ACCH & Snort

Adir Gabai | January 2015

Project Goal

Implementing ACCH inside Snort

Agenda

- Deep Packet Inspection
- ACCH Algorithm
- Introduction to Snort
- Solution Details
- ACCH Analysis
- Results

Deep Packet Inspection

DPI Overview

- Inspecting the payload of a packet.
- Packet is filterred based on pre-defined policy.
- Main uses:
  - Intrusion detection / prevention
  - Identifying various malware and attacks
  - Leakage prevention
  - Protocol classification
  - Network Monitoring

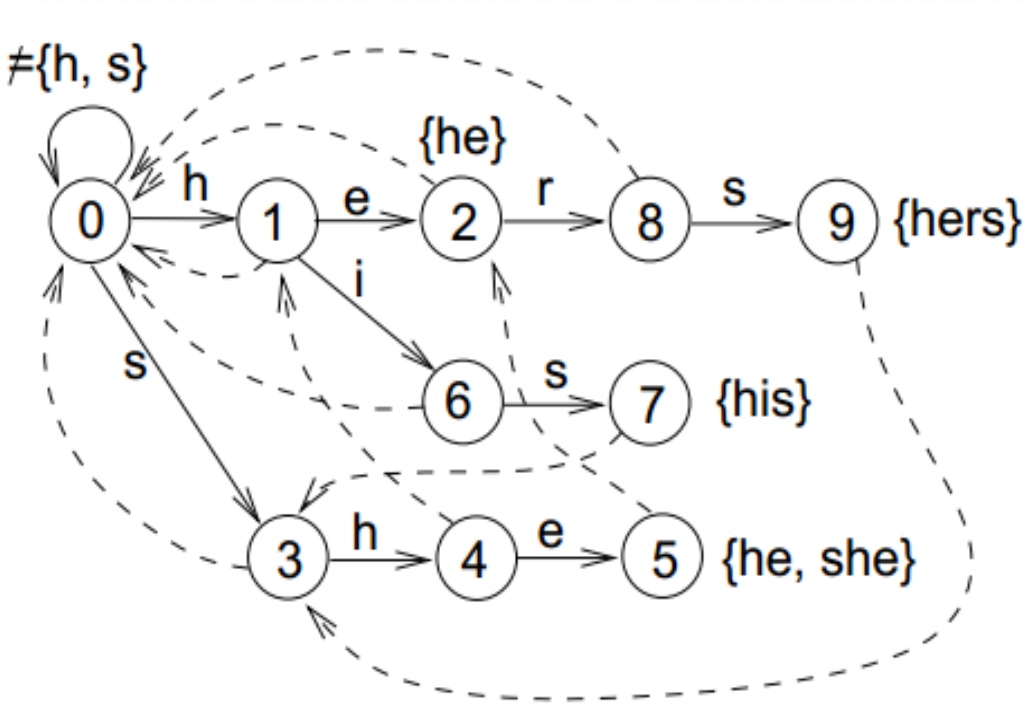
DPI Overview (Cont.)

- Payload can be inspected using:
  - String matching methods
  - Regex matching methods
- ACCH is a **string matching** algorithm.

Aho-Corasick

- Multi pattern matching algorithm.
- Invented by Alfred V. Aho and Margaret J. Corasick.
- Constructs finite state machine (FSM).
- O(n) search time, where n = text size.

Aho-Corasick Example



Patterns
he
hers
his
she

Compressed Http

- Compresses textual data sent from server to client.
  => better bandwidth utilization
- Most common compression algorithms:
  gzip, deflate

Compressed Http Request

```
GET /encrypted-area HTTP/1.1
Host: www.example.com
Accept-Encoding: gzip, deflate
```

Compressed Http Response

```
HTTP/1.1 200 OK
Date: mon, 28 Jul 2014 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Accept-Ranges: bytes
Content-Length: 438
Connection: close
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
```

Gzip

- Based on the DEFLATE algorithm.
- Two phases:
  - LZ77 compression:
    Reduce string presentation size
  - Huffman coding:
    Reduce symbol coding size

LZ77 Compression

- Replaces repeated sequences of bytes with pointers to the original sequeance within the last 32KB.
- Maintains circular buffer often called **sliding window**.

Uncompressed	a	b	b	c	d	b	b	c	e	f	g	e	a	c	c	f	g	e	a
Compressed	a	b	b	c	d	{4,3}			e	f	g	e	a	c	c	{6,4}			
	literals					pointer			literals							pointer			

DPI on Compressed Http

- How to search compressed http for patterns?
- Naïve approach:
  - Uncompress the traffic.
  - Scan the uncompressed data.
- Can we improve?

ACCH

ACCH Approach

- **A**ho-**C**orasick based algorithm for **C**ompressed **H**ttp.
- Main Idea:
  Accelerate the scan of repeated sequence of bytes, that were already scanned for patterns.
- Key Observation:
  LZ77 pointers point to an already scanned bytes.

Pointer Scan

- Can we skip a pointer?
  - Almost.

- Left Boundary: A prefix of the referred bytes may be a suffix of a pattern.
  - The pattern can be detected by continuing the scan up to a certain point.

Pointer Scan (Cont.)

- Right Boundary: A suffix of the referred bytes may be a prefix of a pattern.
  - The pattern can be detected by starting a scan from a certain point.

- All bytes between the left and right boundaries can be skipped!

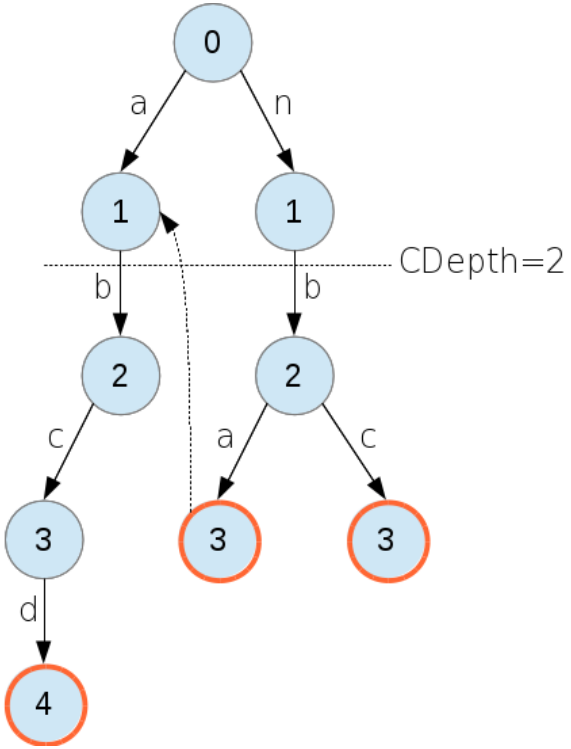
Pointer Scan (Cont.)

- In order to skip bytes and start scanning at the right boundary, we need **additional info** about the states of the referred bytes.

Status	Meaning
Uncheck	Depth <CDepth
Check	Depth>=CDepth => Suspicious!
Match	Match state at the DFA

=> We must maintain a circular buffer of the statuses of the last 32KB.

ACCH: Example



Patterns
abcd
nba
nbc

Traffic	e	b	c	e	c	d	c	e	n	{8,8}								b	a
Uncompressed	e	b	c	e	c	d	c	e	n	b	c	e	c	d	c	e	n	b	a
Depth	0	0	0	0	0	0	0	0	1	2	3	0	?	?	?	?	1	2	3
Status	u	u	u	u	u	u	u	u	u	c	m	u	u	u	u	u	u	c	m


SNORT®

Introduction to Snort

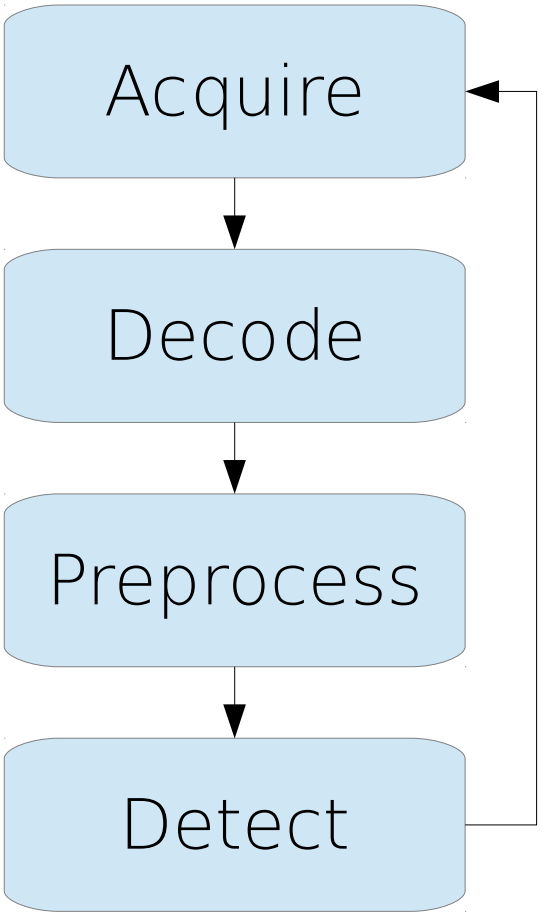
- Free and open source NIDS and NIPS.
- Now developed by Sourcefire.
- Supports various protocols.
- Main string matching algorithm: Aho-Corasic.

Snort Rules

- The suspicious signatures are defined in a special form called “Rule”.
- Example:
  The following rule searches for pattern “foo” in the Http response body.

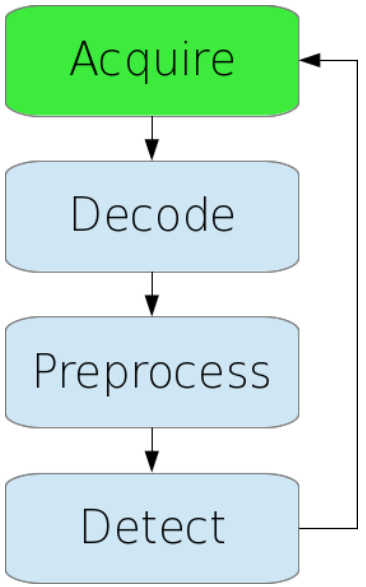
```
alert tcp any 80 -> any any \
(msg:"ALERT!"; file_data; content:"foo";)
```

Main Packet Loop



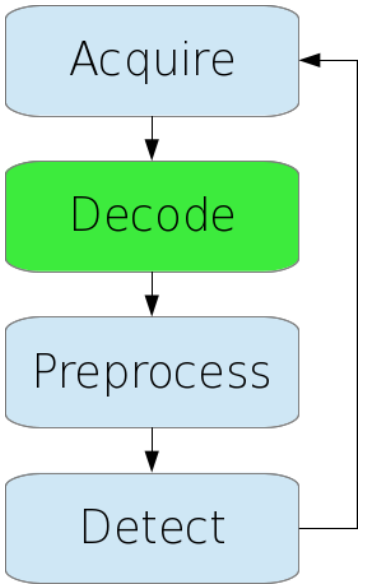
Acquire Packet

- Acquires packet from either network interface or pcap file.
- Packets are acquired using DAQ – Data Acquisition library.
  - Introduces general interface for reading packet from a veriety of software and hw interfaces.



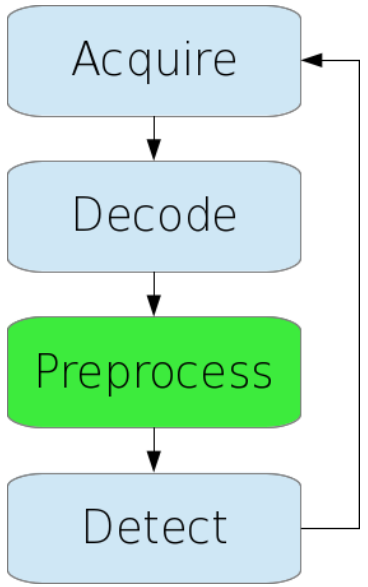
Decode Packet

- Decodes the packet according to the data link type (i.e. Ethernet, IEEE 802.11, etc).
- Extracts all packet fields into convenient structure.



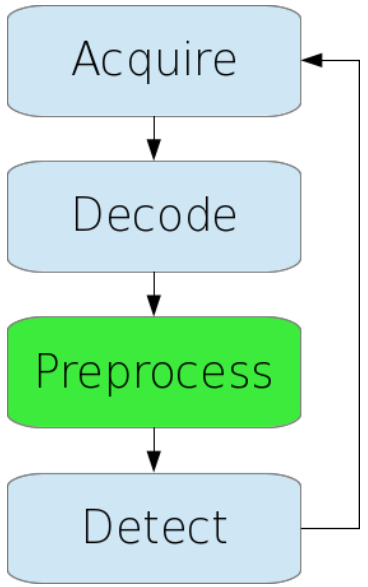
Preprocess Packet

- Prepares the packet for pattern matching.
- May include manipulation on the data such as **reassembly**, **normalization** and **decompression**.
- Built-in preprocessors:
  - Stream5
  - HttpInspect
  - And many more.



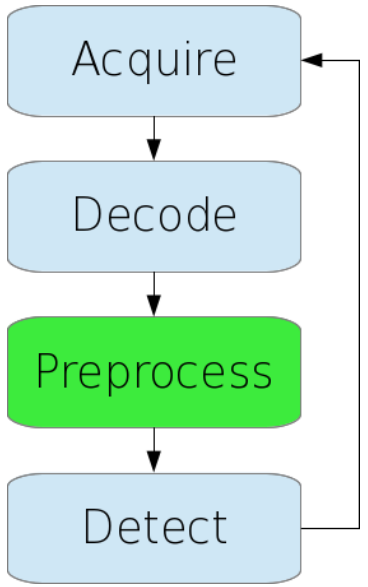
Stream5

- Used to track TCP sessions.
- Can also handle TCP anomalies.
- Supports Stream API
  - Other preprocessor can attach private data to sessions (to be used during the session).
  - Other preprocessors can configure reassembly behaviour.



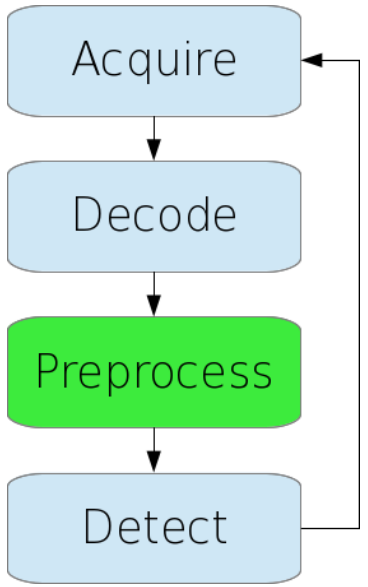
PAF

- **P**rotocol **A**ware **F**lushing
- TCP knows nothing about how the data of the application is structured.
- PAF allows the application preprocessors to determine how to reassmeble the accumulated data.
- N Packets → 1 Buffer.



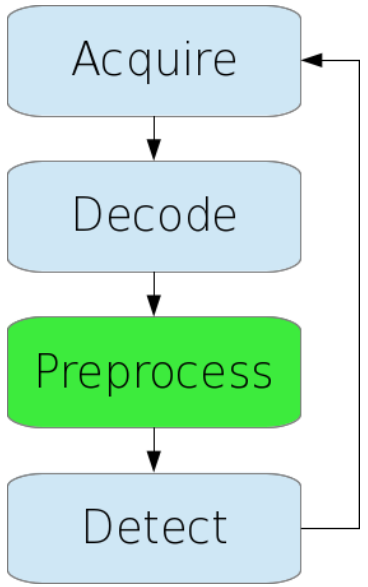
HttpInspect Preprocessor

- Generic Http preprocessor
- Preprocesses both client requests and server responses.
- Given a http payload HttpInspect will:
  - Decode the payload
  - Extract Http fields
  - Normalize Http fields & payload



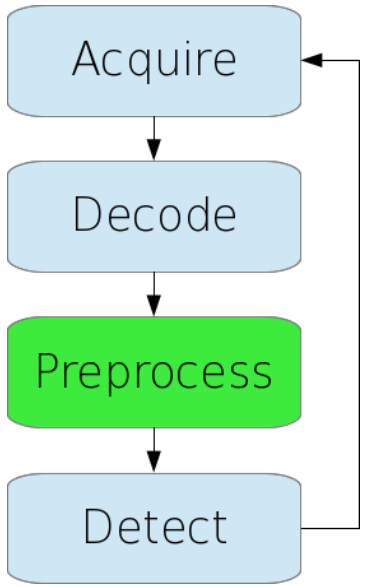
Compressed Http Inspection

- HttpInspect handles compressed Http responses in a naïve way:
  - It uncompresses the data.
  - It sends the uncompressed data for scanning.
- The buffer is uncompressed using zlib library.



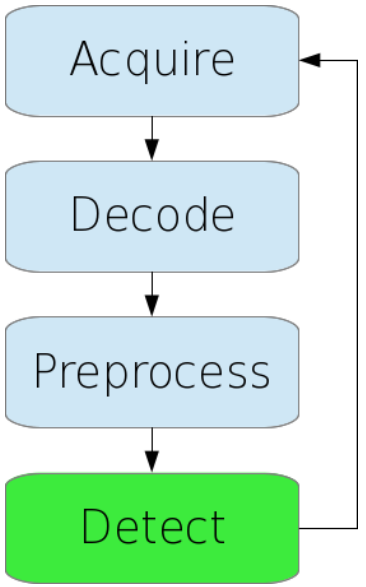
Compressed Http Inspection (Cont.)

- What happens when the compressed data is spanned across multiple buffers?
  - The state of the last uncompressed buffer is used to uncompress the next one.
  - But, the uncompressed buffers are individually inspected.



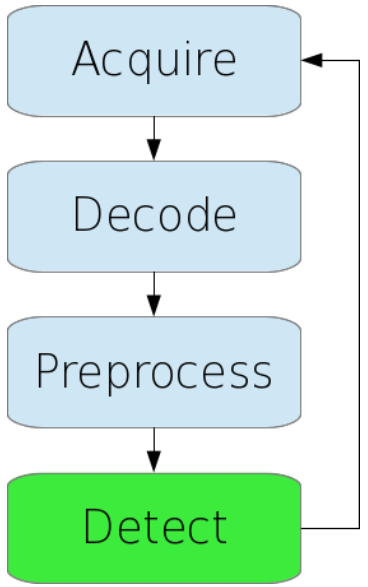
Detect Packet

- Scans buffer for suspicious patterns.
- When a pattern is found, it is handled according to its rules.
  - Pass, Drop, Alert.
- Uses MPSE for pattern matching.

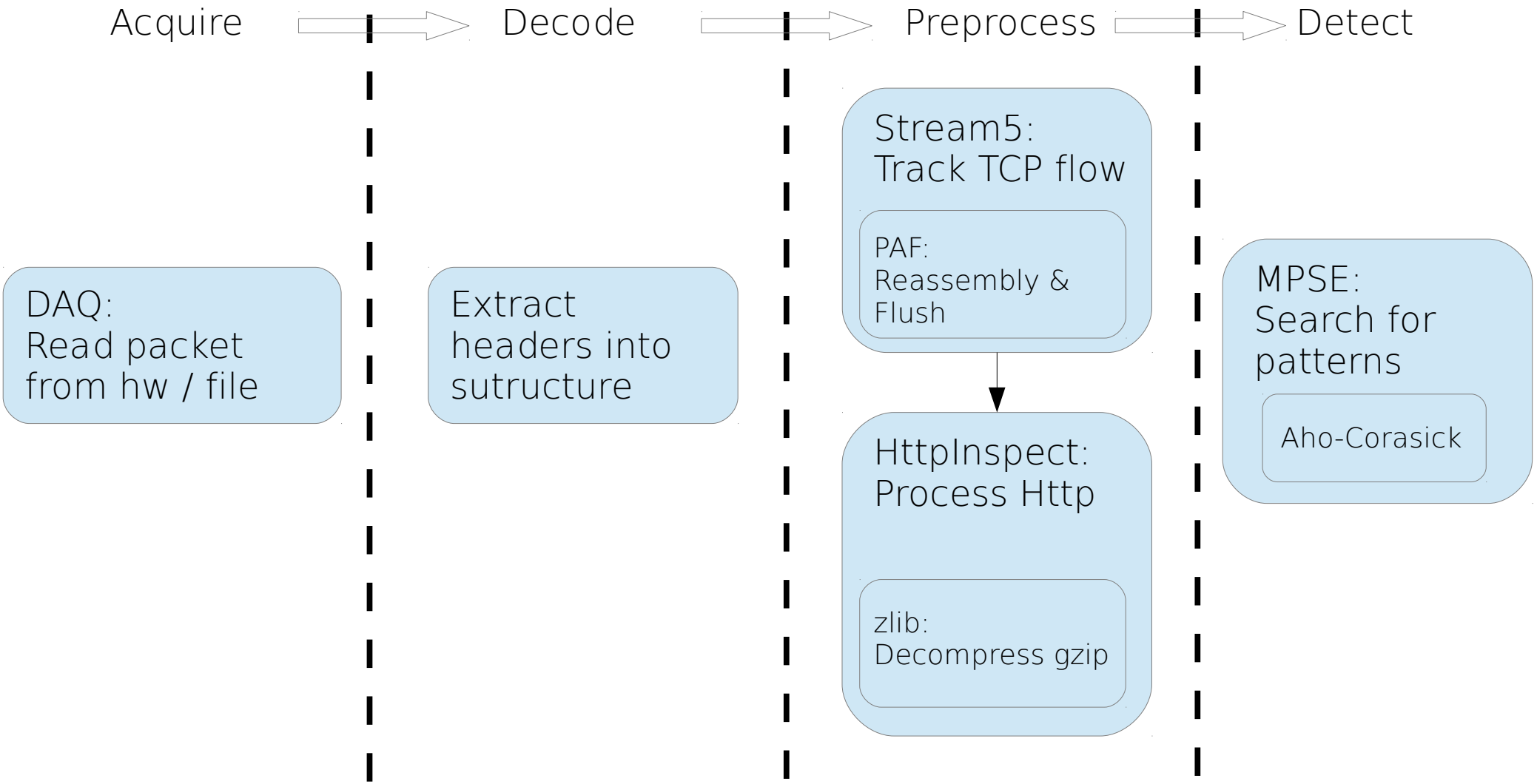


MPSE

- **M**ulti **P**attern **S**earch **E**ngine
- Introduces a generic interface for pattern matching algorithms.
- Actually wraps the configured pattern matching algorithm (usually Aho-Corasick).



Compressed Packet Flow

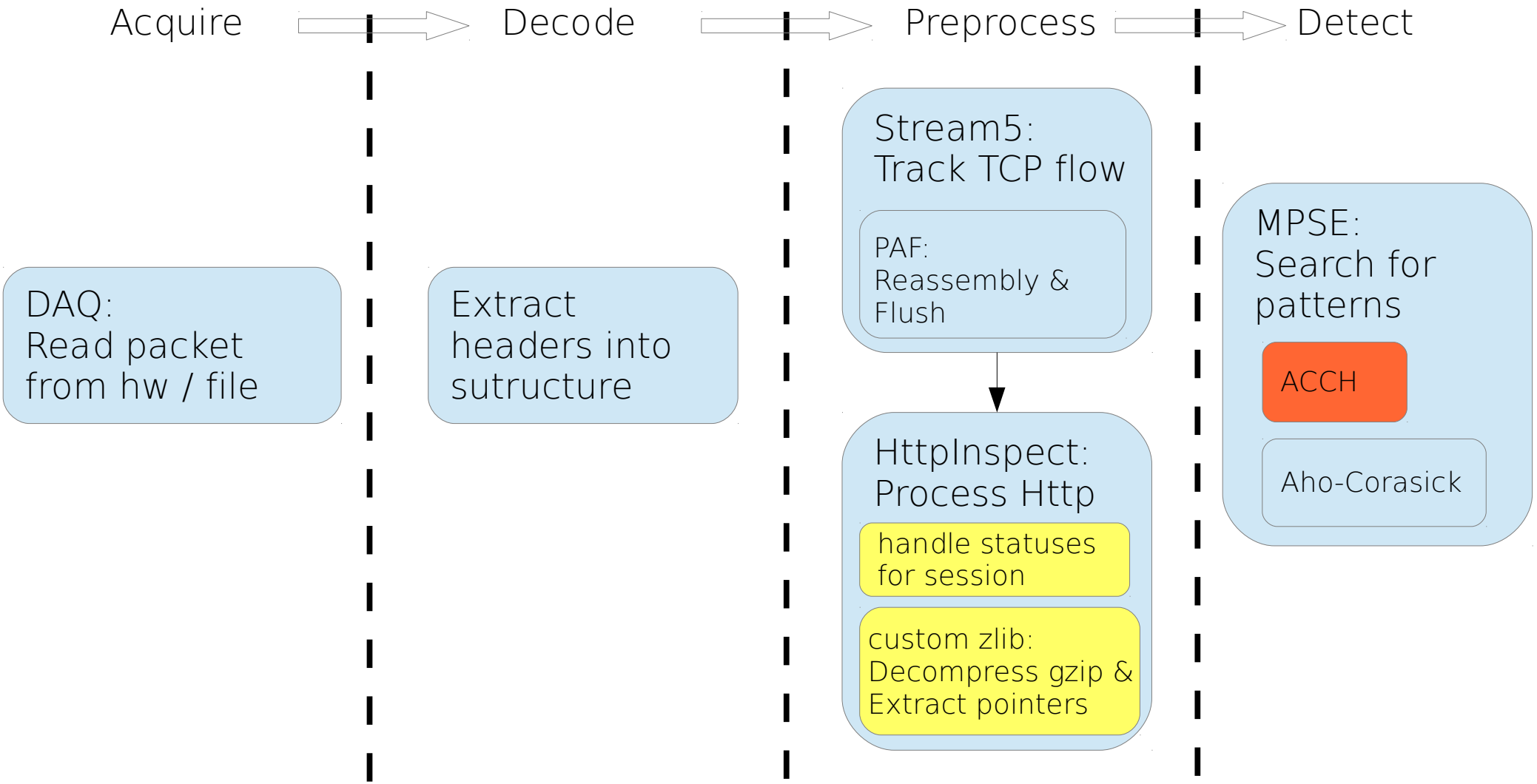


ACCH-Snort Integration

Solution Details

- ACCH as a new search method.
- ACCH will implement MPSE interface.
- ACCH will use regular AC (Snort built-in) to perform the search (when bytes are not skipped).

Compressed Packet Flow



Differences From The Article

- The original article evaluated ACCH in a simplified environment:
  - Offline gzip decompression
  - Working with files: uncompressed and pointers
  - No packet reassembly
  - No circular buffer
  - The uncompressed data is not normalized
- Running ACCH inside Snort is much more complicated.

Pointers Retrieval

- HttpInspect uses zlib for decompression.
  - By calling to inflate().
- inflate() was modified to extract the pointers information to a dedicated buffer.
- The custom version of zlib was statically compiled into Snort.

TCP Sessions

- The compressed data can be spanned across session buffers.

  => We need to maintain the status information of the last 32K bytes during the session.

TCP Sessions (Cont.)

- Recall that Stream5 identifies and tracks TCP sessions.
- Stream5 also allows other preprocessors to attach private data structures to sessions.
- Stream5 API was used to store the last 32K statuses for each session.

Data Modification

- The correctness of ACCH requires that the scanned data must match the compressed data.

  => The uncompressed data should not be altered between the decompression stage and scanning stage!

- How can the the uncompressed data be changed?

Normalization

- HttpInspect can normalize the data before scanning it.
- i.e. javascript normalization (removing extra whitespaces).
- We should disable normalization!

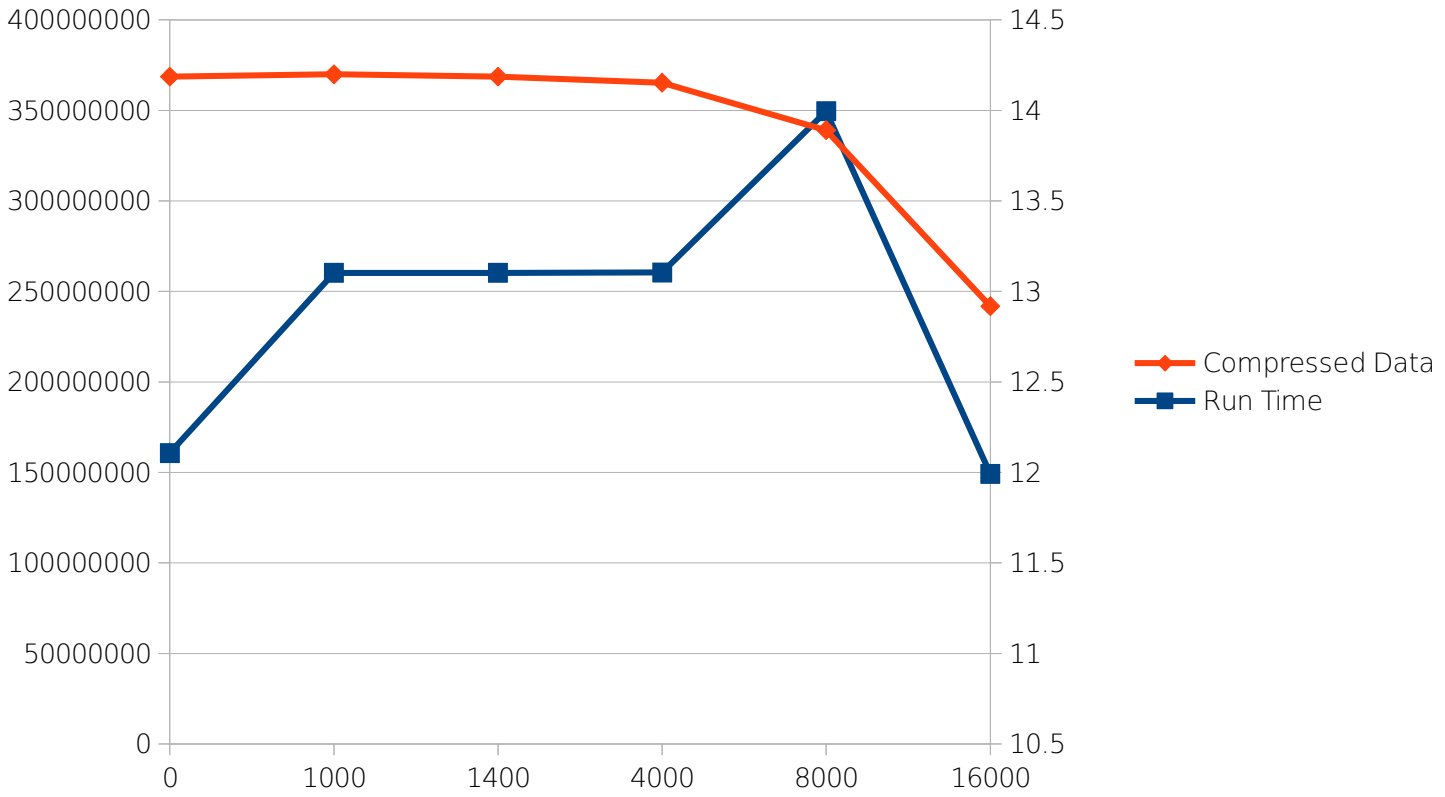
Reassembly

- Snort reassemble packets via Stream5 preprocessor.
- The reassembled data is sent for decompression.
- Limitation: Only up to 64KB can be decompressed.
- More reassembled data => Less decompressed data (anything beyond 64KB is discarded).
- We should limit the size of the reassembled data.

Reassembly Threshold

- The reassembly threshold can be configured.
- Higher threshold => Less data decompressed.
- We get good trade-off when threshold = 1000.

Reassmebly Threshold (Cont.)



Performance Mismatch

- The article claims that ACCH has 60%-70% performance improvement.
- But, during the development tests, ACCH inside Snort did not perform as expected.
- We decided to step back and analyze ACCH performance outside Snort.

ACCH Analysis

ACCH Simulator

- Standalone program, that implements the ACCH algorithm.
- Based on acsmx2 – basic Aho-Corasick pattern matching module from Snort.
- The original version was written by Yaron Koral to analyze ACCH performance.
- The new version is much more compatible with ACCH implementation inside Snort.

ACCH Simulator (Cont.)

- Input:
  - Uncompressed data
  - Pointers data
  - Patterns
  - AC format
  - Scan mode
- Output:
  - Num of matches
  - Num of skipped bytes
  - Running time

AC Formats

- The DFA can be constructed in 2 ways:

Format	Meaning
Full	DFA is represented in memory as full matrix
Sparse	DFA is represented in memory as list of non-default transitions

- Trade-off between run-time and space.

Fundamental Research Assumption

- The article assumed that the performance is determined by **memory accesses**.
  - Therefore, the original simulator scanned each byte seperately.
- New Assumption:
  Function calls & branches may have significant effect on the performance!

Scan Mode

- The new simulator allows us to compare between 3 scan modes:

Scan Mode	Meaning
ACCH	Scans the buffer segment-by-segment
AC	Scans the whole buffer at once
AC Byte-by-Byte	Scans the buffer byte-by-byte

Simulator Tests

- Setup:
  - CPU: Intel i7-2670QM CPU
    - Speed: 2.20GHz
    - Cache:

L1(i)	L1(d)	L2	L3
32K	32K	256K	6144K [2MB]

  - RAM: 8GB [4GB]

* In red – the specs of the original setup from 2008.

Simulator Tests (Cont.)

- Data Set:
  - Uncompressed Data: 2292 Files, 371MB
  - Compressed Data: 58MB
  - Patterns: 9994 patterns

- Statistics:
  - Matches: 0.288%
  - Skipped Bytes by ACCH: 64.10%

Simulator Results

- AC Format: Full
- DFA Size: 130.84MB

Scan Mode	Run Time (Sec)	Bit Rate (Mbit/Sec)
ACCH	3.6578	810.9887
AC	3.1939	928.7854
AC (Byte-by-Byte)	4.7043	630.5913

- ACCH results are **worse** than AC [14%].

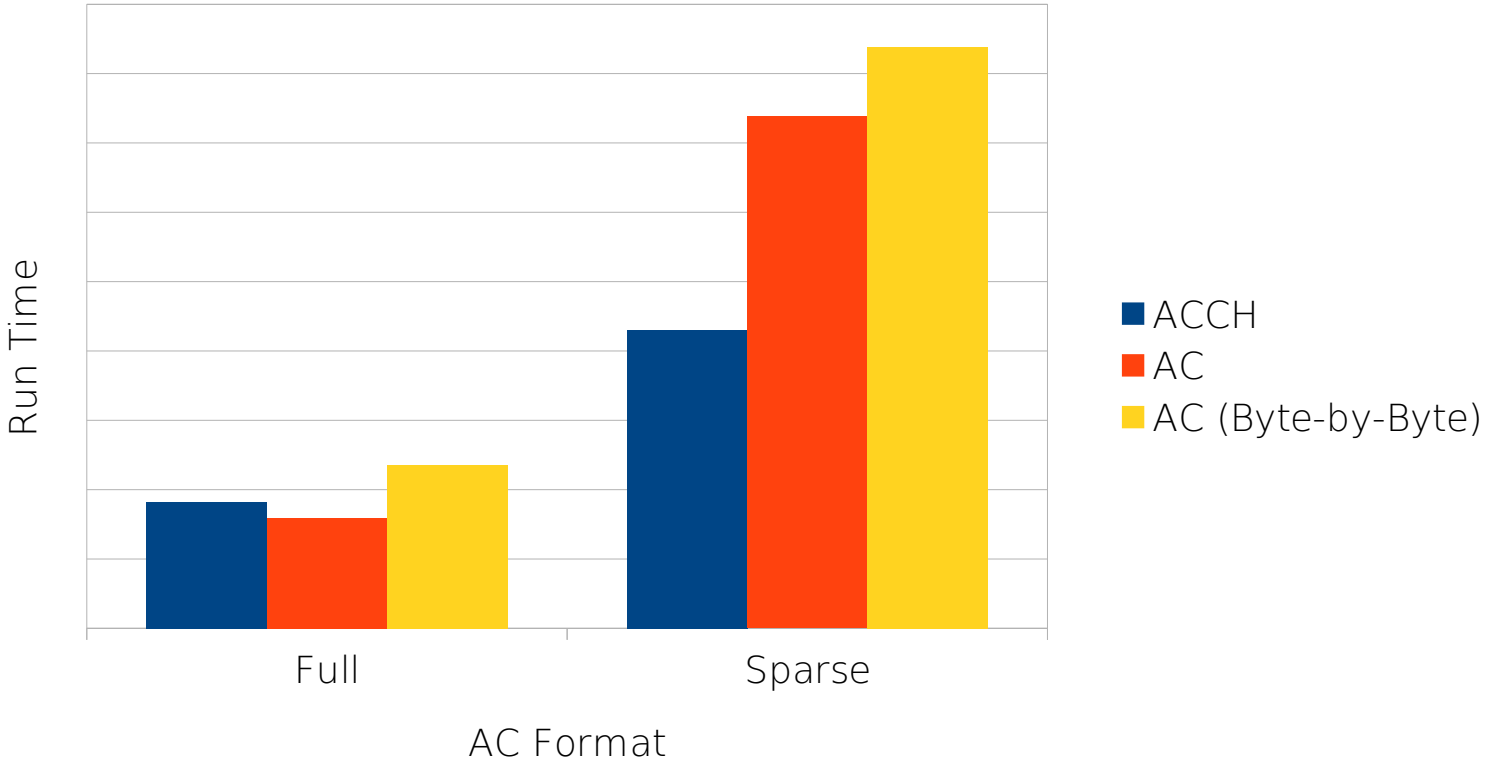
Simulator Results (Cont.)

- AC Format: Sparse
- DFA Size: 63.92MB

Scan Mode	Run Time (Sec)	Bit Rate (Mbit/Sec)
ACCH	8.6201	344.1335
AC	14.7645	200.9210
AC (Byte-by-Byte)	16.7742	176.8481

- ACCH resuls are **better** than AC [42%].

Results Summary



Full vs. Sparse

- Why ACCH gets an improvement for sparse but not for full?
- Access to sparse DFA is much more expenssive than access to full DFA.
- For sparse DFA, finding the next state involves list traversal.
- For full DFA, finding the next state results in direct access (most of the time the DFA entries are stored in the cache).

Intermediate Conclusions

- The simulator does not achieve the run-time results proclaimed in the article.
- When format is full, ACCH is worse than AC.
- When format is sparse, ACCH is better than AC (but not as expected).

  => We should return to the original simulator.

Original Simulator Results

Format	AC (Mbit/Sec)	ACCH (Mbit/Sec)	Improvement
Full	465.5394	816.6068	43%
Sparse	156.8796	389.3298	60%

- In the original simulator, ACCH performed much better than AC for both full and sparse formats.
- While ACCH performaces in both simulators are close, AC is much slower in the original simulator.

Original Simulator Analysis

- Goal: Find explanations to the performance differences between the simulators.
- Means: Walking through the sources and spotting problematic code, that can affect the performance.
- Note: The analysis described in the next slides is focusing on full format, but is relevant to sparse format as well.

Original Simulator Analysis (Cont.)

- Observation #1:
  The original simulator run AC inefficiently.
- We can improve AC performace by applying the following changes.

ID	Change	AC	ACCH	Improvement
0	original simulator	465.5394	816.6068	43%
1	scanAC as inline	499.4566	847.6311	41%
2	no segment traversal in plain scan	522.1409	847.6311	38%

Original Simulator Analysis (Cont.)

ID	Change	AC	ACCH	Improvement
3	no status management in AC	676.8416	847.6311	19%
4	removed byte-by-byte scanning in plain scan	820.6497	847.6311	3%

Original Simulator Analysis (Cont.)

- Observation #2:
  The original simulator does not store the statuses in circular buffer.

ID	Change	AC	ACCH	Improvement
5	changed status vector type to circular buffer	820.6497	344.8108	-138%

Original Simulator Analysis (Cont.)

- Observation #3:
  ACCH impelemtation is not optimal
- We can improve ACCH performace by applying the following changes.

ID	Change	AC	ACCH	Improvement
6	statuses are moved to the circular buffer at the end of segment scan	820.6497	386.2104	-112%
7	optimizing the search for min match and max uncheck	820.6497	500.0273	-64%

Original Simulator Analysis (Cont.)

ID	Change	AC	ACCH	Improvement
8	finding both min match and max uncheck in one scan	820.6497	504.7371	-52%
9	optimizing old statuses copy	820.6497	698.3914	-17%

Analysis Results Summary

Simulator	Format	AC (Mbit/Sec)	ACCH (Mbit/Sec)	Improvement
Original	Full	820.6497	698.3914	-17%
	Sparse	199.6243	344.7446	42%
New	Full	928.7854	810.9887	-14%
	Sparse	200.9210	344.1335	42%

Intermediate Conclusions

- ACCH improves performace in some configurations.
  - For sparse, but not for full.
- The performance is not determined only by memory accesses, but also by -
  - Function calls
  - Branches

Can We Do Better?

- Most of the overhead is a result of the circular buffer management.
- Removing the circular buffer will improve the performance significantly.

Back To Snort

- Can we remove the circular buffer from ACCH implementation inside Snort?
- In practice, zlib (used by HttpInspect) already implements circular buffer.
- Theoretically, we could extend this buffer to store also the statuses of the last 32KB.
- Problem: The solution breaks current Snort architecture!

Snort Tests

- Setup:
  - CPU: Intel i7-2670QM CPU
    - Speed: 2.20GHz
    - Cache:

L1(i)	L1(d)	L2	L3
32K	32K	256K	6144K

  - RAM: 8GB

Snort Tests (Cont.)

- Data Set:
  - Traffic:
    - PCAP file: Generated from simulator data set
  - Rules:
    - 1 rule file: Genereated from simulator patterns
    - 9994 rules

Snort Tests (Cont.)

- Statistics:
  - Uncompressed Htmls: 370MB
  - Compressed Htmls: 58MB
  - PCAP: 68MB
  - Matches: 0.315%
  - Skipped Bytes by ACCH: 49.29%

Snort Results

- AC Format: Full
- DFA Size: 130.84MB

Scan Mode	Total Run Time (Sec)	Total Bit Rate (Mbit/Sec)	Scan Run Time (Sec)	Scan Bit Rate (Mbit/Sec)
ACCH	11.2620	47.6564	4.6675	114.9881
AC	11.2966	47.5102	4.6591	115.1939

- ACCH results are as good as AC.

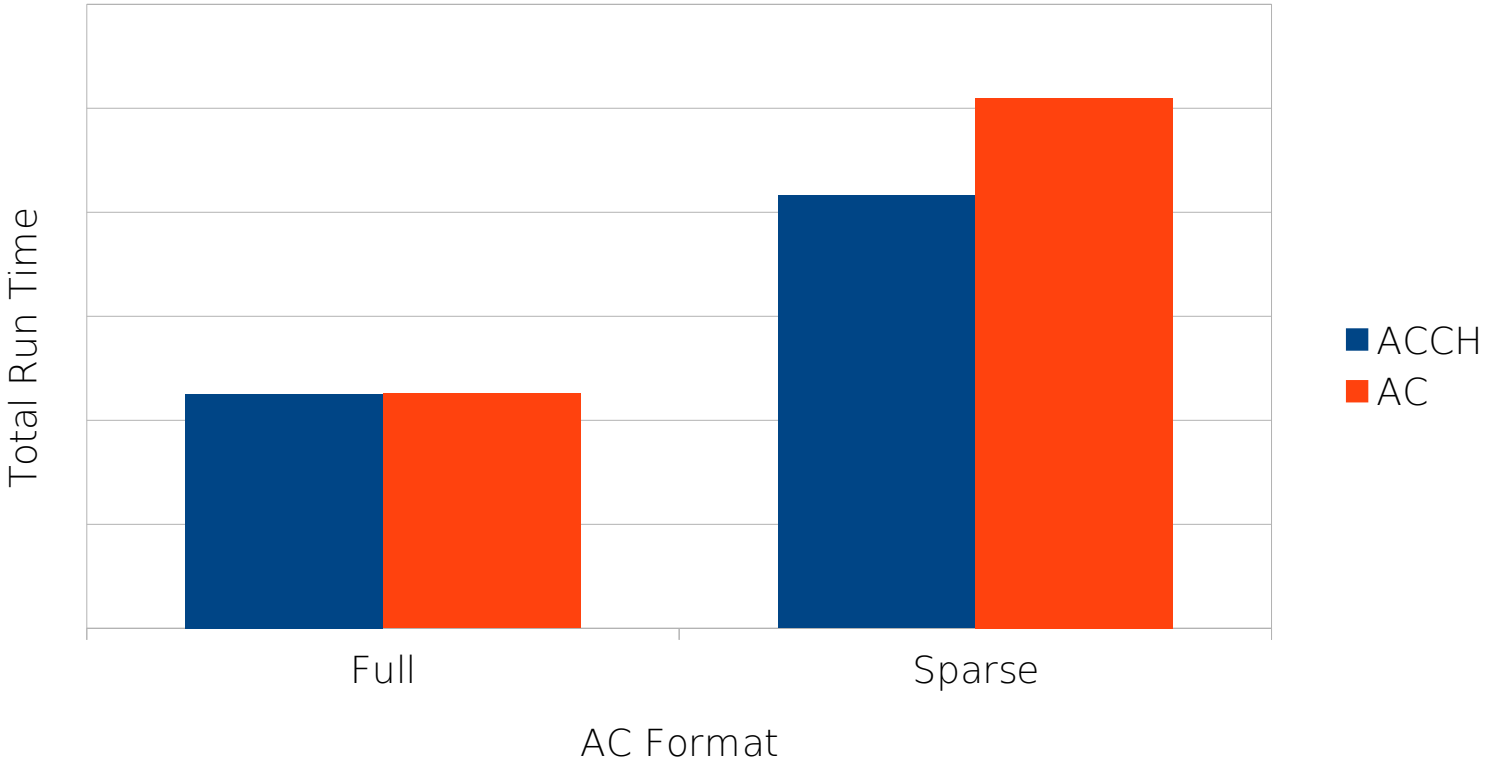
Snort Results (Cont.)

- AC Format: Sparse
- DFA Size: 63.92MB

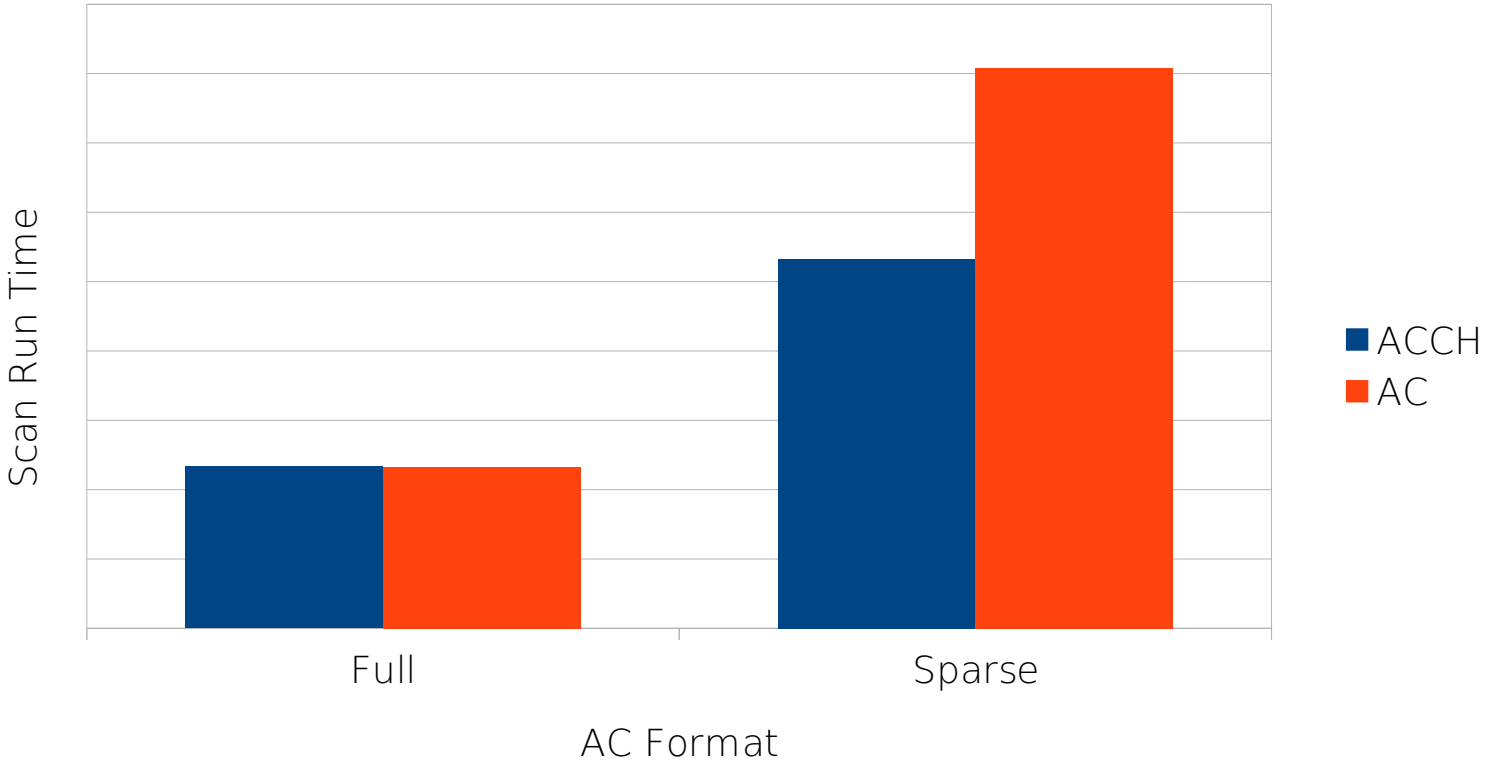
Scan Mode	Total Run Time (Sec)	Total Bit Rate (Mbit/Sec)	Scan Run Time (Sec)	Scan Bit Rate (Mbit/Sec)
ACCH	20.8161	25.7831	10.6651	50.3235
AC	25.5024	21.0452	16.1707	33.1899

- ACCH resuls are **better** than AC.

Results Summary



Results Summary (Cont.)



Summary

- Original assumption:
  Performace depends only on memory accesses.
- In practice, performance is also affected by:
  - Function calls
  - Branches (if, switch-case, loops)
  - Current cache technology
  - Actual Snort architecture

Summary (Cont.)

- ACCH still improves performance in some configuration.
  - But, less than expected.

Future Work

- Remove circular buffer from ACCH implementation – may require re-design of Snort.
- Extend ACCH to support pattern matching algorithms other than Aho-Corasick.

Questions?

Thanks for listening!