Leveraging Traffic Repetitions for High-Speed Deep Packet Inspection

Yotam Harchol
The Hebrew University of Jerusalem, Israel

Joint work with:

Anat Bremler-Barr
IDC Herzliya, Israel

Shimrit Tzur-David
IDC Herzliya, Israel

David Hay
Hebrew University, Israel

This research was supported by the European Research Council, the Israeli Centers of Research Excellence (I-CORE) program, and the Neptune Consortium.
Deep Packet Inspection (DPI)

- Popular process in many middleboxes:
  - Intrusion Detection and Prevention
  - Content Filtering
  - Leakage Prevention
  - More...

- Search for a set of patterns in packets payload

  user=alice&pass=1234
  user=bob&pass=5678
  user=carl&pass=abcd
  user=evil&pass=1234'; DROP TABLE users
  user=delta&pass=efgh
  user=eric&pass=ijkl
  user=fox&pass=6543

Pattern can be:
Exact string: "DROP TABLE"
Regular expression: ";/s*DROP/s*TABLE"
• Many repetitions in sent/received traffic
• Partial or whole files
  – Similar web pages, HTML tags, scripts (javascript), request headers, ...

• **Middleboxes / VNFs** keep scanning these repetitions over and over
Our Contribution

• An algorithm that **identifies** repetitions and **accelerates**
  *deep packet inspection*
  
  – **25%-150% speedup** in software
  
  – **250% potential speedup** on hardware
DPI Engine – Complicated Challenge

• Hundreds of academic papers over recent years

  - scalability
  - throughput
  - latency
  - power
  - resiliency
  - updates
  - compression

• Pattern set size varies between $10^2$-$10^5$ patterns DPI

• Engine is considered a **system bottleneck** in many of today's MBs *(30%-80%)*

  [Laboratory simulations over real deployments of Snort and ClamAV]
Background: Aho Corasick Algorithm

- DFA for multi-string matching
  - Also used for pre-filtering for regular expression matching

- One memory access per input character

- Huge data structure
  - In Snort IDS: ~80MB, ClamAV: ~1.8GB

- Performance penalty for jumping between distant memory locations

- Our algorithm skips Aho-Corasick steps to boost performance
System Overview

• An algorithm that *identifies* repetitions and *accelerates* 
  
  *deep packet inspection*

  – **Offline**: building a dictionary of repeating strings

  – **Online**: identifying repeating strings and skipping them
System Overview

Dictionary of Repeating Strings

<table>
<thead>
<tr>
<th>String</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;!DOCTYPE html&gt;&lt;html&gt;&lt;head&gt;&lt;meta http-equiv=&quot;Content-Type&quot; content=&quot;text/html; charset=UTF-8&quot;&gt;</code></td>
<td>732</td>
</tr>
<tr>
<td><code>title&gt;Gmail&lt;/title&gt;</code></td>
<td>23141</td>
</tr>
<tr>
<td><code>http-equiv=&quot;Content-Type&quot; content=&quot;text/html; charset=UTF-8&quot;&gt;</code></td>
<td>1564</td>
</tr>
<tr>
<td><code>t</code></td>
<td>912</td>
</tr>
<tr>
<td><code>arset=UTF-8&quot;&gt;&lt;ti</code></td>
<td>10238</td>
</tr>
<tr>
<td><code>&gt;title&gt;Gmail&lt;/title&gt;</code></td>
<td>42390</td>
</tr>
</tbody>
</table>

Identify repetitions

Offline Slow Path

Online Data Path

Restore state

Without re-scanning!

Skip characters

Improve Throughput
Repetitions in HTTP Traffic

If we had an infinite dictionary we could skip over 80% of HTML traffic...

Which is about 20% of the overall HTTP traffic
Our Algorithm: Slow Path

Scan recently inspected traffic

Heavy Hitters Algorithm [Mek et al.: ANCS 2013]

Fixed length grams are faster to identify (online) than variable-length grams

Store the last DFA state after scanning each gram

Fixed length k-grams

Run DPI (Aho-Corasick) on each gram

Runs periodically (every couple of hours, or even days) - so performance penalty is marginal!

<table>
<thead>
<tr>
<th>k-gram</th>
<th>jmpState</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>732</td>
</tr>
<tr>
<td></td>
<td>23141</td>
</tr>
<tr>
<td></td>
<td>1564</td>
</tr>
<tr>
<td></td>
<td>912</td>
</tr>
<tr>
<td></td>
<td>10238</td>
</tr>
<tr>
<td></td>
<td>42390</td>
</tr>
</tbody>
</table>

New Dictionary
The Dictionary

- Dictionary per content type
  - As HTML strings will not appear in javascript files
- Dictionary per source/destination host (e.g., only for traffic to/from youtube.com)

- Size of dictionary is a tradeoff:
  - Larger: leverage more repetitions, longer lookup, higher cache miss rate
  - Smaller: faster dictionary lookup, smaller potential skips → lower throughput gain

Luckily, the size is easily configurable

Typical gram lengths ($k$) are 16 and 32 bytes (configurable)
Algorithm Outline
Avoid missing matches when skipping

**Pattern set:**
1. `<img`
2. `img src=`
3. `=virus>`

**Dictionary:**

<table>
<thead>
<tr>
<th>k-gram</th>
<th><code>img src</code></th>
<th><code>=virus&gt;</code></th>
</tr>
</thead>
</table>

**Input:** `<img`

*Left Margin Resolution:*
Continue DFA traversal to make sure no match is skipped

Specifically, let \( j \) be the index in gram, then continue traversal until \( j \geq \text{depth}(\text{state}) \)

\( \text{state}=s_0 \)

\( j=0 < \text{depth}(s_1) \)

Cannot jump to \( s_{17} \):
Will miss the match on \( s_{10} \)!
Algorithm Outline

**Pattern set:**
1. `<img ✓`
2. `img src= ✓`
3. `=virus> ✓`

**Dictionary:**

<table>
<thead>
<tr>
<th>k-gram</th>
<th>jmpState</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>img src</code></td>
<td><code>s_{17}</code></td>
</tr>
<tr>
<td><code>=virus&gt;</code></td>
<td><code>s_{25}</code></td>
</tr>
</tbody>
</table>

**Input:** `<img src=virus>`

- Match!
- Found in dictionary!
- `state=s_{17}`
- `j=0 < depth(s_{18})`
- Can not jump
- Skip 3 characters to `s_{23}`
- Continue DFA traversal

**Diagram:***

- DFA transitions with forward and cross transitions shown.
- Depths 0 and 1 not shown.
Efficient Dictionary Lookup

Gram in dictionary ➔ Performance gain
Gram not in dictionary ➔ Performance pain

• Dictionary lookup may take time:
  Hash computation, String comparison, Possible hash collisions

• Problem: Reduce number of redundant hash table lookups

• Solution: Bloom filter before dictionary lookup

<table>
<thead>
<tr>
<th>k-gram</th>
<th>Hash functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based on crc32q hardware instruction</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bloom Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 1 0 1 1 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hash Table (~45K entries)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>k-gram</th>
<th>jmpState</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Goal: Find repetition level for which skips beat regular Aho-Corasick

\[ p \times \frac{(BF + DICT + c \times AC)}{k} + (1 - p) (BF + fpr \times DICT + AC) < AC \]

Minimal \( p \) for speedup:

\[ p_{\text{min}} = \frac{k (BF + DICT)}{(k - 1)BF + (k - fpr - 1) \times DICT + (k - c)AC} \]

\( \approx 60\% \)

On HTML traffic

In our experiments

<table>
<thead>
<tr>
<th>Traffic</th>
<th>( p_{\text{actual}} )</th>
<th>Predicted Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>youtube.com</td>
<td>79%</td>
<td>66%</td>
</tr>
<tr>
<td>General HTML</td>
<td>47%</td>
<td>-14%</td>
</tr>
</tbody>
</table>
Analysis – Hardware Implementation

With hardware solution we can do better:
Simultaneously access dictionary and Aho-Corasick DFA

Expected byte processing time: $\text{Expected byte processing time: (Hardware)}$

Bytes in dictionary (left-margin or skipped) Bytes not in dictionary

$$p \times \frac{\max(BF + \text{DICT}, c \times AC)}{kk} + (1 - p) \max(BF + \text{DICT}, AC) + ACC$$

<table>
<thead>
<tr>
<th>Traffic</th>
<th>Predicted Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>youtube.com</td>
<td>256%</td>
</tr>
<tr>
<td>General HTML</td>
<td>145%</td>
</tr>
</tbody>
</table>

$p_{\text{min}} = 0\%$
For all traffic types, with any reasonable parameter values
Experimental Results – Software
On Snort IDS pattern matching engine

Dictionary update every 6 hours
Experimental Results – Software
On Snort IDS pattern matching engine

Dictionary update every 6 hours

\[
p_{\text{actual}} \quad \text{Predicted Speedup} \quad \text{Actual Speedup}
\]
\[
79\% \quad 66\% \quad 53\%
\]
Experimental Results – Software
On Snort IDS pattern matching engine

Local News Website (ynet.co.il)

Time [Hours]

Speedup [%]

<table>
<thead>
<tr>
<th>$p_{\text{actual}}$</th>
<th>Predicted Speedup</th>
<th>Actual Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>87%</td>
<td>110%</td>
<td>132%</td>
</tr>
</tbody>
</table>

Dictionary update every 6 hours
• Even when not updating dictionary for *days*, algorithm still provides high speedup.
Related Work

• **Hardware solutions:**
  
  *Require specialized hardware such as FPGA or TCAM*
  
  [Baker and Prassana, 2004], [Clark et al., 2004], [Lee et al., 2007], [Meiners et al., 2010], [Dharmapurikar and Lockwood, 2006], [Pao et al., 2010], [Chen et al., 2013]

• **Software solutions:**
  
  *Can be used on top of our proposed architecture*
  
  [Kumar et al., 2006], [Ficara et al., 2008], [Schuff et al., 2007], etc.

• Shenoy et al., 2012 presented a limited algorithm that also leverages repetitions using a basic algorithm that provides lower throughput
  
  – Only skips large chunks of 256 and more bytes, and only from root state
Conclusions

In many points in the network data is **highly repetitive**

Middleboxes keep scanning the data **over and over**

This work presents an algorithm that **accelerates DPI** over repeated data by **25%-150%** in software

(In general, accelerates traffic with >60% skip potential)

On parallel hardware speedup may get to **250%** and more
Thank you!

Questions?
Regular Expressions Matching

• Common approach (e.g. Snort): string matching with a global DFA → single regex DFA

\(<\!\!\!\text{DOCTYPE}\!\!\!\text{s+[^>]*SYSTEM[^>]*}.\!\!\!\text{parseError}\>\>

\(<\!\!\!\text{DOCTYPE} \text{SYSTEM} \!\!\!\text{parseError}\>\>

• Repetition operators (e.g., .*) may cause memory blowout in DFA
• NFA’s performance degrades due to multiple simultaneous active states
Algorithm

**Input:**

```
<html><head><meta http-equiv="Content-Type" ...>
```

- pos = 0
- state = root

**Dictionary**

**Aho-Corasick DFA**

**Init:** j = 0

- depth(state) > j?
  - Yes
    - Scan one byte
    - state = jmpState
    - pos = pos + (k - j)
  - No
    - j++

- Update state
- pos++

**Miss**
Input:

```html
<HTML><HEAD><META HTTP-EQUIV="Content-Type"...
```

pos = 0
state = root

Dictionary

Can replace Aho-Corasick with other string matching algorithms!
(with some adaptations)
Algorithm Outline

**Pattern set:**
1. `<img`
2. `img src=`
3. `=virus>`

**Dictionary:**

<table>
<thead>
<tr>
<th>k-gram</th>
<th>jmpState</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>img src</code></td>
<td><code>s_17</code></td>
</tr>
<tr>
<td><code>=virus&gt;</code></td>
<td><code>s_25</code></td>
</tr>
</tbody>
</table>

**Input:** `<img src=virus>`

---

**Matches:**

- `<img`
- `img src=`
- `=virus>`

---

```python
pos = 0
state = root
while pos < length:
    gram = input[pos : pos + k]
    if gram is in dictionary:
        j = 0
        while depth(state) > j:
            state = AC(input[pos], state)
            pos++
            j++
        state = dictionary[gram].jmpState
        if state has a match, handle it
        pos = pos + k – j
    else:
        state = AC(input[pos], state)
        pos++
```