workload of Table 4 and the ET-Pro ruleset without network I/O. With 26K patterns, 6.2% of windows reach progressive filtering. Progressive filtering further filters out up to 83.9% of windows. As a result, 4% reach the verification phase. A relatively large fraction of windows (8–14%) that arrives at the verification phase generates an exact match.

We evaluate the importance of the 2-stage hierarchy and progressive filtering by measuring the performance of DFC without each of them using real traffic and 100% malicious traffic. For the latter, we insert a malicious pattern for every packet in the real traffic workload.

Without progressive filtering, hash tables are looked up for verification whenever a window passes through the initial DF. In this case, the performance drops by 26% for the real workload and 24% for the malicious workload.

Our 2nd-stage hierarchy is especially beneficial when there are malicious patterns in the input, improving the worst case performance. Without the two-stage hierarchy, the performance drops down by 2.7X in the real traffic workload and 3.9X in the 100% malicious case. This is because finer-grained 2nd-state progressive filtering reduces the false positives in the average case, and reduces the collision and the number of patterns to be inspected in the final hash table in the malicious workload.

We now discuss parameters of DFC that affect its performance in the order of their importance.

Size of initial DF: We evaluate the effect of the size of the initial DF. We compare the performance of DFC by varying the size of the initial DF only. Table 7 and Table 6 show the result by varying the number of patterns for ClamAV and ET-Pro rulesets respectively. For ET-Pro, DF indexed by 2 bytes (2B DF) shows the highest performance across all cases under our workload. For ClamAV, DF indexed by 3 bytes (3B DF) peforms better when the number of patterns exceeds 50K. The difference is that ClamAV patterns are longer, which makes verification relatively more expensive. At the same time, as the number of patterns increases the verification phase becomes more expensive. Because 2B DF triggers verification more often than 3B DF, using 3B DF becomes more beneficial as the number of patterns increases.

Classification granularity: We quantify the effect of classification granularity. With finer-grained classification, the number of patterns inspected in the verification phase decreases at the cost of additional DF lookups in progressive filtering. We vary the number of classes, n, from 1 to 5 and measure the performance of DFC. Given n, we create pattern classes, $P_0, P_1, ..., P_{n-1}$. Figure 20 shows the performance comparison. As the number of patterns increases, the benefit of having a finer-grained classification becomes noticeable. This is because we can filter out the long patterns more effectively, if we can identify the pattern size with finer-grained classification. However, increasing the number of classes provides marginal benefit because the cost also increases. With 5 classes or more, the cost out-weights the benefit.

5.4 Comparison of Pattern Matching Primitives

The benefit of DFC comes from effectively leveraging the simple DF primitive. We benchmark the performance of our direct filter primitive in comparison with three other primitives used in existing algorithms: DFA's state transition, skip table, and hashing. [8]

DFA-based algorithms use state transition tables that cause frequent memory lookup and cache misses. Table 2 shows the number of memory references and cache misses for DF, DFC, and AC. DFC reduces memory accesses frequency by 1.8 times and L1 cache misses by 3.8 times compared to AC.

We compare the performance of DF and the "skip" table used for heuristics-based algorithms [71]. Note the skip approach has sequential data dependency—the next window to examine is dependent upon the value of the skip table entry—whereas DF lookups can be easily be pipelined. To quantify its performance benefit, we use a skip table that is indexed by the same two byte window as DF. Each table entry contains two bits that indicate how many bytes it can skip. We also increase the size of DF to 16 KB by using two bits for each entry. Thus, skip table and DF exhibit the same caching behavior. We set all skip values to be the same, but vary them from 1 to 4 across different experiments. When the skip value is four, the skip table is looked up on every *five* byte of input, whereas DF is looked up on every byte. Figure 21 shows the lookup performance of the two tables for our input. Surprisingly, DF is much faster than the "skip" approach even when the skip table performs five times less lookup (skip=4). DF's instruction throughput, measured in instructions per cycle, is also 3.5X times faster. This coupled with existence of many short patterns make the heuristics ineffective.

Finally, we compare the performance of a direct filter and a Bloom filter lookup that uses two light-weight rolling hash functions used in feed-forward Bloom filter (FFBF) [53].[9] We set the size of both data structures to the same value (8 KB). Table 5 shows that DF is 2.4 times faster than Bloom filter that performs hashing on every window. This is because hashing requires 2.5X more instructions and incurs more frequent memory accesses and cache misses due to additional memory lookups during the calculation of a rolling hash.

6 Related Work

Multi-pattern string matching: There have been a number of multi-string pattern matching algorithms, such as Aho-Corasick [24], Commentz-Walter [33] and Rabin-Karp [42] algorithms. Among these, AC is the most popular in security applications. For example, most software-based IDSs employ AC as the first-pass filter. For the traffic caught in the first phase, they perform perl compatible regular expression (PCRE) matching to confirm an intrusion against more sophisticated attack patterns. Since AC has to deal with all input traffic, its performance often determines the overall performance of an IDS in a normal situation. There have been optimization works that reduces the state transition table size by compressing DFA states [65], that splits AC tables into RAM and CAM [34], or that employs binary transition tables [64]. Some algorithms (e.g., SigMatch [41] and FFBF [53]) perform filtering at the cost of false positives. To support exact

[8]All results are measured on a machine with two Intel Xeon E5-2690 CPUs, using a randomly generated byte stream with 26K string patterns from the ET-Pro ruleset.

[9]The rolling hash we use [53] is known to be 2.5 times faster than a rolling hash used in the Rabin-Karp algorithm [32, 42].

matching, these algorithms typically rely on traditional exact pattern matching.

Many proposals also leverage hardware support, including GPUs [39, 44, 67–69], FPGAs [62, 63], many-core processors [54], new ASIC designs [64], and network processors [49]. Kargus [39] implements a GPU-based Aho-Corasick engine that delivers 39 Gbps for randomly-generated payloads and 2.4K Snort rules with a NVIDIA GTX 580 GPU and an Intel X5680 CPU. Sourdis and Pnevmatikatos [63] present a FPGA-based string pattern matching that reduces the area cost by sharing comparators for different patterns. It achieves 9.7 Gbps of throughput with 1.5K Snort rules on Virtex2-6000. These studies focus on enhancing the performance of existing algorithms on hardware, whereas DFC presents a new algorithm that leverages the performance characteristics of modern CPUs. We believe that DFC can also benefit from hardware implementations and leave it as future work.

Regular expression matching: Many studies [26–28, 45, 46, 49, 51, 52, 56, 59] have proposed new NFA/DFA-based algorithms for regular expression matching. These studies predominantly use FPGA or ASIC to take advantage of the high degree of parallelism and on-chip memory for high performance [45]. The overarching goal is to create a single automaton that matches multiple regular expressions (regex). The core challenge is that a single automaton is fast, but space-inefficient because combining multiple regex into a single DFA causes state explosion [61]. In contrast, NFA-based solutions are slow, but require small amount of memory. Thus, many efforts [26, 45, 50] focus on reducing the memory footprint of DFA to utilize the fast on-chip memory as much as possible. XFA [60, 61] presents a design that tries to achieve the best of both worlds. It uses much less memory than using a single DFA and is faster than using multiple DFAs. But, the performance is still much slower (at least 6 times) than using a single DFA. Note, DFA-based regular expression matching is much slower than string matching. Due to the distinct trade off in performance and expressiveness, both string and regular expression matching are commonly used for deep packet inspection.

Advances in software-based packet processing: Advances in packet processing technologies [37, 38, 40, 57] are accelerating cloud-based middleboxes [58] and network function virtualization, which call for efficient primitives for fast deep packet inspection. Recently, G-opt [40] showed that CPU can deliver most of the benefits of GPU by proposing an element switching technique and hiding the memory latency. DFC accelerates pattern matching by avoiding dependencies between instructions and reducing memory access frequency, whereas G-opt proposes a generic technique for memory latency hiding. The two approaches are thus orthogonal. Our preliminary evaluation shows that DFC outperforms G-opt's hand-optimized pattern matching by 42 to 71% for Snort VRT ruleset, depending on the number of pattern groups to match. However, we believe applying memory latency hiding can further enhance DFC's performance (e.g., G-opt's efficient hash table lookup can be applied to DFC). We leave this as future work.

7 Conclusion

Middlebox services that inspect packet payloads have become commonplace. With the popularization of cloud-based DPI services, their performance and cost-efficiency have become important. However, for many DPI applications, pattern matching poses a severe performance bottleneck. This paper addresses an important problem of scaling the performance of multi-pattern string matching. Using cache-efficient data structures and progressive filtering based on the pattern lengths, we develop a high-performance string matching algorithm that works well across a variety of pattern sets and workloads. DFC classifies patterns based on the their size and applies filters progressively in multiple stages. Our evaluation shows that DFC improves state-of-the-art by 100 to 150% with a popular IDS pattern set and real traffic traces. We demonstrate that replacing existing pattern matching module with DFC results in a 57-160% performance improvement in four different applications.

Acknowledgement

We would like to thank our shepherd Paolo Costa and anonymous reviewers for their feedback and suggestions and Jaehyun Nam, Hyeontaek Lim and Iulian Moraru for their comments on initial design. We also thank Huy Hoang for his help on testing. This work was supported by IITP granted by the Korea government (MSIP) [B0126-15-1078, Creation of PEP based on automatic protocol behavior analysis and Resource management for hyper connected for IoT Services], [B0101-15-1368, Development of an NFV-inspired networked switch and an operating system for multi-middlebox services], and by NRF-2013R1A1A1076024.

References

[1] A developer's guide to complying with PCI DSS 3.0 Requirement 6. http://www.ibm.com/developerworks/library/se-pcireq6/index.html. [accessed 01-Sep-2015].

[2] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.2/en/programs/ab.html. [accessed 01-Sep-2015].

[3] Akamai Cloud Security Solutions: Kona Web Application Firewall. https://www.akamai.com/us/en/multimedia/documents/product-brief/kona-web-application-firewall-product-brief.pdf. [accessed 01-Sep-2015].

[4] Alexa - Actionable Analytics for the Web. http://www.alexa.com.

[5] Can the WAF help with a DDoS attack? https://support.cloudflare.com/hc/en-us/articles/200172116-Can-

the-WAF-help-with-a-DDoS-attack-. [accessed 29-Aug-2015].

[6] Capacity Planning for Snort IDS: Bilbous, Not Tapered. http://mikelococo.com/2011/08/snort-capacity-planning/. [accessed 01-Sep-2015].

[7] Configuring nDPI for custom protocol detection. http://www.ntop.org/ndpi/configuring-ndpi-for-custom-protocol-detection/. [accessed 01-Sep-2015].

[8] Emerging Threats Ruleset. http://rules.emergingthreats.net/.

[9] How The Great Firewall Works. http://cs.stanford.edu/people/eroberts/cs201/projects/international-freedom-of-info/china_2.html. [accessed 09-Sep-2015].

[10] How the NSA's domestic spying program works. https://www.eff.org/nsa-spying/how-it-works. [accessed 09-Sep-2015].

[11] HTTP Archive. http://httparchive.org.

[12] Intel Performance Counter Monitor - A better way to measure CPU utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor.

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

[14] OWASP ModSecurity Core Rule Set. http://spiderlabs.github.io/owasp-modsecurity-crs.

[15] Proofpoint ET Pro - Ruleset The Expert Solution. http://www.emergingthreats.net/products/etpro-ruleset.

[16] Secure, fast, and easy Web Application Firewall. https://www.cloudflare.com/waf. [accessed 09-Sep-2015].

[17] Shorewall: iptables made simple. http://shorewall.net/.

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

[19] Snort Ruleset 2.9.7.0 from Snort Downloads. https://www.snort.org/downloads/. [accessed 01-May-2015].

[20] Snort User Manual 2.9.7. http://manual.snort.org/.

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

[22] The Great Firewall of China: Keywords Used to Filter Web Content. http://www.washingtonpost.com/wp-dyn/content/article/2006/02/18/AR2006021800554.html. [accessed 09-Sep-2015].

[23] Writing Good Rules: Content Matching. http://manual.snort.org/node36.html#SECTION00491000000000000000. [accessed 01-Sep-2015].

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

[25] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. Generating Realistic Workloads for Network Intrusion Detection Systems. *ACM Special Interest Group on Software Engineering (SIGSOFT) Software Engineering Notes (SEN)*, 29(1), January 2004.

[26] 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.

[27] Michela Becchi and Patrick Crowley. Extending Finite Automata to Efficiently Match Perl-compatible Regular Expressions. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2008.

[28] Michela Becchi and Patrick 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):4:1–4:26, April 2013.

[29] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM (CACM)*, 13(7):422–426, July 1970.

[30] R.S. Boyer and J.S. Moore. A Fast String Searching Algorithm. *Communications of the ACM (CACM)*, 20(10):762–772, October 1977.

[31] S. K. Cha, I. Moraru, J. Jang, J. Truelovea, D. G. Andersen, and D. Brumley. SplitScreen: Enabling Efficient, Distributed Malware Detection. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[32] J. D. Cohen. Recursive Hashing Functions for N-grams. *ACM Transactions on Information Systems (TOIS)*, 15:291–320, July 1997.

[33] Beate Commentz-Walter. A String Matching Algorithm Fast on the Average. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, London, UK, UK, 1979.

[34] Vassilis Dimopoulos, Ioannis Papaefstathiou, and Dionisios Pnevmatikatos. A Memory-efficient Reconfigurable Aho-Corasick FSM Implementation for Intrusion Detection Systems. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2007.

[35] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational Experiences with High-volume Network Intrusion Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004.

[36] Branislav Durian, Jan Holub, Hannu Peltola, and Jorma Tarhio. Tuning BNDM with Q-grams. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2009.

[37] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-Accelerated Software Router. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[38] Intel. Data Plane Development Kit (DPDK). http://dpdk.org.

[39] 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)*, 2012.

[40] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[41] Ramakrishnan Kandhan, Nikhil Teletia, and Jignesh M Patel. SigMatch: Fast and Scalable Multi-pattern Matching. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 3(1-2):1173–1184, 2010.

[42] Richard M Karp and Michael O Rabin. Efficient Randomized Pattern-matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

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

[44] Lazaros Koromilas, Giorgos Vasiliadis, Ioannis Manousakis, and Sotiris Ioannidis. Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures. In *Proceedings of the ACM/IEEE symposium on Architectures for Networking and Communications Systems (ANCS)*, 2014.

[45] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced Algorithms for Fast and Scalable Deep Packet Inspection. In *Proceedings of the ACM/IEEE symposium on Architectures for Networking and Communications Systems (ANCS)*, 2006.

[46] 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 (SAC)*, 2007.

[47] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf. [accessed 01-Sep-2015].

[48] Po-Ching Lin, Zhi-Xiang Li, Ying-Dar Lin, Yuan-Cheng Lai, and F.C. Lin. Profiling and Accelerating String Matching Algorithms in Three Network Content Security Applications. *IEEE Communications Surveys and Tutorials*, 8(2):24–37, 2006.

[49] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen, and Chia-Nan Kao. A Fast String-matching Algorithm for Network Processor-based Intrusion Detection System. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):614–633, 2004.

[50] Chad R Meiners, Jignesh Patel, Eric Norige, Alex X Liu, and Eric Torng. Fast Regular Expression Matching using Small TCAM. *IEEE/ACM Transactions on Networking (TON)*, 22(1):94–109, 2014.

[51] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast Regular Expression Matching using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *Proceedings of the USENIX Conference on Security*, 2010.

[52] 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.

[53] Iulian Moraru and David G Andersen. Exact Pattern Matching with Feed-Forward Bloom Filters. *Journal of Experimental Algorithmics (JEA)*, 17:3–4, 2012.

[54] Jaehyun Nam, Muhammad Jamshed, Byungkwon Choi, Dongsu Han, and KyoungSoo Park. Haetae: Scaling the Performance of Network Intrusion Detection with Many-Core Processors. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[55] Marc Norton. Optimizing Pattern Matching for Intrusion Detection. *Sourcefire, Inc., Columbia, MD*, 2004.

[56] Jignesh Patel, Alex X Liu, and Eric Torng. Bypassing Space Explosion in High-speed Regular Expression Matching. *IEEE/ACM Transactions on Networking (TON)*, 22(6):1701–1714, 2014.

[57] Luigi Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[58] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2012.

[59] Reetinder Sidhu and Viktor K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[60] Randy Smith, Cristian Estan, and Somesh Jha. XFA: Faster Signature Matching with Extended Automata. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2008.

[61] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM)*, 2008.

[62] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. *Field Programmable Logic and Application (FPL)*, pages 880–889, 2003.

[63] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for Efficient and High-speed NIDS Pattern Matching. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[64] Lin Tan and Timothy Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.

[65] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2004.

[66] Chris Ueland. Scaling cloudflare's massive waf. http://www.scalescale.com/scaling-cloudflares-massive-waf/. [accessed 09-Sep-2015].

[67] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2008.

[68] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P Markatos, and Sotiris Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2009.

[69] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: A Multi-parallel Intrusion Detection Architecture. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[70] Shinae Woo, Eunyoung Jeong, Shinjo Park, Jongmin Lee, Sunghwan Ihm, and KyoungSoo Park. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proceeding of the ACM International Conference on Mobile systems, applications, and services (MobiSys)*, 2013.

[71] Sun Wu and Udi Manber. Fast Text Searching: Allowing Errors. *Communications of the ACM (CACM)*, 35(10):83–91, October 1992.