Software-based Acceleration of Deep Packet Inspection on Intel Architecture

OISF-Suricata User Conference
November 5, 2015
Barcelona, Spain

Geoff Langdale
Principal Engineer
Intel Corporation
Legal Disclaimers

General Disclaimer:

© Copyright 2015 Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Inside, the Intel Inside logo, Intel. Experience What’s Inside are trademarks of Intel. Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.

Technology Disclaimer:

Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at [intel.com].

Performance Disclaimers:

Results are provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.
“Some people, when confronted with a problem, think ‘I know, I'll use regular expressions.’ Now they have two problems.”

– Jamie Zawinski, 1997
Welcome and Goals

- Welcome to the OISF Keynote!
- Goal: present Intel’s Networking vision
  - Big vision is SDN/NFV
  - Medium-sized vision: Hyperscan, Open Source, Suricata!

- Who are you and why are you telling us about your vision?
  - Former CTO of Sensory Networks
  - Principal Engineer at Intel
  - Chief Architect of the Hyperscan software-based pattern matcher

- Hyperscan is a software library
  - Aim to be best-of-breed software regular expression matching for DPI
  - Will present Suricata integration
History

Sensory Networks (2003-2013)

- **Hardware**-based PCI-X cards 2003-2006 for Pattern Matching
  - “grep chip” in FPGA
  - Hired me in 2006 to work on new GPGPU project
  - Team size: 70

- **Vaporware**, prototypes, GPGPU, 2007-2008
  - We made a lot of slideware
  - Also, regex ‘bits and pieces’
    - Bit-parallel NFA
    - Literal matchers
History

Sensory Networks had a corporate Near Death Experience in Jan 2009

- **Reorganized with Sab Gosal as CEO, me as CTO**
- **Team size: 5**

**Software** – “Hyperscan” 2009-2013

- Hyperscan: “Software based pattern matching library”
  - No more hardware/GPGPU
  - Just algorithms…
  - Version 2.0 in 2009
  - … up to 2.1.22 in late 2013

Acquired by Intel in October 2013
Transformation

From This, Today...
Traditional networking topology
Monolithic vertical integrated box
TEM proprietary solutions

ADC*  Network Security  Router

To This, the Vision
Networking within VMs
Standard x86 server hardware
Open SDN standard solutions

ADC VM  Network Security  Router

SDN + NFV

Hypervisor (e.g. ESXi, KVM, Hyper-V, etc.)

TEM/OEM  ASIC, DSP, FPGA, ASSP
Proprietary OS

Processor  NIC  Acceleration  Switch  Open Software  TPM
More than a Decade of Investment

4:1 Workload Convergence on IA

Application Processing
- Other Architectures
- Intel Architecture
- NPU/ASIC
- DSP

Control Processing
- Other Architectures
- Intel Architecture
- NPU/ASIC
- DSP

Packet Processing
- NPU/ASIC
- NPU/ASIC
- NPU/ASIC
- DSP

Signal Processing
- DSP
- DSP
- DSP
- DSP

High-Speed I/O
- Intel® Ethernet

Key:
- Non-Intel
- Intel with Networking Investment
- Acquisition

Intel® QuickAssist Technology, Comms Chipset, SOCs
Hyperscan
Data Plane Development Kit
NPU/ASIC
IA
IA
Intel® Media Server Studio
Mindspeed, Axxia Assets

Intel Architecture
Multi-core, Memory & I/O Integration

2014+
2008-2014
Mid-2000s
Early 2000s
Legacy

One Tool Suite
One Instruction Set Architecture
Rich Ecosystem
Hyperscan

Hyperscan is a software-based library for regex and literal matching

- **libpcre** is the syntax: semantics slightly different
- **Multiple regular expressions**
  - We’ve had commercial deployments with 20K+ regular expressions
  - Tested 2M regular expressions in the lab (once…)
- **Streaming** (aka ‘cross packet inspection’)
- **High performance** (and scales well on multicore)
- **Low overheads** (stream state, bytecode size, compile time)

Hyperscan 2.0 -> 3.4: closed source library, in commercial use at a range of Tier 1 IPS/IDS and NGFW vendors
Hyperscan 4.0 and open source announcement

Hyperscan 4.0 is now open source software
Under 3-Clause BSD license as of October 19, 2015
Get it at 01.org/hyperscan
Why Open Source Hyperscan?

Open source now an industry expectation

Make a software solution that works without special drivers/hardware

- “Virtualization ready”

Easy to get into an evaluation
Why Open Source Hyperscan?

Enable *everyone* not just a few hand-picked customers

- Get mindshare with engineers and organic success in the market
- Bottom-up tech driven not top-down:

*“Win in the lab, not on the golf course”*
Hyperscan 4.0 Contents

Hyperscan library

- Compiler and run-time modules
- Examples
- Simple tools and bench markers
- Documentation

Platforms

- Linux: Many modern variants
- FreeBSD
- Windows
- MacOSX
  - Requires modern C++ toolchain

- Does not include games console
Hyperscan Open Source Strategy

Permissive license

- Use with GPL/BSD licensed software or in a proprietary closed-source system
- We welcome and encourage contributors from outside Intel

We’re not done!

- Active 10+ member team in Intel; 4.1 out soon
- We will provide patches to OSS projects ourselves
  - First wave: 2 projects - Suricata and Snort
  - Roadmap/discussed: ClamAV, Bro, L7Filter, SpamAssassin, YARA, ...
- We encourage independent development of Hyperscan integrations
Next release: Hyperscan 4.1

- 3 month cycle (+/- one month)
- “Normal” development
  - Optimizations for performance, stream size, bytecode size
  - New optimizations for Intel ISA extensions
- Tools and testing
  - Benchmarking software
  - “Fuzzing software”: automatically generate “interesting” ...
    - … test data for regex
    - … regular expressions
Hyperscan Operation

How does this thing work, anyhow?

- Answer: it’s complicated
- A hybrid between lots of different algorithms
  - NFA
  - DFA
  - Literal matching
  - Duct (duck?) tape
- We will cover a few key parts here…
Hyperscan Operation

- Literals vs regex:
  - Literals are easy to scan for, regex is hard(er)
  - Find a literal in a regex and scan for that instead: “literal factor”
- Rule writers: you may be familiar with this process
  - You may also get it wrong!
  - To quote ANTLR’s Terence Parr:

  “Why Program by Hand in Five Days what You Can Spend Five Years of Your Life Automating”

  - Good news: we have already spent those 5 years, so you don’t have to!
- We have code for this
  - Graph algorithms!
Literal Match Extraction

```bash
#alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any (msg:"ET ACTIVEX Remote Desktop Connection ActiveX Control Heap Overflow clsid access"; flow:established,to_client; file_data; content:"7390f3d8-0439-4c05-91e3-cf5cb290c3d0"; nocase; distance:0; pcre:="/<object[^>]*\s*classid\s*=\s*(.+<q1>\x22|\x27)>\s*\x3a\s*{\s*\?s*7390f3d8-0439-4c05-91e3-cf5cb290c3d0\s*}?\s*\?P=q1}\s*\>/si"; reference:cve,2009-1929; reference:url,www.microsoft.com/technet/security/Bulletin/MS09-044.mspx; reference:url,doc.emergingthreats.net/2009907; classtype:attempted-user; sid:2009907; rev:10;)
```

**Note highlighted content string was manually extracted!**

Hyperscan does this for free, but doesn’t find anything better in this case

- That’s still easier and less error-prone

*Example rule from ETOpen Set, emerging-activex.rules, stillsecure is rule author*
Literal Match Extraction

content:"url=" [... ]
pcre:"/url=\s*(ftps?|https?|php):\//\//Ui" [...]

In this case, Hyperscan gets more power:

- Conjunction of "url=" ...

- and any one of the set: { "ftp:/", "tps:/", "http:/", "php:/" }
  - *Surprise:* not { "ftp:/", "ftps:/", "http:/", "https:/", "php:/" }

- Is this a dumb idea? Now we’re looking for 5 literals!
  - Generally not: we need to look for lots of literals anyhow
    - More of a choice between 501 and 506 literals…
  - We become better insulated from ‘bad choices’

  e.g. ‘I predict that “http://www.” will happen very infrequently because it’s a 11-character string’
Literal Match Extraction

Graph algorithms to find best “cut”
Can find conjunctions of disjunctions (effectively)
Under some circumstances we go to extreme lengths to get some sort of literal

- Following loops
- Expanding our character classes
- Use when no plausible literals are easily available
- There are limits (e.g. we don’t blow out /a·b/s into 256 literals)
Hyperscan Operation: Decomposition

The Trouble With Factors:
When they are present in the data, you have to do the work anyway

Solution: “Rose” engines – decomposition of the regex task

- \(<R1>\text{teakettle}<R2>\text{badgerbrush}<R3>/s\) for arbitrary regexes \(<R1>\), \(<R2>\), \(<R3>\)
  - \(<R1>\) is a “prefix”: we only need to know its state when we see a “teakettle” string
  - \(<R2>\) is an “infix”: we never need to run it at all unless we see a “teakettle” at the right point to match R1, and we only need to know its state when we see “badgerbrush”
  - \(<R3>\) is a “suffix”: we never need to run it at all unless we see a “badgerbrush” at the right point to match R2, but it can throw matches anywhere

- Breaking up the task allows many optimizations
Hyperscan Operation: Engines

Literal Matchers: the first line of defense

- Large scale literal matcher (100-50K strings) – “bucketed super-character shift-or”
- Small scale literal matcher (2-100 strings) – “fast SIMD search using shuffle”
- Single string literal matcher

These are the main engines behind the performance described in the next section

Pattern match engines

- NFA and DFA engines
- SIMD implementation of “acceleration”

Specialized stuff: small write optimizations, large bounded repeats
Suricata Integration

Our Suricata+Hyperscan patch will patch several places in scanning:

- **Multiple literal match (done) - results presented here**
- **Single literal match – Work in Progress (WiP)**
- **Single regular expression match (WiP)**

Patch based on original work at Wind River:

- Now under direct Intel development
- Plan is to release by EOY 2015
Suricata Integration: Multiple Literal Match

- Large scale literal match a ‘staple’ of Hyperscan
  - We use this for bulk regex matching
  - We promote this capability directly

- Just literals in this pass?
  - No: we do additional constructs like `/^.{10,50}xyz/s`
    “Anchored literals”
    Avoid match ‘floods’
    Hyperscan decides whether to optimize or treat as bare literal and check later
    
    `/^.{10}a/s vs /^.{10,1400}teakettle/s`

    Big saving to not throw a match on every ‘a’ … but the anchored match of teakettle stinks
Suricata Integration: Single Literal Match

*Note: WiP (not for original patch; well understood from other systems)*

We match single literals pretty fast

- Tuned SIMD match on Intel – >100+Gbps typical
- Grab a couple characters, PCMPEQB and some shift/and

Use Hyperscan opportunistically (not very profound)
Suricata Integration: Single Regex

**Note:** WiP (not for original patch; well understood from other systems)

Simulate behavior of backtracking regular expressions adequately

- Raise the right match in the right place!
- Implement all regular expressions (somehow)

Not a big performance improvement, but great risk management

At Sensory Networks, we found the ‘sergeant attack’ on an IDS with libpcre:

“RCPT TO:<<<<<<<<<“ against an ambiguous regex that backtracked across the “<“ string

Every “<“ character halved performance…

… until the whole system failed open after a timeout
Suricata Integration: Single Regex (Prefiltering)

Standard escape hatch: Prefiltering

- **HS_MODE_PREFILTER**: replace unsupported constructs with weaker substitutes
  
  Example: `\foo (\d)+bar\1baz\` -> `\foo (\d)+bar (\d)+baz`
  
  Now accepts foo123bar456baz (false positive)

- Allows false positives but not false negatives

Same weakness as “literal factors”: If we match, we must call libpcre to confirm (ouch)

- When can Hyperscan do better than just “prefiltering”?  
  - When we can support the regex constructs 100%  
  - When we can **know** that we got the right match  
    - **Static**: Fixed width patterns (done), Unambiguous (WiP)  
    - **Dynamic**: When there is only one match!
Suricata Integration – Performance results

Machine under test: cubit, an Intel® Xeon® CPU E5-2680 v2 @ 2.80 GHz

Software versions:

- Hyperscan 4.0.0 (first public release as available on github)
- Suricata 2.1 beta 4
- Experimental integration of Hyperscan into Suricata (coming soon as !) – available upon request

The following disclaimer applies to the results presented in the next slides:

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit http://www.intel.com/performance.
Suricata Integration – Performance results

Rule set: ET-Open rules snapshot on 2015-10-26
Input PCAPs: two downloads of Alexa Top 100

- “alexa” – manual download of non-adult site Alexa Top 100 landing pages
- “new-alexa” – automated script (no compressed/encrypted data included in trace)

Suricata run as:

```
suricata -c <config> -r <pcap file> --runmode=single
```

- Modification to run repeats to stabilizes data (yes, this has flaws)
- Measurement is “all processing time” (not just scan)
  - But it’s not true end-to-end (not a network measurement)
Suricata Integration – Performance results

Multiple literal match only

- Preliminary benchmarking showed no real time spent on our workloads in regular expressions or single literal match
- Not true of all workloads, inputs
- **Call to action:** If you’ve got rules/inputs with slow regexes, let us know!

Literal matcher memory usage results still preliminary

- Hyperscan is the same size or smaller than Aho-Corasick default
- Medium/Single or High/Single policies about the same
- Medium/Full or High/Full show huge difference (process size or MPM’s own instrumentation) – at least 5x, maybe more (~50x ?!?)
- **Deduplication** is a big deal for bytecode size (merge identical literal sets)
  - More nuanced version of this built for other systems; still a WiP.
## Hyperscan vs default Aho-Corasick

*Rule set without –event rule files*

**Best Hyperscan vs Best AC: Alexa 1.90X, new-alexa 2.28X**

Measured in Packets per second for Default AC, Hyperscan, Ratio is HS/AC

<table>
<thead>
<tr>
<th>AC Matcher</th>
<th>PCAP</th>
<th>Medium/Single</th>
<th>Medium/Full</th>
<th>High/Single</th>
<th>High/Full</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Default AC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>alexa</td>
<td>43337</td>
<td>43356</td>
<td>46250</td>
<td>46777</td>
<td></td>
</tr>
<tr>
<td>new-alexa</td>
<td>55202</td>
<td>52445</td>
<td>58999</td>
<td>51681</td>
<td></td>
</tr>
<tr>
<td><strong>Hyperscan</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>alexa</td>
<td>73823</td>
<td>88710</td>
<td>79293</td>
<td>87006</td>
<td></td>
</tr>
<tr>
<td>new-alexa</td>
<td>104466</td>
<td>120167</td>
<td>102212</td>
<td>134601</td>
<td></td>
</tr>
<tr>
<td><strong>Ratio</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>alexa</td>
<td>1.70</td>
<td>2.05</td>
<td>1.71</td>
<td>1.86</td>
<td></td>
</tr>
<tr>
<td>new-alexa</td>
<td>1.89</td>
<td>2.29</td>
<td>1.73</td>
<td>2.60</td>
<td></td>
</tr>
</tbody>
</table>
Note: Other Literal Matchers Don’t Change Things
Performance Summary

You can integrate a free (as in speech, and as in beer) library into Suricata …

- … and get \(~1.5 - 2x\) performance improvements from a simple (MPM only) integration
- … and use higher performance modes without a blowout in bytecode size
- … and get a substantial upside for future integrations involving regex and single literal match search
- Integration still primitive – like using a hammer to drive in screws…
Our Vision: All Pattern Matching In Software

- Software is flexible: handle a new matching demand in days not years
  - Keep optimizing the hard stuff! Hardware makes easy things easier. “Hard stuff” winds up dominating the workload

- Software is adjacent to the workloads
  - Immediately prior to the regex scan: Reassembly, normalization, etc.
  - Immediate following regex match: More rules, confirmation, etc.

Example: this is a VISA card: `/\^4[0-9]{12}(?:[0-9]{3})?$/` Run Luhn’s algorithm as quick as possible (don’t flood false matches back across a bus)

- Software is agile: stop matching right away if we’ve “seen enough”

- Software is simple to integrate: no drivers, no OS arbitration, runs bare metal or under virtualization
Our Vision: Pattern Matching Beats Ad Hoc Rules

• “Regex is slow” philosophy
• Content strings separated by distance/within/offset type operators
  • Rewrite as regex
• Not all rule constructs are amenable: byte_jump
• Gets rid of interpreter overhead
• Put all the stuff in one basket (as long as it’s a good basket)
• … and it leads to the next slide
Our Vision: Push Regex Earlier in the Process

Multiple pattern matching beats just multiple literal matching

- **Let us make the choices about which factor literals are good!**
  - Currently – patch implements anchored matching for fixed-depth strings
  - We could push a lot more regex logic earlier

- We can implement much of what’s currently done in the Big Snort Rule Interpreter Code
Our Vision: Use Streaming

- Hyperscan allows streaming regular expression: our semantics make these three cases identical
- No limits in terms of detection window (# of writes, distance in bytes)
- Using streaming is a big transformation in the Suricata model
  - Flowbits and rescanning window an ugly workaround to not having streaming
  - We don’t know how to do this conversion! **CALL TO ACTION**…

```
xxxxabcxxxxxxxxxxdefxx
```
```
xxxxab  cxxxxxxxx  xdefxx
```
```
x  x  x  a  b  c  x  x  x  x  x  x  x  x  x  x  x  x  d  e  f  x  x  x
```

Time (earlier writes to later writes)
Conclusions

Hyperscan gets 1.5-2.0x speedup on a simplistic integration

Call to action: try Hyperscan!

Could get a much bigger speedup if problem is transformed

- **Bad news:** Tier 1 commercial IPS has *already done this transformation*
  - We built a viable startup on multi-match regex and streaming (mostly) so your commercial competition got started on this up to 6 years ago…

Call to action: design based on new thinking rather than old assumptions:

- “regex is slow” -> “regex is fast”
- “regex is one pattern at a time” -> “regex is as many patterns as you want”
- “regex doesn’t stream” -> “regex streams efficiently”

Questions?
Additional Links

Me

geoff.langdale@intel.com

Hyperscan Open Source Software Project:

https://01.org/hyperscan

Github Code Repository:

https://github.com/01org/hyperscan

Hyperscan Intel.com Landing Page:
