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

Xiang Wang¹, Yang Hong¹, Harry Chang¹, KyoungSoo Park², Geoff Langdale³, Jiayu Hu¹ and Heqing Zhu¹

1 Intel Corporation; 2 KAIST; 3 branchfree.org
Networking Applications with Regex Matching

- Deep packet inspection (DPI) – key functionality of L7 traffic monitoring
- Regular expression (regex) matching – core element of DPI
- Big problem – regex matching is **SLOW**
Current Best Practice: Prefilter-based Pattern Matching

Rule 0:
content: "SEARCH";
pcre: "\sSEARCH\s\w+\s{\d+}\{\r\}?\n[^n]*?%/smi"

Rule N: ...
content: "UNSUBSCRIBE";
pcre: "\w+\s+UNSUBSCRIBE\s[^n]{100}/smi"

Two-stage Pattern Matching

Multi-string matching

SEARCH

UNSUBSCRIBE

Single regex matching

Match!

\sSEARCH\s\w+\s{\d+}\{\r\}?\n[^n]*?%/smi

Single regex matching

No Match

/^\w+\s+UNSUBSCRIBE\s[^n]{100}/smi
Problems with Prefilter-based Pattern Matching

Manual choice of improper string keywords

content: "\[\]/;
pcre: '\[defghilmnoqrstuvwxyz\]?=[defghilmnoqrstuvwxyz](m(ookflofcnmo\x2fnmot\.fmu|clvomycem|x2fcen\vcn)"

Duplicate matching of the string keywords

Complex regexes lead to slow NFA

Pattern

.*foo[^x]barY+

Input

XfoZbarY

String Matching for “bar”

X f o Z b a r

Regex Matching

X f o Z b a r Y

Slow
Contributions

Issues

Problems with current best practices
- Manual choice of improper string keywords
- Duplicate matching of the string keywords
- Complex regexes lead to slow NFA

Suboptimal matching performance
- Slow multi-string matching
- Slow NFA matching

Solutions

Novel regex decomposition

SIMD-based pattern matching
- Efficient multi-string matching
- Fast bit-based NFA

Outcome

- Snort: 8.7x Speedup
- Multi-string matching: 3.2x Speedup over DFC
- Multi-regex matching: 13.5x Speedup over RE2
Wide Adoption of Hyperscan

• Successfully deployed by over 40 commercial projects globally
• In production use by tens of thousands of cloud servers in data centers
• Integrated into 37 open-source projects
Regex Decomposition
Decomposition-based Matching

**Pattern**

.*foo[^x]barY+

**Input**

XfoZbarY

- Decomposes a pattern into string (STR) and subregex (FA) components
- String matching is the entrance
- All components have to be matched in order

**String Matching**

X f o Z b a r Y

**FA Matching**

- No duplicate string keyword matching
- Smaller FAs with fast DFA matching
- Facilitate multi-regex matching

**FAs**

- FA0
- FA1
- FA2
- STR0
- STR1
- STR2

- FA1 is Dead!
  - Don't trigger FA0
Key Issues with Regex Decomposition

• How to automatically decompose a regex?

• How many real-world regexes can be decomposed?
Key Issues with Regex Decomposition

• How to automatically decompose a regex?

• How many real-world regexes can be decomposed?
Graph-based Regex Decomposition

- Textual regex decomposition is often tricky, e.g. /b[il1]\s{0,10}/
- Graph structure delivers more insights

Graph-based Decomposition
1) Dominant Path Analysis
2) Dominant Region Analysis
3) Network Flow Analysis
Graph-based String Extraction

Dominant Path Analysis

Dominant Region Analysis
Graph-based String Extraction

Network Flow Analysis
- Finds a string (or multiple strings) that ends at the edge
- Assigns a score inversely proportional to the length of the string(s) ending at the edge
- Runs “max-flow min-cut” algorithm \([1]\) to find a minimum cut-set

Key Issues with Regex Decomposition

• How to automatically decompose a regex?

• How many real-world regexes can be decomposed?
Effectiveness of Graph Analysis on Real-world Rules

<table>
<thead>
<tr>
<th>Ruleset</th>
<th>Total</th>
<th>All Graph Analyses</th>
<th>Dominant Path</th>
<th>Dominant Region</th>
<th>Network Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snort Talos (May 2015)</td>
<td>1663</td>
<td>94.0%</td>
<td>93.3%</td>
<td>1.9%</td>
<td>1.0%</td>
</tr>
<tr>
<td>Snort ET-open 2.9.0</td>
<td>7564</td>
<td>89.3%</td>
<td>86.9%</td>
<td>1.3%</td>
<td>2.7%</td>
</tr>
<tr>
<td>Suricata 4.0.4</td>
<td>7430</td>
<td>87.5%</td>
<td>85.0%</td>
<td>1.3%</td>
<td>2.7%</td>
</tr>
</tbody>
</table>

Majority of Regex Rules are Decomposable

Dominant Path Analysis is Effective
Quality of Automatically Extracted Keywords

**Snort Talos***

* Left vertical axis: # of regex matching process invocations (In logarithmic scale based on 10)

* Right vertical axis: reduction of Hyperscan

**Snort ET-Open***
SIMD-based Pattern Matching
How to Accelerate Pattern Matching Algorithms?

- Modern CPUs support SIMD (Single Instructions Multiple Data) to exploit data level parallelism
- SIMD instructions can boost database pattern matching by 2x [1]
- Accelerates both multi-string and FA matching with SIMD as the goal

Multi-string Pattern Matching Overview

- Extended shift-or matching
  - Finds candidate input strings that are likely to match some string patterns
- Verification
  - Filters false positives with hashing
  - Confirms an exact match with string patterns with the same hash value
Shift-or String Matching

Limitations:
- Single string pattern matching only
- Cannot benefit from SIMD instructions

Multi-string Shift-or-Matching

- Pattern grouping: Groups the patterns into N buckets
- SIMD acceleration: Uses 128-bit sh-masks with 128-bit SIMD instructions (e.g., pslldq for "left shift" and por for "or")

![Diagram of multi-string shift-or-matching with buckets and sh-masks]
Multi-string Shift-or Matching

Pre-shifting the sh-masks increases instructions per cycle (IPC)!

Match! (bucket = 0, position = 3)  Match! (bucket = 4, position = 3)

128-bit SIMD operations increase throughput!
Bit-based NFA Matching

- Uses DFA as much as possible – but often impossible
- Classic NFA is slow - $O(m)$ memory lookups per input character ($m =$ # of current states)
- Represents each state with one bit in a state bit-vector
- Exploits parallel bit operations of SIMD to compute the next states
Other Subsystems

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small string-set (&lt;80) matching</td>
</tr>
<tr>
<td>NFA and DFA cyclic state acceleration</td>
</tr>
<tr>
<td>Small-size DFA matching</td>
</tr>
<tr>
<td>Anchored pattern matching</td>
</tr>
<tr>
<td>Suppression of futile FA matching</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Evaluation
Evaluation of Hyperscan

• Primary evaluation points:
  1. Performance of **string matching** and **regex matching** vs. state-of-the-art solutions
  2. **Application-level** performance improvement with Hyperscan

• Experiment setup:
  – Machine: Intel Xeon Platinum 8180 CPU @ 2.50GHz (48 GB of RAM)
    ❖ Runs with a single core
    ❖ GCC 5.4
  – Ruleset: Snort Talos (May 2015), Snort ET-Open 2.9.0, Suricata rulesets 4.0.4
  – Workload: random traffic, real-world web traffic
Multi-String Matching Performance with Snort ET-Open

1 Random workload.
2 Real web traffic trace.
Regex Matching Performance

* Test with Snort Talos (1,300 regexes) and ET-Open (2,800 regexes) rulesets under real Web traffic trace.
Real-world DPI Application - Snort

- Stock Snort (ST-Snort) employs
  - AC for multi-string matching
  - PCRE for regex matching
  - Boyer-Moore algorithm single-string matcher
- Hyperscan-ported Snort (HS-Snort) replaced all the algorithms with Hyperscan
- Snort Talos (May 2015) with real-world web traffic

![Snort Performance Chart]

*Legend: ST-Snort, HS-Snort*
Conclusion

• Regex matching is at the core of DPI applications
• Hyperscan’s performance advantage is boosted by:
  − Novel regex decomposition
  − Efficient multi-string matching and bit-based NFA implementation
• Hyperscan achieves significant performance boosts
  − 3.2x compared to DFC in multi-string matching
  − 13.5x compared to RE2 in regex matching
• Hyperscan accelerates DPI application Snort by 8.37x
Thank You

• Thanks Matt Barr, Alex Coyte and Justin Viiret for their development contribution
• Source code at https://github.com/intel/hyperscan