DFC: Accelerating String Pattern Matching for Network Applications

Byungkwon Choi, Jongwook Chae, Muhammad Jamshed, KyoungSoo Park, Dongsu Han
Trend: Popularity of Network Function Virtualization (NFV)

- NFV: Commodity hardware appliances → Software layer
  - Virtualizes entire class of network functions
  - E.g., IDS, Firewall, NAT, Load balancer, ...

SDN, NFV & network virtualization market will grow at CAGR of 37% by 2020 according to market forecasts

Vodafone uses Affirmed Networks NFV Solutions to Deliver M2M Services

Cisco’s new NFVI solution to speed up network services
Pattern Matching for Deep Packet Inspection

- Looking for known patterns in packet payloads
  - String pattern matching (Fixed-length string) and Regex matching (PCRE)
  - 5K ~ 26K rules in public rule-sets for network applications

- Rule Examples
  - Rule 1: Content: “Object” PCRE: “/(ActiveX|Create)Object/i”
  - Rule 2: Content: “Persits.XUpload” PCRE: “\s*\([\x22\x27]Persits.XUpload/\i”
  - Rule 3: Content: “FieldListCtrl” PCRE: “ACCWIZ\xeFieldListCtrl\xe1\xe8/i”

String pattern matching     Regular expression matching
Pattern Matching for Deep Packet Inspection

• Looking for known patterns in packet payloads
  – String pattern matching (Fixed-length string) and Regex matching (PCRE)
  – 5K ~ 26K rules in public rule-sets for network applications

• Network applications using pattern matching
Pattern Matching for Deep Packet Inspection

• Looking for known patterns in packet payloads
  - String pattern matching (Fixed-length string) and Regex matching (PCRE)
  - 5K ~ 26K rules in public rule-sets for network applications

• Network applications using pattern matching

- Attack patterns
- Intrusion Detection
- Banned words
- Parental Filtering

- Attack patterns
- Web Application Firewall
- Watermark
- Exfiltration Detection
However, String Pattern Matching is Performance Bottleneck

70-80% of CPU cycles consumed by string pattern matching *

However, String Pattern Matching is Performance Bottleneck

70-80% of CPU cycles consumed by string pattern matching *

Can we improve software-based string matching?
How does it affect application performance?

DFC: High-Speed String Matching

1) Outperforms state-of-the-art algorithm by a factor of up to 2.4
2) Improves network applications performance

- **Intrusion Detection**
  - Existing-approach-based: 12.8 Gbps
  - DFC-based: 29.6 Gbps
  - **130% ↑**

- **Web Application Firewall**
  - Existing-approach-based: 4,155 req./s
  - DFC-based: 6,537 req./s
  - **60% ↑**

- **Traffic Classification**
  - Existing-approach-based: 4.2 Gbps
  - DFC-based: 6.7 Gbps
  - **60% ↑**
Three Requirements of String Matching

• Support **exact matching**
  – As opposed to false positives

• Handle **short and variable size patterns** efficiently
  – 52% of patterns are short (< 9 byte).

• Provide efficient online lookup **against a stream of data** (e.g., network traffic)

---

* Commercial pattern sets of IDS & Web Firewall (ET-Pro, Snort VRT, OWASP ModSecurity CRS)
Limitations of Existing Approaches

• Aho-Corasick (AC)
  – Widely used by Suricata, Snort, CloudFlare, ...
  – Constructs a finite state machine from patterns
  – Locates all occurrences of any patterns using the state machine

* Example

Patterns:
- HIS
- HERS
- HE
- SHE

Input text: FINISHED

Result: SHE HE
Limitations of Existing Approaches

- **Aho-Corasick (AC)**
  - Widely used by Suricata, Snort, CloudFlare, ...
  - Constructs a finite state machine from patterns.
  - Locates all occurrences of any pattern.

- **Limitations of AC**
  - State machine is very large.
  - Working set $\gg$ CPU cache size.
  - Instruction throughput is slow.

![Graph showing memory footprint vs. number of patterns](image)
Limitations of Existing Approaches (Cont.)

• Heuristic-based approach (Boyer-Moore, Wu-Manber, ...)
  – Advances window by multiple characters using “bad character” and “good suffix”
  – Not effective with short and variable size patterns
  – Hard to leverage instruction-level pipelining

• Hashing-based approach (Feed-forward Bloom filters (FFBF), ...)
  – Compares hash of text block with hash of pattern
  – Requires expensive hash computations (2.5X more instructions than DFC)
  – Not effective with short and variable size patterns
  – Induces false positives
DFC: Design Goal

• Overcomes the limitations of existing approaches
  – Consumes small memory
  – Works efficiently with short and variable size patterns
  – Delivers high instruction-level parallelism

• Works efficiently even in worst case
  – Worst case where all packets contain attack patterns
DFC: Overview

• Exploits a simple and efficient primitive
  – Used as a key building block of DFC
  – Requires small number of operations and memory lookups
  – Filters out innocent windows of input text

• Progressively eliminates false positives
  – Handles each pattern in a different way in terms of pattern length

• Verifies exact matching
  – Exploits hash tables
DFC: Component Overview

- **Initial Filtering**
  - Uses an efficient primitive “Direct filter”
  - Eliminates innocent windows of input text comparing few bytes (2~3 byte)

- **Progressive Filtering**
  - Eliminates innocent windows further
  - Determines lengths of patterns that window might match
  - Applies additional filtering proportional to the lengths

- **Verification**
  - Verifies whether exact match is generated
DFC: Initial Filtering

• Uses a single Direct filter
  – A bitmap indexed by several bytes of input text
  – Example (Using 2B sliding window)

Packet Payload: GET /destroy/attack/try-20

Direct filter → 01100100 01100101

Example pattern:
- attack
- athlete
- author

```plaintext
01100100 01100101
```

```plaintext
GET /destroy/attack/try-20
```

```
0 0 0 0 1 1 0 0
```

```
dc  dd  de  as  at  au
```
DFC: Initial Filtering

- Uses a single Direct filter
  - A bitmap indexed by several bytes of input text
  - Example (Using 2B sliding window)

Packet Payload:

```
GET /destroy/attack/try-20
```

Example pattern:

- attack
- athlete
- author

No pattern beginning with ‘de’
DFC: Initial Filtering

- Uses a single Direct filter
  - A bitmap indexed by several bytes of input text
  - Example (Using 2B sliding window)

Packet Payload:

```
GET /destroy/attack/try-20
```

Direct filter →

```
... 0 0 0 ... 0 1 1 ... 0 0 0 ...
```

Example pattern:

```
attack
athlete
author
```

Further inspection
DFC: Initial Filtering

- Uses a single Direct filter

1) No data dependency (Instruction parallelism ↑)

2) 2 SHIFTs and 1 AND + 1 memory reference

3) 2 byte → $2^{16}$ = 65536 = 8KB

Example pattern:

- attack
- athlete
- author

94% of windows are filtered out.
DFC: Progressive Filtering

- Further eliminates innocent windows

Packet Payload: `GET /destroy/attack/try-20`

Additional filtering
DFC: Verification

• Exact matching: \((100 - 94\%) \times (100 - \text{up to 84\%}) = \text{only 4\%!}\)

Packet Payload: GET /destroy/\textcolor{red}{\textit{attack}}/try-20

Hash (‘atta’) 4~7B

\[ \begin{align*}
1 & \rightarrow \text{atta} \quad \text{ck} \quad \text{Pattern ID} \\
0 & \quad \text{Pattern ID} \\
2 & \rightarrow \text{athl trasf} \quad \text{ete fic} \quad \text{Pattern ID}
\end{align*} \]
DFC: Two-Stage Hierarchical Design

1st Stage

- Initial Filtering
- Progressive Filtering
- Verification

2nd Stage

- Progressive Filtering
- Verification

Pattern Set

- .asp
- .asp?
- .asp?a=
- .asp?p=
- .asp?u=
- .asp<k
- .asp<k?

* Found from ET-Pro
Evaluation

• Two questions
  1) Can we improve software-based string matching?
  2) How does it affect application performance?

• Machine Specification & Workload
  – Intel Xeon E5-2690 (16 cores, 20MB for L3 cache)
  – 128 GB of RAM
  – Intel® Compilers (icc)
  – Using real traffic trace from ISP in south Korea
Standalone Benchmark (1/2) – Average Case

Throughput (Gbps)

Improvement over AC

Improvement

Heuristic-based (MWM)*
Aho-Corasick (AC)
DFC
Improvement

0 0.5 1 1.5 2 2.5
1K 5K 10K 15K 26K

2.4 2.2 2.1 2.1 2.0

Number of patterns
(From ET-Pro, May 2015)

1K 5K 10K 15K 26K

24

* MWM: Modified Wu-Manber
**Standalone Benchmark (2/2) – Worst Case**

- **Worst case 1 (Single pattern)**
  
  ... innocent ATTACK innocent ...

- **Worst case 2 (Concatenated)**
  
  ... ATTACK1 ATTACK2 ATTACK3 ...

**Graphs:**

- **Throughput (Gbps):**
  - AC: 70% increase
  - DFC

- **Fraction of malicious packets:**
  - 0%
  - 50%
  - 100%

**AC: 62X increased size of working set**

**Remarks:**

- Packet size: 1514B

---

25
Why does DFC work well?

Factor increase with DFC over AC

<table>
<thead>
<tr>
<th>Instruction Count</th>
<th>IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2</td>
<td>2.3</td>
</tr>
</tbody>
</table>

# of cache misses per one byte processing

<table>
<thead>
<tr>
<th>Cache</th>
<th>AC</th>
<th>DFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-D cache</td>
<td>1.07</td>
<td>0.28</td>
</tr>
<tr>
<td>L2 cache</td>
<td>0.19</td>
<td>0.04</td>
</tr>
</tbody>
</table>

3.8X ↓

4.8X ↓
Accelerating Network Applications using DFC

- Intrusion Detection (Kargus - CCS `12, 6K)
  - Normalized throughput: 12.8 Gbps
  - AC-based: 4,155 req./s
  - DFC-based: 6,537 req./s (60% ↑)

- Web Application Firewall (ModSecurity, 5K)
  - Normalized throughput: 29.6 Gbps
  - AC-based: 231 Gbps
  - DFC-based: 29.6 Gbps (130% ↑)

- Traffic Classification (from nDPI, 100K)
  - Normalized throughput: 4.2 Gbps
  - AC-based: 60% ↑
  - DFC-based: 6.7 Gbps (60% ↑)

Large # of patterns
**DFC: High-Speed String Pattern Matching**

- String pattern matching is a performance-critical task.
- DFC accelerates string pattern matching by
  - Using small size of basic building block
  - Avoiding data dependency in critical path

- DFC delivers **2.4X speedup** compared to Aho-Corasick.
  - 1.4X in the worst case

- DFC improves application performance by up to **130%**.

- Detailed information at [ina.kaist.ac.kr/~dfc](ina.kaist.ac.kr/~dfc)