The Interdisciplinary Center, Herzliya
Efi Arazi School of Computer Science

Integrating Aho-Corasick based algorithm for Compressed Traffic (ACCH) Inside Snort

M.Sc. final project submitted in partial fulfillment of the requirements towards the M.Sc. degree in computer science

by

Adir Gabai

This work was carried out under the supervision of

Prof. Anat Bremler-Bar

and

Dr. Yaron Koral

March 2015

Acknowledgments

I would like to thank my advisor Prof. Anat Bremler-Bar for her guidance throughout my final project. Her professional help and patience were vital for the success of this project. I would also like to thank Dr. Yaron Koral for his technical help. His wise advices helped me to face the challenges along the way.

During the work on the project, I was exposed to an exciting field in computer science, which I was not familiar with. I learned a lot about deep packet inspection and the solutions to perform such inspection at wire speed. I gained some important tools, which helped me to properly analyze the gaps between theory and practice, and successfully finish this project.

Abstract

Snort is a very popular network intrusion detection system (NIDS). One of its main methods of inspection is to scan the traffic payload for malicious content. Unfortunately, Snort deals with compressed http traffic in a näive way. It first decompresses the traffic, and then performs a multi-pattern matching scan. Thus, Snort suffers from a performance penalty in pattern matching on compressed http data.

Recently, an algorithm that tackles the above problem, called ACCH, was proposed. The algorithm presented impressive performance results on simulation environment but not on a real Snort setup.

In this work we present the integration of ACCH algorithm inside Snort in order to improve the performance over compressed http traffic. However, the improvement was not significant as expected. We show an analysis of ACCH algorithm and a comparison between our implementation and the original implementation of ACCH.

Our analysis reveals a new set of properties, which are relevant when trying to incorporate an algorithm with Snort. The original ACCH paper assumed a simplified model which should be extended to include the properties we found. We modify the original algorithm to support these properties and suggest future directions to improve furthermore the performance of the compressed traffic scanning algorithm.

Contents

1 Introduction 7

2 Background 8
- 2.1 Aho-Corasick 9
- 2.2 Compressed Http 10
  - 2.2.1 Gzip 10
- 2.3 ACCH Overview 11
- 2.4 Snort Overview 13
  - 2.4.1 Snort Rules 14
  - 2.4.2 Packet Loop 14
  - 2.4.3 Acquire 15
  - 2.4.4 Decode 15
  - 2.4.5 Preprocess 16
  - 2.4.6 Stream5 17
  - 2.4.7 HttpInspect 18
  - 2.4.8 Detect 18

3 Related Work 18

4 Architecture Overview and Considerations 19
- 4.1 Considerations 20
- 4.2 Pointers Retrieval 21
- 4.3 TCP Sessions 21
- 4.4 Data Manipulation 21

5 ACCH Simulator Analysis 23
- 5.1 ACCH Simulator 23
  - 5.1.1 Simulator Inputs and Outputs 23
  - 5.1.2 AC Format 24
  - 5.1.3 Scan Mode 24

5.2 Simulator Tests 25
5.2.1 Setup Description 25
5.2.2 Data Set 25
5.2.3 Simulator Results 26
5.2.4 Full vs. Sparse 27
5.3 Original Simulator Analysis 27
5.3.1 Observation 1: AC is implemented inefficiently 28
5.3.2 Observation 2: Missing use of circular buffer 29
5.3.3 Observation 3: ACCH is not optimized for circular buffer 30
5.3.4 Analysis Results Summary 31
5.4 Circular Buffer Considerations 31

6 Snort Tests **32**
6.1 Setup Description 32
6.2 Data Set 33
6.3 Snort Results 33

7 Conclusions and Future Work **35**

List of Figures

1 Aho-Corasick DFA 10
2 LZ77 compression 11
3 Compressed packet flow 15
4 Processing an Ethernet packet 16
5 Compressed packet flow revisited 20
6 New simulator results summary 27
7 Snort total time results summary 34
8 Snort scan time results summary 34

List of Tables

1 Simulator inputs and outputs 24
2 Comparison of performances for full DFA 26
3 Comparison of performances for sparse DFA 26
4 Original simulator results 28
5 Original simulator modifications 31
6 Simulators results 31
7 Snort results for full format 33
8 Snort results for sparse format 33

1 Introduction

Deep packet inspection (DPI) is a method of traffic filtering, that examines both the header and the payload of a packet. It can be implemented by intrusion detection systems (IDS) to detect suspicious patterns or anomalies. Snort, one of the most popular IDS systems that exist today, implements DPI using the Aho-Corasick pattern matching algorithm [6], which scans the data in a linear complexity. Snort can inspect various protocols, including http. The current version of Snort handles compressed http data in a näive manner. It first decompresses the traffic, and then it scans the uncompressed traffic for patterns.

The default Aho-Corasick algorithm can be replaced with ACCH algorithm [10], which extends Aho-Corasick to support scanning of compressed traffic in a more efficient manner. This way we can improve Snort performance over compressed http traffic. At the original work of ACCH the authors have provided an algorithm which was tested only on simulation environment and was never integrated with Snort.

This work describes the integration of ACCH algorithm within Snort. We had not only added a new search algorithm to Snort, we also had to modify Snort to support ACCH. Unfortunately, the results did not match the theory, and the improvement of ACCH was less than expected. The original work of ACCH assumed that the performance is determined mainly by memory access. After analyzing ACCH we discovered that there are other meaningful factors, that impact the performance.

In this work we describe the way these factors were incorporated to our implementation of ACCH in Snort and also suggest some architectural changes that should take place in future Snort versions to make it more robust to scanning compressed traffic.

The structure of the remaining of this work is as follows: In section 2 we introduce the background for this work, which includes a brief overview of ACCH and Snort. In section 3 we survey related works, that are intended

to improve Snort performance. In section 4 we describe the technical details of the proposed solution and the considerations we made. Section 5 is the heart of the project. In this section we analyze the performance of ACCH. Section 6 summarizes the results of ACCH implementation inside Snort. Finally, we present our conclusions in section 7.

2 Background

Deep packet inspection (DPI) is a form of network packet filtering that scans both the header and the payload of a packet for certain patterns. The packet is filtered according to the scan results and pre-defined policies. It is currently being used by enterprises, service providers and governments in a wide range of applications. For example:

1. *Intrusion Detection / Prevention System*: Intrusion detection system (IDS) watches the network traffic and tries to match it against signature-based or anomaly-based pre-configured rules. If it detects anything suspicious, the IDS sets off an alarm. An IPS has all the capabilities of IDS with the added ability to filter out suspicious traffic.

2. *Leakage Prevention*: DPI can be used to identify a leakage of confidential data from a private network to the internet.

3. *Protocol Classification*: Since DPI examines the packet payload, it can be used to identify and classify the traffic according to the application layer protocols.

4. *Targeted Advertising*: ISPs can use DPI to gain information about their customers which can be used by companies specialized in targeted advertising.

5. *Quality of Service*: DPI allows operators to distribute equitable bandwidth to all users by preventing network congestion. Additionally,

higher priority can be allocated to special services such as VoIP, which requires low latency.

DPI may be implemented by various pattern matching algorithms. One of the most common pattern matching algorithm is Aho-Corasick [6].

2.1 Aho-Corasick

Aho-Corasick is a string matching algorithm [6]. It is a kind of dictionary matching algorithm that locates elements of a finite set of strings within an input text. Aho-Corasick constructs a deterministic finite automaton (DFA) from the set of strings to be matched. Therefore, it can find suspicious string in text of size n with running time complexity of $O(n)$.
In figure 1 we can see the DFA constructed by Aho-Corasick for the strings *he*, *his*, *she* and *hers*. The algorithm starts at state 0. Given an input byte b, the algorithm looks for a valid transition (solid edge). If there is one, the algorithms moves to the next node. Otherwise, the algorithm uses the failure transition (dashed edge) to find a node from which a valid transition will be found, and so forth.

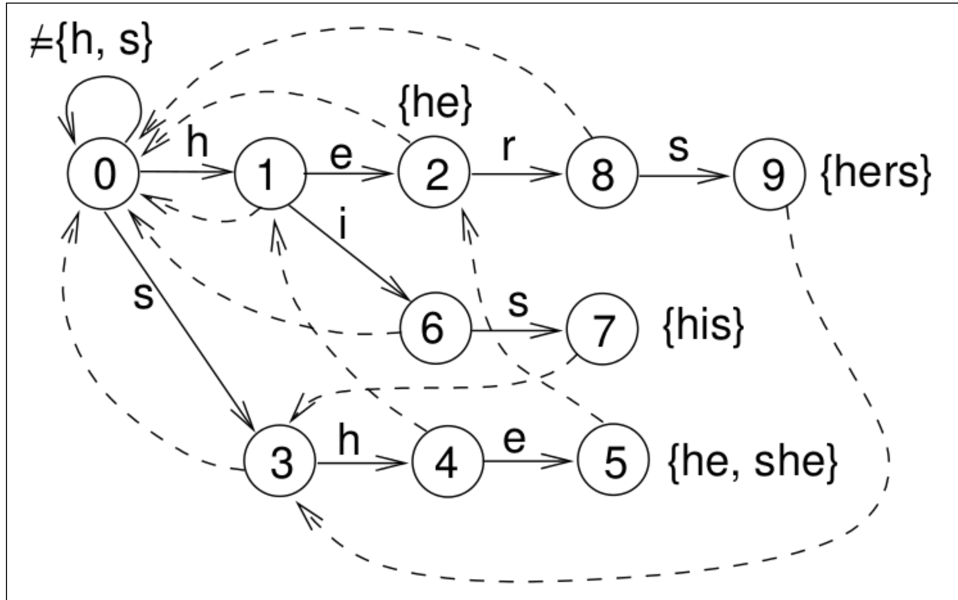

Figure 1: Aho-Corasick DFA

2.2 Compressed Http

Http compression is used to achieve a better utilization of the bandwidth. Http data is compressed before it is being sent from the server to the client. The client will announce in the http request what compression methods are supported. If the server supports one or more of the compression methods, it will use one the methods to compress the data in the response. The most common compression algorithms include deflate and gzip.

2.2.1 Gzip

One of the HTTP 1.1 recommended compression algorithms, and the most commonly used, is the gzip algorithm [2]. It is based on the deflate compression algorithm [1]. The compression constitutes two phases:

1. *LZ77 Compression*: Used to replace repeated sequences with pointer to the original sequence. This way gzip decreases string representation.

2. *Huffman Coding*: Used to decrease symbols representation.

A string is replaced with a pointer only if it was seen in the last 32KB. The last 32KB are stored in a circular buffer, often called *sliding window*. A pointer is consisted of two numbers - distance to the original string and length of the original string.
Figure 2 demonstrates how LZ77 compression works. The additional instances of the strings *bbc* and *fgea* were replaced with pointers to the original strings.

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			

Figure 2: LZ77 compression

The open source zlib library [5] implements the gzip algorithm, and it is widely used by other applications.

2.3 ACCH Overview

ACCH stands for Aho-Corasick based algorithm for Compressed Http [10]. The main idea behind ACCH is that the scan of repeated string can be accelerated using the fact that gzip pointers are pointing to an already scanned strings.
LZ77 is an adaptive compression as each symbol is determined dynamically by the data. Therefore, there is no way to perform DPI without decompressing the data first. Since decompression is a fairly inexpensive process, the real challenge is to optimize the scanning of the decompressed data. The

idea behind the acceleration algorithm is to use the information gathered by the decompression phase to skip scanning significant parts of the data. An LZ77 pointer represents a repeated string that was already scanned. Therefore, it is possible to skip scanning most of it without missing any pattern. Still, tracking previous matches does not suffice due to cases where a pattern crosses a border of a repeated string. For example, given a pattern nbc and an input string $abcdnbxn(7,5)c$, while no match occurs at the repeated string $bcdnb$, there are matches occurring on both its left and right borders. This problem is tackled by the DFA-based ACCH algorithm. Upon encountering a repeated string it works as follows:

1. Scan the left border of the repeated string and update scan results.
2. Check whether the previous scan results of the repeated string contain matches.
3. Scan the right border of the repeated string and update scan results.
4. Update estimated scan results of skipped bytes within the repeated string.

In ACCH, scan results of previous bytes are stored in a 32K-entries Status-Vector. Its values are determined by DFA-Depth(s), the length of the shortest path from root to state s. There are three possible status values:

- MATCH: A pattern was matched (the match ends in the scanned byte).
- UNCHECK: DFA-Depth(s) is below a threshold parameter CDepth (in practice, CDepth is set to 2).
- CHECK: Otherwise.

ACCH uses the Status-Vector in the following manner:

1. *Left Border Scan*: Upon processing a repeated string, ACCH scans j bytes until reaching a state s where $j \geq$ DFA-Depth(s). From this

point on, any pattern prefix is already included in the repeated string area. We refer to the first j bytes of the repeated string as the left border of the pointer.

2. *Repeated Pattern Detection*: This procedure examines the corresponding Status-Vector values of the referred string, starting at its $(j+1)$th byte. If some byte at position m has a MATCH value, ACCH checks if the previously matched string is repeated entirely. This is done by estimating the DFA-Depth of the matched byte.

3. *Right Border Detection*: ACCH determines the position to start inspection again such that no pattern (whose prefix is part of the repeated string suffix) is missed. This is done by estimating the DFA-Depth of the repeated string's last byte (say this estimation is d) and scanning (from state 0) the last d bytes of the repeated string.

4. *Update Status of Skipped Bytes*: Skipped bytes cannot obtain their status from the scan procedure. Therefore, ACCH copies the status value for the skipped bytes from the corresponding referred string. Note that since ACCH skips only after it ensures that there is no pattern whose prefix started prior to the repeated string, the value of the estimated DFA-Depth could be only equal or greater than its actual value (namely, setting a byte as CHECK while it should be UNCHECK). Such a mistake leads to minor reduction in the amount of skipped bytes but never to a miss-detection.

2.4 Snort Overview

Snort is a modern security application with three main functions: it can serve as a packet sniffer, a packet logger or a network-based intrusion detection system (NIDS). Snort provides signature-based rule matching and alerting mechanisms. Snort is an open source [3], but there are many commercial

solutions and products using Snort. The most famous is Sourcefire [4], which also contributes to Snort.

2.4.1 Snort Rules

A rule is a set of instructions designed to pick out network traffic that matches a specified pattern, and then takes a chosen action when it sees traffic that matches. A rule consists of a rule header and a rule body. A rule header describes the traffic on a packet level. A rule body fills in additional details such as content, references and documentation. One of the most highly praised functions of Snort is the capability for the users to write their own rules, in addition to the large rule-base that Snort comes with by default. Rules are useful for matching traffic flows, particular combinations of ports and IP addresses, particular contents of packets, protocol options and much more.

2.4.2 Packet Loop

Every packet is passing through four stages inside Snort: Acquire, Decode, Preprocess and Detect.
Figure 3 demonstrates the flow of a compressed http packet in Snort.

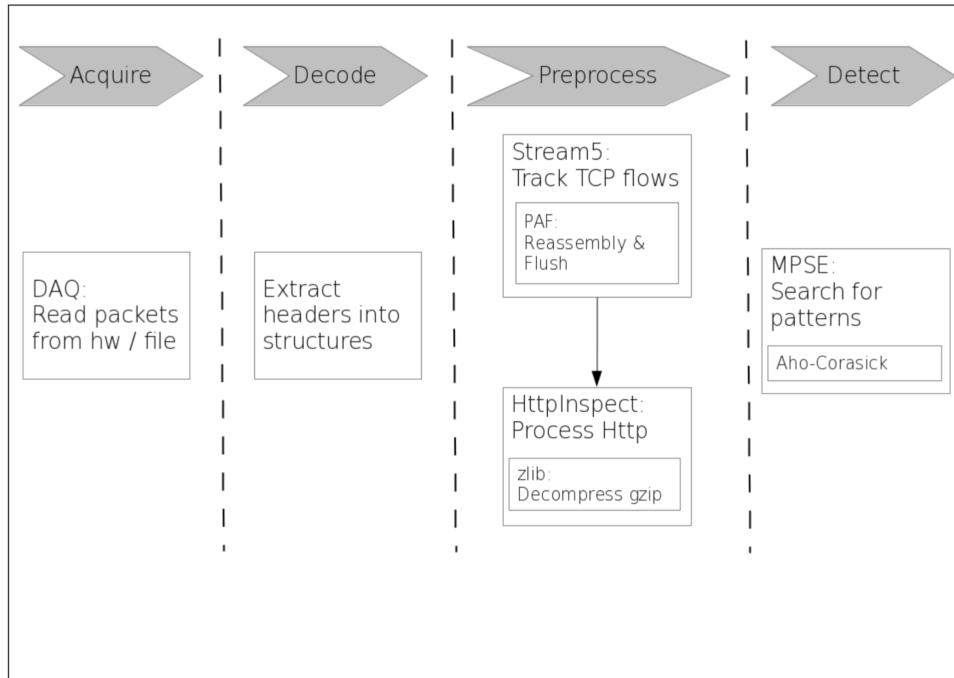

Figure 3: Compressed packet flow

We will now describe those stages in detail.

2.4.3 Acquire

Every packet is first read either from a network interface or a pcap file. Snort uses DAQ, Data Acquisition library, for packet I/O. DAQ actually adds an abstraction layer that facilitates operation on a variety of hardware and software interfaces without requiring changes to Snort.

2.4.4 Decode

Whenever a packet is read, Snort decodes it. Based on the link layer, Snort calls different functions to handle decoding the link layer. Snort supports

various link layers: Ethernet, 802.11, Token Ring, PPP, etc. Each link layer decoder function sets various pointers into the packet structure. Then, based on the information it decoded, it sets up pointers into the packet structure where the next layer starts, and calls the next layer's decoder. Each layer has a hard-coded list of layers it supports underneath. Figure 4 shows how standard Ethernet packets are decoded. In the standard Ethernet case, incoming packets feed into DecodeEthPkt, which calls DecodeIP, which calls DecodeTCP.

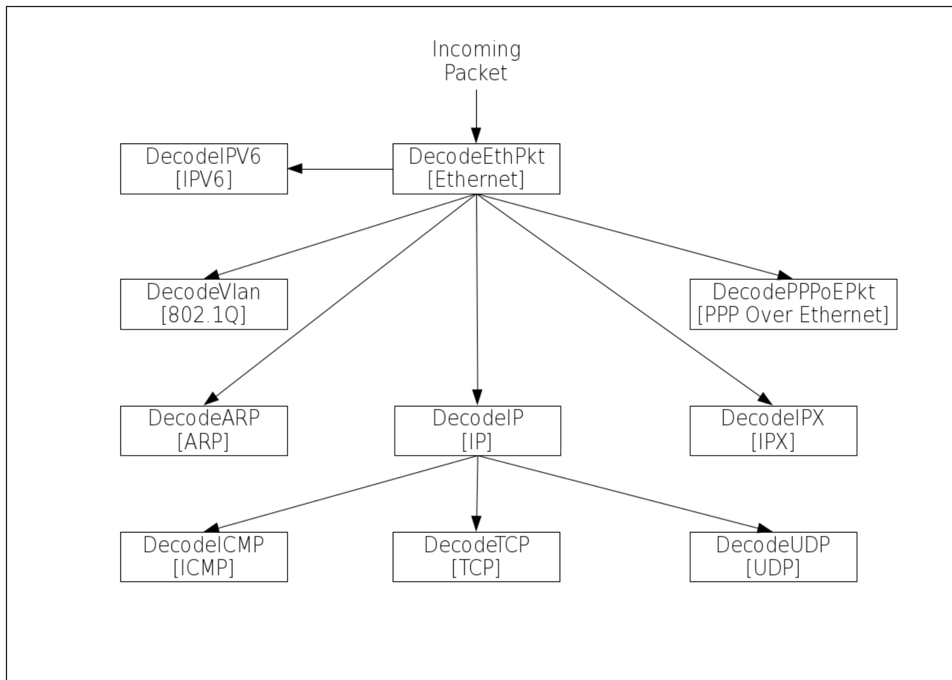

Figure 4: Processing an Ethernet packet

2.4.5 Preprocess

Snort preprocesses the decoded packet before scanning it. Some of the major purposes for which preprocessors are used are -

1. Reassembly of packets
2. Decoding protocols
3. Normalization
4. Non-rule or anomaly-based detection

Snort is shipped with numerous preprocessors, like HttpInspect and Stream5 which we will present now.

2.4.6 Stream5

Stream5 preprocessor is a target-based TCP reassembly module for Snort. It is capable of tracking TCP sessions. Stream5 has two goals:

1. *Anomaly Detection*: TCP protocol anomalies, such as data on SYN packets, data received outside the TCP window, and many more, can be detected by Stream5. Some of these anomalies are detected on a per-target basis. For example, a few operating systems allow data in TCP SYN packets, while others do not.
2. *Stream Reassembly*: Many attacks will definitely be spread across several packets and thus will be undetectable to a nonsession-reassembling rule-matching IDS - that is the motivation for stream reassembly. Stream5 preprocessor reassembles the TCP stream so that Snort can try rule matches against the whole of the flowing data. Simply put, Stream5 combines all the data in a stream into a large über-packet that can then be passed through the other preprocessors and then the detection engine.

Stream5 fully supports the Stream API, allowing other preprocessors to dynamically configure reassembly behavior as required by the application layer protocol, identify sessions that may be ignored (large data transfers,

etc.), and update the identifying information about the session (application protocol, direction, etc.) that can later be used by rules.

2.4.7 HttpInspect

HttpInspect is a generic http decoder for user applications. Given a data buffer, HttpInspect will decode the buffer, find http fields, and normalize the fields. HttpInspect works on both client requests and server responses. HttpInspect looks for http fields on a packet-by-packet basis, and will be fooled if packets are not reassembled. This works fine when there is another module handling the reassembly, such as Stream5, but there are limitations in analyzing the protocol, as we will see later.

2.4.8 Detect

After the preprocessing is finished, Snort is ready for detection. The preprocessed buffer is scanned for patterns using MPSE, Multi Pattern Search Engine. The MPSE module calls to the underlying multi-pattern search algorithm.

3 Related Work

Many works try to improve the performance of intrusion detection systems. Most of them use Snort as a case study, due to its popularity.
This work deals with DPI over compressed traffic. The basic reference algorithm is ACCH, that was suggested in [10] as a new approach for accelerating compressed http traffic inspection. In this work we implement ACCH inside Snort instead of evaluating it in an isolated simulation environment. There are other more recent works that tackle the problem of DPI over compressed traffic. Becchi et al. [8] extend ACCH for the case of regular

expression matching over compressed http. Furthermore, redundant bytes in the packet can be identified, so their redundant scan can be skipped as shown in [12].

An IDS can take advantage of the hardware, such as the graphic card, as suggested in [13], or even a ternary content addressable memory (TCAM) as shown in [14].

We can gain improvement not only from better algorithms, but also from better configuration. For example, Salah et al. [11] demonstrate how Snort performance can be improved by changing the default configuration of the Linux networking subsystem. Alhomoud et al. [7] provide comparison of the performance of two IDS systems and recommendation for the ideal environments for these systems.

4 Architecture Overview and Considerations

The idea was to implement ACCH inside Snort as a new multi-pattern search algorithm, which will be wrapped by the MPSE module. Since ACCH is based on Aho-Corasick, the implementation will use Snort built-in Aho-Corasick module (acsmx2).

Figure 5 describes the new flow of a compressed http packet inside Snort when ACCH is activated. The green and blue rectangles specifies where we intervened inside Snort. The blue rectangle represents the implementation ACCH algorithm, while the green rectangles represents the code sections, we had to modify in order to support the ACCH implementation.

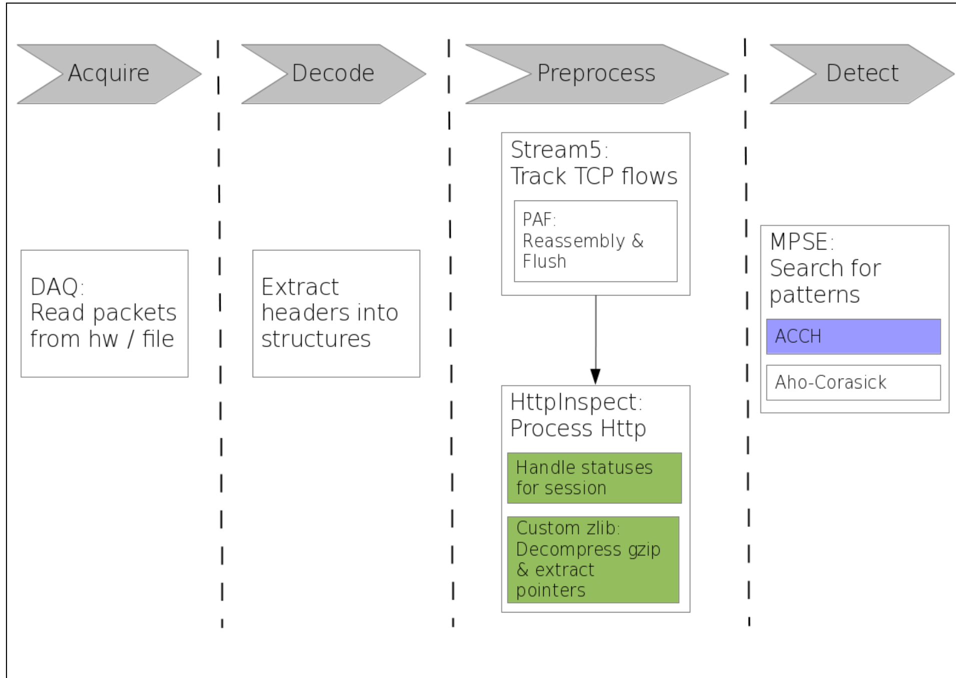

Figure 5: Compressed packet flow revisited

4.1 Considerations

The article evaluated ACCH algorithm in a simplified environment. Integrating ACCH inside Snort is much more complicated and involves some challenges:

1. How to retrieve the pointers information needed by ACCH (4.2)?
2. How to deal with TCP sessions rather than only packets (4.3)?
3. How to deal with manipulated data (4.4)?

4.2 Pointers Retrieval

ACCH must know pointers locations and lengths in order to be able to skip bytes. Therefore, HttpInspect has to prepare this information and supply it to ACCH.

HttpInspect calls $inflate()$ function of zlib library to decompress compressed http responses. We had to modify $inflate()$ function to extract the pointers information into a dedicated buffer, which is passed to ACCH search function. The custom version of zlib was statically compiled to Snort.

4.3 TCP Sessions

The compressed data can be spanned across multiple buffers of the same TCP session. In order to scan the current buffer, ACCH must know the last 32K statuses of the previous scanned buffer. That is because pointer in the current buffer may point to a byte sequence in the last 32KB of the previous buffer. Therefore, we need to maintain a state of 32K statuses between session buffers.

We can do that with some assist from Stream5 preprocessor. Recall that Stream5 identifies and tracks TCP sessions. It also allows other preprocessors to attach private data structures to sessions, such that these data structure are available when the next buffer of the same session is processed. We use this capability within HttpInspect to receive the last 32K statuses calculated for the previous buffer, and store the last 32K statuses calculated for the current buffer.

4.4 Data Manipulation

ACCH requires that the scanned data will match the uncompressed data. Any change to the data may have a significant effect on the search results. HttpInspect, as a preprocessor, may manipulate the data in several ways:

1. *Decompression*: HttpInspect decompresses the reassembled buffer using zlib. Unfortunately, only up to 64KB can be decompressed from a single TCP stream due to a Snort limitation (the data is decompressed into a fixed size buffer).

   Hence, the more packets Stream5 assembles from the same stream, the less data is decompressed. That's because anything beyond 64KB is discarded. We can handle this issue using one of the following approaches:

   (a) When ACCH receives a buffer of size 64KB to scan, it will assume that the last 32K statuses are all fixed to CHECK. Thus, a pointer that is pointing to a byte sequence in the previous buffer will be fully scanned without any skipping.

   (b) The reassembly threshold, which is the maximum amount of data Stream5 reassembles before flushing it, can be limited. This way we assure that no decompressed data is discarded.

   In the first method we may not scan the whole data, because bytes may be discarded. Therefore, we decided to choose the second approach.

2. *Normalization*: HttpInspect can normalize the data before sending it to detection. The normalization enhance the detection engine ability to identify attacks. For example, javascript obfuscation is a technique being increasingly used to modify the malicious javascript code and make it difficult for analysis or detection. Such obfuscation may involve adding extra whitespaces. A normalization can be used to remove those extra whitespaces.

   Since normalization takes place after decompression, it may change string location, which makes it inconsistent with the pointer offset. In order to use ACCH inside Snort, we had to disable all HttpInspect normalizations through Snort configuration.

5 ACCH Simulator Analysis

The article claims that ACCH gets an improvement of 60%-70% over AC. But, during the development tests we realized, that ACCH inside Snort is not performing as expected. In some configurations ACCH did not improve the performance at all, while in other configurations the improvement was not enough. Therefore, in order to understand the results, we decided to analyze ACCH implementation outside Snort. We used ACCH simulator for the analysis, which is the simulation method used at the ACCH paper.

5.1 ACCH Simulator

ACCH simulator is a standalone program that implements the ACCH algorithm. The original simulator was written as part of the original article [10] to evaluate ACCH performance. The new simulator was written from scratch for this work, and it is based on ACCH implementation for Snort.

We start the analysis with the new simulator, because it is more compatible with the implementation of ACCH inside Snort.

5.1.1 Simulator Inputs and Outputs

The simulator receives as input the traffic data after it was already reassembled and decompressed along with the pointers meta data. It is also supplied with a list of suspicious patterns. The simulator has to be configured how to construct the DFA, and how to scan the data. We will describe those configurations later in details.

The simulator will output interesting measurements, such as the running time, number of matches detected and number of skipped bytes.

Table 1 summarizes the inputs and outputs of the simulator.

Inputs	Outputs
Uncompressed data	Running time
Pointers data	Number of matches
Patterns	Number of skipped bytes
AC format (5.1.2)	
Scan mode (5.1.3)	

Table 1: Simulator inputs and outputs

5.1.2 AC Format

The DFA can be constructed in two formats:

1. *Full*: The DFA is represented in memory as a full matrix. For every state and possible input byte, there is an entry for the next state.

2. *Sparse*: The DFA is represented in memory such that for every state, there is a list of non-default transitions.

There is a trade-off between the above formats. Full DFA is more expensive in memory usage (explicit entries for all transitions), but the access is much faster (direct access). On the other hand, sparse DFA is cheaper in memory usage (entries only for non-default transitions), but the access is much slower (list traversal).
When memory is limited we will prefer the sparse format, otherwise we will prefer the full format, which is Snort default.

5.1.3 Scan Mode

The fundamental research assumption of the article was that the performance is determined mainly by memory accesses. Therefore, optimizations, that are not related to memory access, were left out. For example, the original simulator scanned each byte separately for both AC and ACCH instead of scanning the whole buffer at once for AC and each segment in ACCH.

Although having the same number of memory accesses, this implementation suffered penalty caused by function call overhead.
In this project we created a new assumption: The performance may be affected not only by memory accesses but also by function calls and branches. To assure our new assumption, the new simulator enables us to compare three scan modes:

1. *ACCH*: The ACCH algorithm designed for Snort.
2. *AC*: The Aho-Corasick algorithm, which scans the whole buffer at once (Snort built-in).
3. *AC-Byte-by-Byte*: An Aho-Corasick algorithm variant, which scans each byte separately (same as the original simulator).

5.2 Simulator Tests

In this section we compare between the performances of ACCH, AC and AC-Byte-by-Byte using the new simulator.

5.2.1 Setup Description

We use a machine with the following specifications for the measurements:

- CPU: Intel i7-2670QM, 2.20GHz
- Cache: L1(i): 32KB, L1(d): 32KB, L2: 256KB, L3: 6144KB
- RAM: 8GB

5.2.2 Data Set

The data set is constructed from two parts - the traffic and the patterns. We use a traffic that contains 2292 html files that take 371MB in uncompressed

form and 58MB in compressed form (compression ratio of 6.4). For the patterns data set we use 9994 patterns taken from Snort rules, each of length greater than 10 bytes.

5.2.3 Simulator Results

We execute the simulator for every combination of *AC format* and *scan mode*. For all scan modes the matches ratio is 0.288%. For ACCH scan mode the simulator skips 64.10% of the bytes.
Tables 2 and 3 describe the results for full and sparse formats respectively. We can see that AC performs better than ACCH when the format is full, while ACCH performs better than AC when the format is sparse. Figure 6 summarizes the results achieved by the new simulator.

Scan Mode	Run Time (sec)	Bit Rate (Mbit/sec)
ACCH	3.6575	810.9887
AC	3.1939	928.7854
AC (Byte-by-Byte)	4.7043	630.5913

Table 2: Comparison of performances for full DFA

Scan Mode	Run Time (sec)	Bit Rate (Mbit/sec)
ACCH	8.6210	344.1335
AC	3.1939	200.9210
AC (Byte-by-Byte)	16.7742	176.8481

Table 3: Comparison of performances for sparse DFA

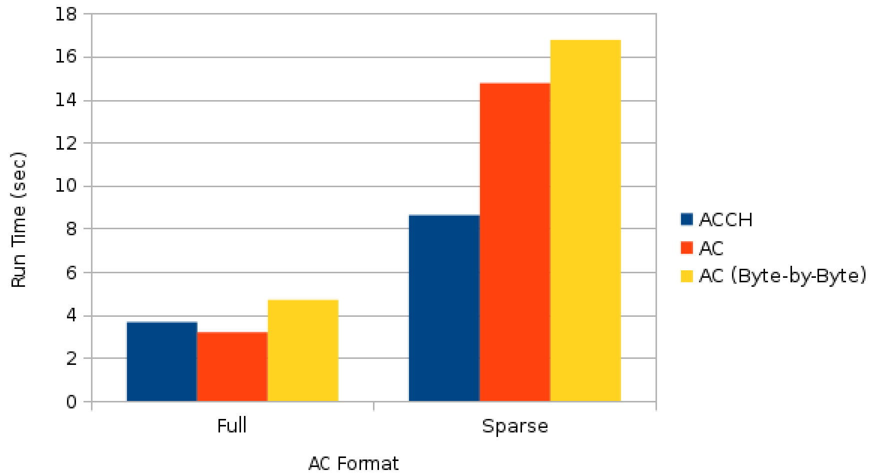

Figure 6: New simulator results summary

5.2.4 Full vs. Sparse

As seen above, when the format of the DFA is full, ACCH does not improve. On the other hand, when the format of the DFA is sparse, ACCH gets an improvement. In the full matrix the access cost is mainly memory, but as later works showed (e.g. [9]), most of the memory accesses are to cache, therefore they are very fast. The sparse transition cost involves link traversal, which is much slower.

5.3 Original Simulator Analysis

Table 4 summarizes the results of the original simulator. We can see that ACCH performs much better than AC for both full and sparse formats. While ACCH results in both original and new simulators are close, AC is much slower in the old simulator.

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

Table 4: Original simulator results

We decided to examine the code of the original simulator in order to spot differences between the simulators, which explain the performance mismatch. The analysis below focuses on full DFA, but it is relevant to sparse DFA as well.

5.3.1 Observation 1: AC is implemented inefficiently

AC implementation in the original simulator tries to mimic the implementation of ACCH. Algorithms 1 and 2 describe the implementation in pseudocode of ACCH and AC respectively.

Algorithm 1 ACCH algorithm in original simulator

```
 1: procedure ACCH(seg_list)
 2:     for seg in seg_list do
 3:         if seg is literal segment then
 4:             for byte in seg do
 5:                 status ← scanAC(byte)
 6:                 add status to statusVec
 7:             end for
 8:         else
 9:             scanPtr(seg)
10:         end if
11:     end for
12: end procedure
```

Algorithm 2 AC algorithm in original simulator

```
1: procedure AC(seg_list)
2:     for seg in seg_list do
3:         for byte in seg do
4:             status ← scanAC(byte)
5:             add status to statusVec
6:         end for
7:     end for
8: end procedure
```

We can improve AC performance with the following changes:

1. The function $scanAC()$ can be defined as inline. This change will improve both AC and ACCH, but the improvement will be more significant for AC, because it calls $scanAC()$ for every byte, while ACCH calls $scanAC()$ only for the bytes it does not skip.

2. AC does not need to traverse the segments, so we can spare the outer loop by simply traversing all the bytes in the uncompressed data buffer, and call $scanAC()$ for every byte.

3. AC does not need to keep track of statuses, so we can remove the statuses management code.

4. The function $scanAC()$ calls to Aho-Corasick search function for one byte. But, Aho-Corasick search function can scan the whole buffer at once. Therefore, the implementation of AC can be replaced with a simple call to the Aho-Corasick search function.

5.3.2 Observation 2: Missing use of circular buffer

The original simulator stored the calculated statuses in a regular buffer of size same as the data buffer. The access to the regular buffer is much faster than the access to the circular buffer, because circular buffer adds extra

branches to the code.

Replacing the regular buffer with a circular buffer will degrade ACCH performance.

5.3.3 Observation 3: ACCH is not optimized for circular buffer

After adding the circular buffer we noticed that the code is not optimized to work with it.

ACCH performance can be improved with the following changes:

1. We can simulate work against regular buffer. During the scan of the current segment, all the statuses will be saved in a regular buffer. After finishing with the scan of the current segment, its statuses will be copied to the circular buffer.

2. The search for minimum MATCH and maximum UNCHECK can be done in one search loop.

3. We can optimize the search for minimum MATCH and the maximum UNCHECK by reading all pointer statuses from the circular buffer into a regular buffer, and finding the minimum MATCH and the maximum UNCHECK in the regular buffer.

4. When ACCH skips byte, it copies the statuses of the skipped bytes from the circular buffer. If we already copied the pointer statuses into a regular buffer, the copy of old statuses can be from the regular buffer.

Table 5 summarizes all the modifications to the original simulator and the impact on the performance on AC and ACCH.

ID	Change	AC (Mbit/sec)	ACCH (Mbit/sec)	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	522.1709	847.6311	38%
3	no status management in AC	676.8416	847.6311	19%
4	no byte-by-byte scanning in plain scan	820.6497	847.6311	3%
5	moved to circular buffer	820.6497	344.8108	-138%
6	optimized segment scan	820.6497	386.2104	-112%
7	min match / max uncheck optimization 1	820.6497	500.0273	-64%
8	min match / max uncheck optimization 2	820.6497	504.7371	-52%
9	optimized old statuses copy	820.6497	698.3914	-17%

Table 5: Original simulator modifications

5.3.4 Analysis Results Summary

Table 6 summarizes results of the modified original simulator and the new simulator. We can see that the performances of both simulator are similar for every AC format.

From the results we can see that ACCH is better than AC when the format is sparse, but not when the format is full.

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%

Table 6: Simulators results

5.4 Circular Buffer Considerations

From the aforementioned steps, it appears that most of the overhead of ACCH is related to the circular buffer management. Removing it will im-

prove the performance significantly. Clearly, we can remove it from the simulators, but can we remove the circular buffer from the ACCH implementation for Snort?
In practice, zlib already uses circular buffer for decompressing the traffic. In theory, we can extend this buffer to store the statuses along with the decompressed data. The problem with this solution is that it is not supported by the current Snort architecture. The decompression is part of the preprocessing stage, while the calculation of statuses is part of the detection stage. Decompressing the traffic during the calculation of the statuses and vice versa will break the modularity of Snort.
What we can do is to use Snort limitation in our favor. Recall that Snort decompresses at most 64KB of the traffic. We can store the statuses of the current buffer in a regular buffer of size 64KB. At the end of the scan we can attach the last 32K statuses the session (using Stream5 API), so they will be available for the scan of the next buffer. This way we reduce the status buffer management to minimum.

6 Snort Tests

6.1 Setup Description

We used the same machine from the simulator tests:

- CPU: Intel i7-2670QM, 2.20GHz
- Cache: L1(i): 32KB, L1(d): 32KB, L2: 256KB, L3: 6144KB
- RAM: 8GB

6.2 Data Set

We generated the data set for Snort from the data set of the simulator. We created a pcap file of 2292 compressed responses (one response for each html file). The total size of the pcap file is 68MB. Remember that the total size of the compressed files was 58MB, so we get extra 10MB for the headers. We used the patterns of the simulator to create rule file with 9994 rules.

6.3 Snort Results

We compare ACCH and AC performances inside Snort for both full and sparse DFA formats. For both algorithms, the matches ratio is 0.315%. ACCH skips 49.29% of the bytes. The matches ratio and the skipping ratio are different from those achieved by the simulator. That is because the DFA is reset (meaning, the current state is set to the initial state) between iterations of the packet loop.
Tables 7 and 8 describe the results for full and sparse formats respectively.
Figures 7 and 8 summarize the results achieved by Snort.
We can see that ACCH and AC perform similarly when the format is full, while ACCH performs better than AC when the format is sparse.

Search Method	Total Run Time (sec)	Total Bit Rate (Mbit/sec)	Scan 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

Table 7: Snort results for full format

Search Method	Total Run Time (sec)	Total Bit Rate (Mbit/sec)	Scan 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

Table 8: Snort results for sparse format

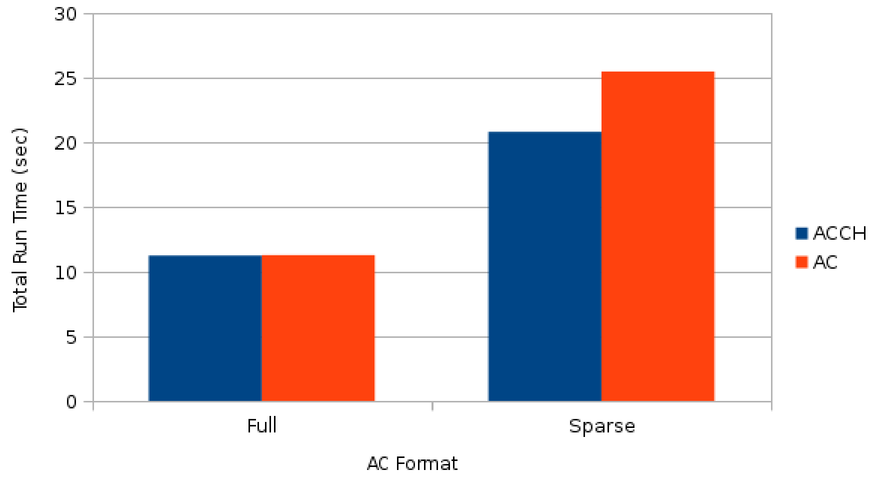

Figure 7: Snort total time results summary

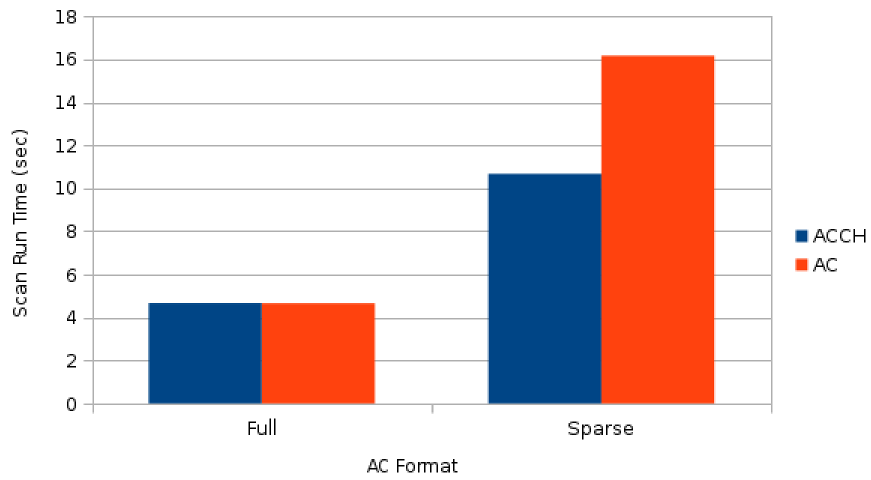

Figure 8: Snort scan time results summary

7 Conclusions and Future Work

In this project we integrated ACCH pattern matching algorithm inside Snort. We had to deal with the challenges of the transition from the theoretical article to the practical implementation. The main challenge was to explain the differences between the expected results and obtained results.
During the analysis of the results we gained a new understanding that the performance is not determined only by memory accesses, but also by branches and function calls. With this insight we get that ACCH improves performance for sparse configuration, but not for full configuration.
Future work can refine the implementation of ACCH inside Snort to maximize its performance. Such refinement may involve re-design of Snort architecture.

References

[1] Deflate compressed data format specification. http://www.ietf.org/rfc/rfc1951.txt.

[2] Gzip file format specification. http://www.ietf.org/rfc/rfc1952.txt.

[3] Snort: The open source IDS. http://www.snort.org.

[4] Sourcefire. http://www.sourcefire.com.

[5] Zlib library. http://www.zlib.net.

[6] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[7] Adeeb Alhomoud, Rashid Munir, Jules Pagna Disso, Irfan Awan, and Abdullah Al-Dhelaan. Performance evaluation study of intrusion detection systems. *Procedia Computer Science*, 5:173–180, 2011.

[8] Michela Becchi, Anat Bremler-Barr, David Hay, Omer Kochba, and Yaron Koral. Accelerating regular expression matching over compressed http.

[9] Anat Bremler-Barr, Yotam Harchol, and David Hay. Space-time tradeoffs in software-based deep packet inspection. In *High Performance Switching and Routing (HPSR), 2011 IEEE 12th International Conference on*, pages 1–8. IEEE, 2011.

[10] Anat Bremler-Barr and Yaron Koral. Accelerating multipattern matching on compressed http traffic. *IEEE/ACM Transactions on Networking (TON)*, 20(3):970–983, 2012.

[11] K Salah and A Kahtani. Improving snort performance under linux. *IET communications*, 3(12):1883–1895, 2009.

[12] Govind Sreekar Shenoy, Jordi Tubella, and Antonio González. Improving the performance efficiency of an ids by exploiting temporal locality in network traffic. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 439–448. IEEE, 2012.

[13] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P Markatos, and Sotiris Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *Recent Advances in Intrusion Detection*, pages 116–134. Springer, 2008.

[14] Yaron Weinsberg, Shimrit Tzur-David, Danny Dolev, and Tal Anker. High performance string matching algorithm for a network intrusion prevention system (nips). In *High Performance Switching and Routing, 2006 Workshop on*, pages 7–pp. IEEE, 2006.

תקציר

סנורט הינו אחד ממערכות ה- IDS הפופולאריות ביותר. סנורט מאפשר לאתר מחרוזות חשודות בפקטות ממגוון רחב מאוד של פרוטוקלים. בין היתר, סנורט מאפשר לסרוק תעבורת http. כיום סנורט מתמודד עם תעבורת http דחוסה בצורה נאיבית. כלומר, בהניתן תעבורת http דחוסה, סנורט פורס את התעבורה, ולאחר מכן מריץ אלגוריתם לזיהוי מחרוזות על התעבורה הפרוסה. כתוצאה מכך סנורט סובל מירידה בביצועים עבור תעבורה http דחוסה.

עבודה זו באה להטמיע את אלגוריתם ה- ACCH בתוך סנורט, ובכך לשפר את ביצועיו עבור תעבורת http דחוסה. אלגוריתם ה- ACCH מנצל את המידע שנאסף בשלב פריסת התעבורה על מנת לדלג על תווים בשלב חיפוש המחרוזות החשודות, ובכך להאיץ את תהליך הסריקה.

יחד עם זאת, התוצאות שהתקבלו לא תאמו את הציפיות. עבור חלק מהקונפיגורציות לא התקבל כלל שיפור, ואילו עבור חלק מהקונפיגורציות השיפור שהתקבל לא היה מספק. על כן, נדרשנו להבין ממה נובע הפער בין התוצאות, כפי שהוצגו במאמר המקורי אודות ACCH, ובין התוצאות שקיבלנו בפועל.

בעבודה זו אנו מציגים ניתוח מפורט של ביצועי ACCH. ניתוח זה מלמד אותנו, שישנם פרמטרים נוספים, שחשיבותם מתגלה כאשר מנסים להוסיף את ACCH לסנורט. המאמר המקורי הניח מודל מופשט, ללא התחשבות בפרמטרים אלו. בעבודה זו אנו מרחיבים מודל זה על ידי שינוי האלגוריתם המקורי, כך שיתמוך בפרמטרים הנוספים. כמו כן, אנו מציעים הנחיות לשיפור עתידי נוסף של האלגוריתם לסריקת תעבורה דחוסה.

המרכז הבינתחומי הרצליה
בית הספר למדעי המחשב על שם אפי ארזי

הטמעת אלגוריתם ACCH במערכת Snort

פרויקט גמר המוגש במילוי חלק מהדרישות לקראת תואר מוסמך במדעי המחשב

על ידי
אדיר גבאי

עבודה זו בוצעה בהנחיית
פרופ. ענת ברמלר-בר
דר. ירון קורל

מרץ 2015