The Andrew and Erna Viterbi Faculty of
ELECTRICAL & COMPUTER ENGINEERING

TECHNION
Israel Institute of Technology

Efficiency Analysis of Traditional Storage Algorithms for Snort Payload Signatures

Project A

submitted on: July 16, 2024

Written By:

Idan Baruch

and

Itai Benyamin

Under the Supervision of:

Alon Rashelbach

Submitting:	Itai Benyamin, Idan Baruch
Student IDs:	208570895, 208688176
Supervisor:	Alon Rashelbach

Contents

List of Figures

List of Tables

List of Abbreviations

DPI Deep Packet Inspection

IoT Internet of Things

IDS Intrusion Detection System

IPS Intrusion Prevention System

NIDS Network Intrusion Detection System

NIPS Network Intrusion Prevention System

PCRE Perl Compatible Regular Expression

Regex Regular Expression

SID Signature Identifier

TCP Transmission Control Protocol

LPM Longest Prefix Match

BF Bloom Filter

IBLT Invertible Bloom Lookup Table

1 Abstract

Quick detection of malicious traffic is a necessity in modern fast networks. To detect a malicious packet, the incoming traffic is matched against a collection of rules for suspicious patterns and signatures. Deep Packet Inspection (DPI) tools use sophisticated regular expressions (regex) for this purpose, ensuring robust traffic classification. However, matching regex patterns can be time-consuming and may introduce a bottleneck to the system. A solution for this challenge is employing an additional early classification system, such as exact term matching.

This work investigates the feasibility of such a preliminary classification system, using only non-ambiguous terms for exact string matching. The project's case-study, Snort's ruleset, is parsed using a generic and fully automated algorithm, extracting the non ambiguous terms called "exact matches". This algorithm's robustness is proven to be hermetic, covering 100% of the ruleset before applying any threshold. Moreover, this project compares the space-efficiency, detection robustness, and false positive rate of two candidates for the suggested system's data structure, namely, Cuckoo hash tables and Aho-Corasick automata. An end-to-end test proves the feasibility of a fast and robust preliminary classifier, showing the Aho-Corasick automaton as the outperforming candidate out of the two data structures. Storage solutions for a secondary data structure show the option of using an IBLT as a trade-off between space-efficiency and robustness accuracy.

2 Introduction

With the ever-increasing volume and speed of data transmission, the potential for malicious activity rises simultaneously. This exponential growth of data transfer between networks () requires rapid virus detection to suffice the standards of data transmission speed. Meeting the demand for quick and safe networks is especially crucial with the growing adoption of Internet of Things (IoT) technologies (**?**). For a network to be secure, it has to protect sensitive data and minimize the impact of cyber threats. The speed aspect of a network is determined by its ability to allow packets to reach their destinations in a timely manner, while maintaining the integrity and availability of network resources and providing uninterrupted services.

These two demands from modern networks, to be secure but also fast, may sometimes come at the expense of one another. To reach security, a robust traffic classification is required. However, for large amount of traffic, a thorough classification could be slow and time consuming. High speed of a network requires quick and rapid classification which might leave the system exposed to threats. Efficient traffic classification plays a pivotal role in optimizing detection processes, ensuring timely, yet robust, identification of threats. The ideal traffic classifier is one that can detect every malicious packet in an instant. That being said, in reality there is a trade-off between detection robustness and time efficiency.

2.1 Deep Packet Inspection

Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) detect suspicious traffic by matching incoming packets against a collection of rules. The rules are based on documented suspicious traffic and known vulnerabilities, and they are the main configuration in which an IDS or an IPS can determine whether or not to take an action against a specific packet. Each rule consists of actions to take when a packet is matched against it, the protocol(s), IP address(es) and/or port(s) of the suspicious packet, as well as the direction(s) of which the traffic is sent. Nevertheless, the rule's payload options include signatures and patterns to be matched against the payload of the suspicious packet. These signatures are the core of a rule, determining whether or not a given packet should be classified as suspicious and raise an alert, or if it should be passed along to its destination.

Most IDS and IPS use Deep Packet Inspection (DPI) methods to ensure robust traffic classification. DPI methods heavily rely on rules with sophisticated regular expressions (regex) in their payload options. Consequently, the overall system prioritizes robustness over speed. Regex offer robust and flexible string matching capabilities using complex pattern definitions, quantifiers and wild cards. However, matching regex patterns can be time-consuming and may introduce a bottleneck to the system. Complex patterns of regex may exhibit exponential worst-case complexity (2) (3) (4), especially when they involve nested quantifiers,

backtracking, or when they are applied to large datasets or long strings.

2.2 Preliminary Classification

A solution for this challenge is to employ an additional early classification system, such as exact term matching (). The idea behind this solution is to use a preliminary classifier that rules out traffic which has low to zero chances to match against the main DPI stage. Only packets that are classified as suspicious in the faster early classifier, are then passed through the slow and robust DPI. Another way to achieve this preliminary classification is by eliminating non-relevant complex regex matching rules that the DPI has to run on a packet. Thus, the DPI only runs a small amount of high-prioritized checks, reducing runtime.
An ideal preliminary classification system should have a much better time complexity than the thorough DPI. Moreover, the early classifier has to be robust in a sense that it does not classify threats as non-threats (zero false negative rate). Another important aspect of the preliminary classifier is to rule out a fair amount of non-threats, or to eliminate non-prioritized rules (low false positive rate). Otherwise, if no significant number of checks are spared by the early classifier, the overall runtime of the traffic classification process stays the same.

2.3 Snort

Snort (5) is an open-source Network Intrusion Detection and Prevention System (NIDS/NIPS), tasked to detect and prevent network-based attacks by analyzing traffic in real-time and matching it against a set of predefined rules and signatures.
Snort's rules (6) are consisted of the **rule headers** and the **rule options**. The Snort rule structure is shown in detail in Figure 2.1.

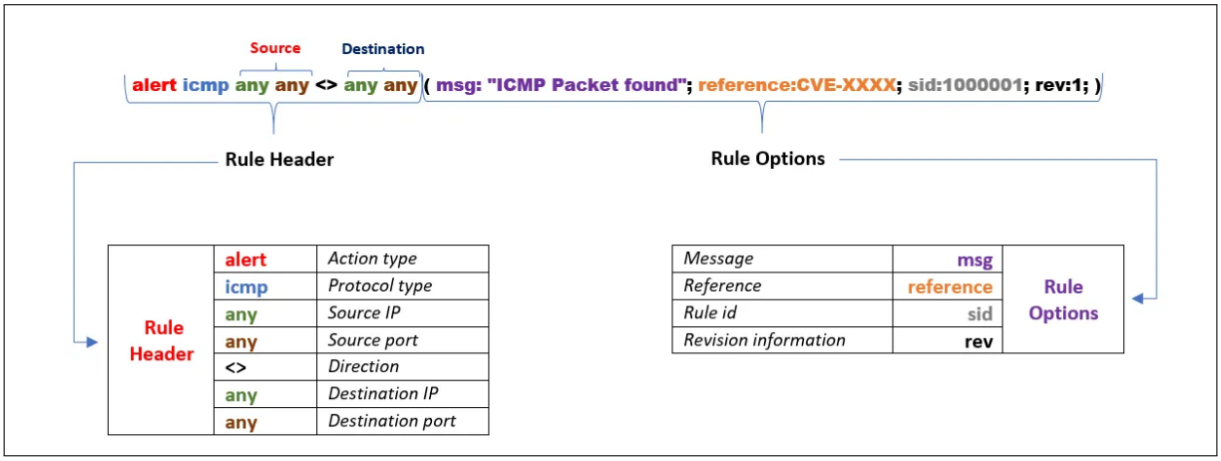

Figure 2.1: Snort rule structure. The rule options are examples out of a large pool of optional fields. Figure taken from (1).

2.3.1 Rule Headers

A rule header tells Snort the general non-payload parameters that a given rule is to matched against in a given packet. The header is divided into the following parts:

- **Rule Actions** state how to handle packets that matched against the rule. Possible rule actions: {alert, block, drop, log, pass}.
- **Protocols** state what type of protocols the rule should check. Supported protocols are: {ip, icmp, tcp, udp}. Snort also allows the usage of services instead of protocols, for example: {ssl, http, smtp, ...}.
- **IP Addresses** state which source and destination IP addresses the rule should apply to. Possible declarations: {any, $EXTERNAL_NET, $HOME_NET, 192.168.0.5, 192.168.1.0/24, ...}.
- **Port Numbers** state which port numbers the rule should match against. For example: {any, 80, 445, ...}.
- **Direction Operators** state the direction of the traffic that the rule should check. Valid direction operators: {->, <>}.

2.3.2 Rule Options

The rule options are the core of a Snort rule, as they determine if a packet is suspicious or harmless. The rule options are unique to each rule, but follow a defined structure. The rule options contain sub-fields of four main categories:

- **General Options** provide additional information for a general rule.
- **Payload Options** include the signatures and patterns to be matched with a packet's payload.
- **Non-Payload Options** include non-payload specific options.
- **Post-Detection Options** determine actions to take after a given packet was matched against the rule.

There are many sub-fields of each category. This project focuses on the following sub-field in the **general** and **payload** options:

- **SID** (Signature Identifier) is a unique identifier given to each and every Snort rule. Snort reserves the values $[0, 10^6 - 1]$ for its official (and community) rules, while rules with $SID \geq 10^6$ can be used locally.

- **Content** are keywords represented by strings or by hex bytes. A given packet's payload is matched against the content keyword. Common content field modifiers are:
  - ***fast pattern*** is a content match from a rule that is the first to be matched against a given packet, in order to determine if the packet should continue the classification process or not.
  - ***no case*** sets the content match to be case-insensitive.
  - ***offset, depth, distance, within*** state the location in a packet's payload or a given buffer to be matched against the content keyword(s).
  - ***[!]*** modifier sets the content keyword(s) as a negative search, i.e., keyword(s) that a suspicious packet should **not** match against.
- **PCRE** (Perl Compatible Regular Expressions) are regex patterns that are to be matched against a given packet's payload at the DPI stage of the process. Common PCRE modifiers are:
  - ***/i*** sets the PCRE match to be case-insensitive.
  - ***/R*** starts the regex search from the **end** of the last match.
  - ***[!]*** sets the PCRE as a negative search, i.e., patterns that a suspicious packet should **not** match against.

In Figure 2.2 highlights from an example of a Snort rule are presented, taken from Snort's community ruleset.

```
alert tcp $HOME_NET 6969 -> $EXTERNAL_NET any
(msg:"MALWARE-BACKDOOR GateCrasher";
content:"GateCrasher",depth 11,nocase;
content:"Server",distance 0,nocase;
content:"On-Line ... ",distance 0,nocase;
pcre:"/^GateCrasher\s+v\d+\x2E\d+\x2C\s+Server\s
+On-Line\x2E\x2E\x2E/ims";
sid:147; rev:12;)
```

Figure 2.2: An example of a Snort rule, taken from Snort's community rules.

The rule header, shown in **red**, sets the rule to alert in case of a match. Only traffic using TCP protocol from the IP addresses of the local network, specifically from port 6969, and directed to any port in IP addresses from the external network will be targeted for further payload investigation.
The rule options are enclosed in **parenthesis**. The **purple** section includes additional information that should be displayed once the rule is triggered. The **blue** section contains

the content keyword strings to be matched against a given packet's payload, all set as case-insensitive. This rule has no fast patterns. The **green** section shows the PCRE pattern of the rule, also set to case-insensitive. Packets that proceed to the DPI stage of the inspection process are matched against this regex pattern, with suspicious packets being those whose payload matched the PCRE signature stated in the rule.

2.3.3 Functionality

A packet going through Snort's inspection is first checked against the header of each rule. A packet that is matched with a given rule header, proceeds to be checked against the rule's payload options. A packet's payload should match with the fast pattern of a rule (if exists) to proceed the inspection. Afterwards, the content keywords are checked, and only then the PCRE patterns. This is due to the fact that regular expressions matching is much more time-costly in comparison to the exact matching with the fast patterns or the content keywords. Snort recommends that every rule that uses PCRE should also include at least one content match (7).

2.3.4 Challenges

Snort offers the fast patterns, as well as the Hyperscan (8), as a solution to compensate for the performance gap between exact matching and PCRE matching. However, Snort's fast patterns as a preliminary classification do not take several factors into account.
Firstly, only one (or less) out of the entire content keywords, that could be extracted out of a given PCRE signature, is being utilized as the main fast pattern of a rule. Secondly, every fast pattern is assigned to one rule only, not taking into account that different rules might have common keywords. Lastly, not storing all keywords in the same data structure lacks space efficiency and does not take advantage of large scale matching hardware, such as the NeuroLPM (9).
Based on these challenges, this project investigates the feasibility of another traffic preliminary classification approach, using Snort's rule-sets as a case-study. It aims to overcome the listed challenges. This is done by firstly presenting a new parsing approach to extract keywords (in this work referred to as **exact matches**) from a PCRE, as well as a threshold-weighting system to the patterns which matched the suspicious packet's payload. Additionally, instead of assigning keywords to each rule, this work suggests the assignment of rules to common keywords. This may save space of the data structure, as well as help facilitating the usage of large scale matching hardware.

3 Methods

3.1 General Assumptions on Data

This project mainly focuses on feasibility, storage efficiency of traditional string matching algorithms, and rule payload options. Thus, the rule header matching is not included in the scope of this project, and it is assumed that all packets are relevant for the payload inspection - regardless of their header information. Moreover, for the sake of simplification, the following assumptions on Snort's rules payload options are made:

- A given rule's identifier is set to be the SID (Signature Identifier) used by Snort.
- The valid range of SIDs is set to be Snort reserved values of $[0, 10^6 - 1]$, ignoring locally defined rules.
- Only the 'PCRE' and 'content' fields are taken into consideration as the rule's signature patterns.
- All patterns are assumed to be case insensitive.
- *offset, depth, distance, within* and similar modifiers are ignored.
- Any negation options using the '[!]' symbol (i.e. 'content!:' or 'pcre!:') are ignored.
- The content keywords are taken as is.
- PCRE signatures ending with the '/R' modifier (matching relative to the end of the last matched pattern) are ignored.
- Other rule options that were not mentioned so far are out of the scope of this project and are not discussed.

3.2 Structural Design for Rules and Keywords

As mentioned in Section 2.3.4, this project offers another approach for the early classification of traffic, before applying DPI methods. Instead of having a fast pattern or content keywords as the preliminary classifier for each rule by itself, this work offers a different approach. Taking advantage of large scale matching hardware, this project strives to have the early classifier match given packets with **all** non-ambiguous strings extracted from the payload options of the ruleset, to determine which rules should be checked in the DPI stage.
This can be done by creating a joint data structure for these non-ambiguous strings (later referred to as **exact matches** or **sub-strings**), **each points to a list of rules it can trigger**. This is different from the ruleset of Snort, where each rule points to a list of

keywords and patterns which can trigger the rule they are assigned to. The two different approaches are compared in Table 3.1.

Table 3.1: Differences between Snort ruleset structure and this project's.

Snort		
SID	→	**Keywords**
1	→	{'get','info'}
2	→	{'get'}
3	→	{'info'}

This Project		
Keyword	→	**SIDs**
'get'	→	{1, 2}
'info'	→	{1, 3}

3.3 Threshold-Based Classification

Utilizing all the keywords extracted from all rules can lead to a very high rate of false positive classifications at the preliminary stage. A false positive classification (classifying a non-threat as a threat) can be caused by very short or generic tokens that can be matched easily - triggering all the SIDs assigned to them. A high false positive rate can drastically harm the functionality of the early classifier, because it keeps the scale of rules that are checked on the DPI stage the same - not addressing the bottleneck challenge described in Section 2.1.
To lower the false positive rate, while maintaining the minimum false negative rate, this project investigates a **threshold-based classification**. To achieve this, two thresholds can be introduced, in two different parts of the early classification:

- **Insertion Threshold** is a **constant** threshold set on the length of the keywords that are inserted into the data structure. Only keywords longer than the insertion threshold are accepted into the data structure.
- **Trigger Threshold** is a **dynamic** threshold that can be set at the decision stage of the preliminary classification. Given a post-matching histogram of all SIDs and the amount each was triggered (later referred to as **SID-hits histogram**), a threshold can set the bar from which the SIDs are checked again by the DPI.

3.3.1 Insertion Threshold

The insertion threshold filters short keywords, lowering the false positive rate, as short keywords tend to be too generic and non-specific - triggering many SIDs simultaneously. Moreover, using the insertion threshold is more space efficient, as less keywords are inserted to the data structure. Despite its advantages, the insertion threshold can damage the robustness of the early classifier. This is due to the fact that filtering out keywords may also filter out

SIDs that were **exclusively represented** by these keywords (for example: $SID = 2$ in Table 3.2).
Therefore, in order to maintain robustness, rules that are not represented in the preliminary classification's data structure, must be always checked in the DPI stage (making them false positives by default). This insures minimal false negative rate, but may increase the false positive rate - making this type of threshold a double-edged sword.

Table 3.2: Possible effects of insertion threshold on SIDs representation.

No Threshold		
Keyword	→	**SIDs**
'get'	→	{1, 2}
'info'	→	{1, 3}

Insertion Threshold = 4		
Keyword	→	**SIDs**
'info'	→	{1, 3}

3.3.2 Trigger Threshold

The trigger threshold, illustrated in Figure 3.1, can be determined dynamically for each run of the classifier (on a given packet or a group of packets).

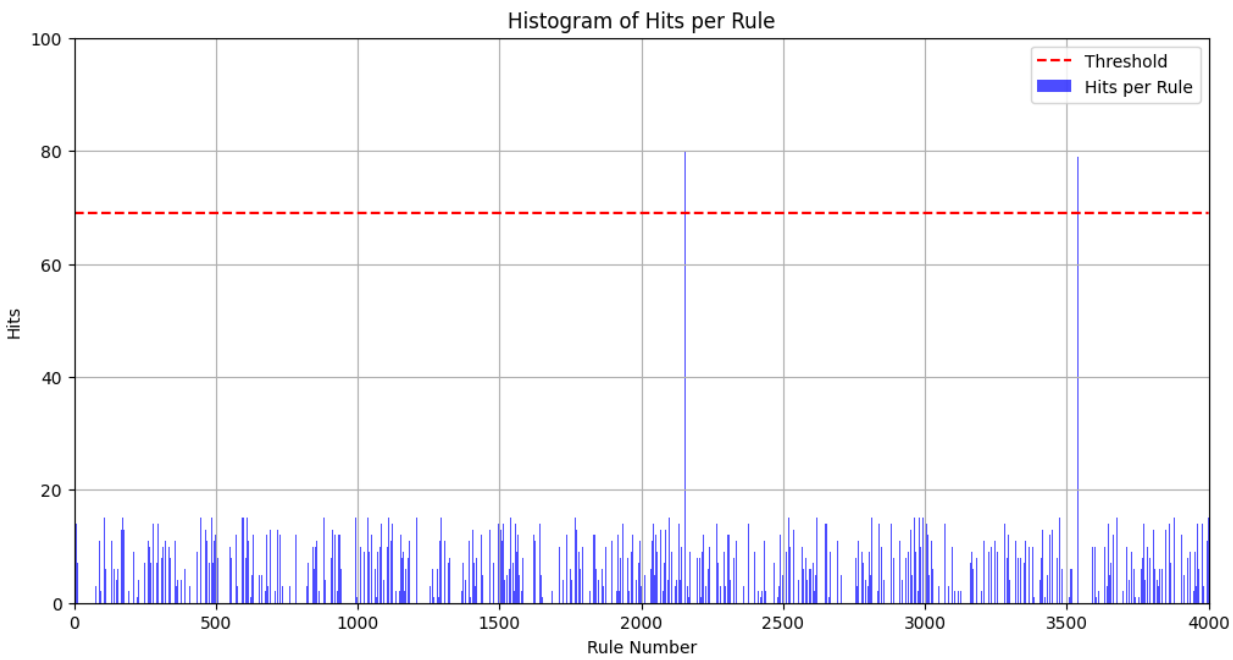

Figure 3.1: An illustration of an SID-hits histogram, generated using mock data. The blue bars indicate the "hits" - the total number of times the rule carrying this SID number was triggered in the current check. The red dashed horizontal line indicates the threshold. SIDs that had more hits than the threshold are checked again at the DPI stage.

This offers a trade-off between lowering false positive rate and risking harming the robustness of the classifier. Setting a high threshold filters irrelevant SIDs that had low amount of hits,

whereas setting the threshold too high can filter out relevant rules with sufficient amount of hits to raise suspicion. Unlike the insertion threshold, however, SIDs that were filtered out due to the trigger threshold cannot be restored - making the determination of the trigger threshold a crucial factor of the early classifier.

3.4 Exact Matches Extraction

As discussed in Section 2.1, one of the main goals of the preliminary classification is to enhance the performances of the overall classification system by speeding up slow processes, and relieve the bottleneck that the DPI may introduce. This project attempts to do so by only using **exact matches**.
Exact matches are deterministic keywords, extracted from the payload options of a rule, and can be matched using an exact string matching algorithm. These non-ambiguous strings can potentially make the matching process in the early classification much faster than the DPI's regex matching. Moreover, the potential speed-up increases when taking into account the possible utilization of large scale matching hardware, such as the NeuroLPM mentioned in Section 2.3.4.
For the task of extraction of exact matches out of Snort's ruleset, a fully-automated and generic algorithm was designed. The algorithm (Algorithm 1) takes as an input **any** ruleset of Snort as a *.zip* file, then proceeds to parse all *.rules* files it contains to generate a dictionary of exact matches and for each exact match - a list of the SIDs of the rules it represents.
The algorithm relies on the assumptions made in Section 3.1, taking content keywords as exact matches. For the PCRE payload option, however, the algorithm unwraps it by applying a set of complex regex patterns, carefully removing ambiguous tokens, resulting in a list of the extracted exact matches. Any deterministic keywords that were separated by ambiguous patterns in the PCRE are expressed as **different exact matches** in the exact matches dictionary. An illustration of this process can be shown in the example given in Figure 3.2.
Snort content keywords are always a sub-set of the exact matches extracted by running Algorithm 1. This poses a major advantage of this work's exact matches over the content keywords suggested by Snort. The reason behind the exact matches being more generic than Snort's content keywords, is that the algorithm not only extracts the trivial full-chunks of strings from a PCRE, but also does the following:

- **OR Expressions Expansion** the algorithm expands simple ambiguous patterns using the OR operator '|', into non-ambiguous ones. An example for this can be seen in Figure 3.2 where PCRE ambiguous pattern *'(info|app)'* is parsed, resulting in *['info', 'app']*.

- **Explicit Repetition Expansion** the algorithm expands simple ambiguous patterns using the explicit repetition operator '$c\{\cdot,\cdot\}$' (where c is a non-ambiguous character) into the minimal repetition accepted of the character c. Namely:

Algorithm 1 Parsing Snort Ruleset

```
Require: Snort ruleset as a .zip file
Ensure: Dictionary of exact matches and associated SIDs
Initialize an empty dictionary exact_matches
Extract all files from the .zip archive
for all file file in extracted files do
  if file ends with .rules then
    Open file for reading
    while not end of file do
      Read next rule line line
      Parse line to extract the content, pcre, and sid fields
      for all keyword keyword in content do
        if keyword in exact_matches then
          Append sid to exact_matches[keyword]
        else
          Initialize exact_matches[keyword] to list containing sid
        end if
      end for
      for all regex pattern pattern in pcre do
        Expand non-ambiguous regex in pattern into their respective characters
        Apply a set of complex regex on pattern to extract non-ambiguous keywords
        for all keyword keyword in keywords do
          if keyword in exact_matches then
            Append sid to exact_matches[keyword]
          else
            Initialize exact_matches[keyword] to list containing sid
          end if
        end for
      end for
    end while
    Close file
  end if
end for
return exact_matches
```

  - For the exactly $n \in \mathbb{N}$ repetitions operator, $c\{n\}$, the non-ambiguous expression of n-times the letter c is used as replacement.
  - For the $n \in \mathbb{N}$ to $n \leq m \in \mathbb{N}$ repetitions or the $n \in \mathbb{N}$ or more repetitions operators, $c\{n, m\}$ or $c\{n, \}$, the non-ambiguous expression of n-times the letter c is used as replacement, choosing the minimal required matching length.

- **Single Characters Extraction** the algorithm also extracts very short sequences of small lengths up to a single character. While seeming redundant, this can actually boost relevant SIDs hits on the histogram illustrated in Figure 3.1, setting the difference between relevant and irrelevant rules.

It is important to note that all extracted exact matches are parts of a rule's payload signature that must match in order for the full PCRE pattern to match. This means that **if an exact match did not match with a packet's payload, the PCRE which it was extracted of will not match the packet's payload either**. Under the assumption that the suspicious signature was not cut into different packets, the exact matches pose a suitable parameter for an early classifier. If this assumption is not met, the trigger threshold (introduced in Section 3.3.2) might still be able to detect a suspicious signature that was cut into several packets. Moreover, the time complexity of this algorithm does not affect the performance of the preliminary classifier. This is due to the fact that the data structure requires an update only when a major change to the ruleset is made. This implies that this parsing needs to be done only rarely and its time complexity can be ignored. Therefore, the parser is implemented in Python, as real-time C++ implementations are only needed for the preliminary classifier itself. The parser's results are then saved in a JSON file, and are read later from the C++ code using the *nlohmann* :: *json* (10) library.

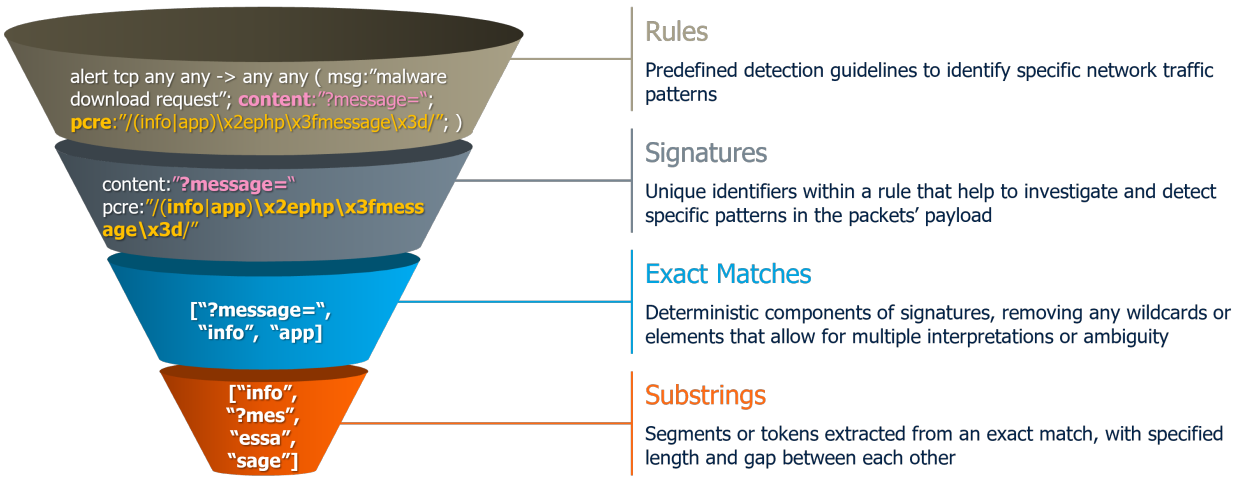

Figure 3.2: An illustration of the parsing process of the exact matches extraction, using a real example and some of its exact matched extracted by Algorithm 1.

3.5 Data Structures

To check the feasibility of the early classifier, the exact matches parsed from the ruleset are stored in a data structure. The early classification stage tries to match inspected traffic against the exact matches stored in the data structure. This is performed by the lookup (search) function of the chosen data structure. Therefore, the main requirement of the data structure is to allow fast lookup.
In order to realize the preliminary classifier suggested in this work, a secondary structure for the list of SIDs should be implemented. As shown in Table 3.1, each entry in the suggested data structure consists of a key, which is the exact match, and a value, which is the pointer to the secondary structure, containing the SIDs triggered by this exact match. The secondary structure for the SIDs requires additional space, which is addressed later on.
The data structures are implemented in C++, to match the requirement of real-time and fast execution of the preliminary classification.

3.5.1 Cuckoo Hash Table

Cuckoo hash table (11) is a suitable candidate for a data structure. A cuckoo hash table is a data structure that uses multiple hash functions to resolve collisions. From this stems its feasibility for storing the exact matches, as its techniques results in a constant-time worst-case lookup of $O(1)$. Even though the relative high-time worst-case insertion time of $O(n)$, in this work the insertion process happens only once, in the beginning of the data structure set-up. Therefore, this insertion-lookup time trade-off fits for the wanted application.
The cuckoo hash table requires careful management of load factors and table size to avoid performance degradation, and it consumes more space than a regular hash table implemented using simple chaining, due to maintaining multiple tables. Thus, cuckoo hashing may be more space-costly.
Another issue that stems from using a hash table is that entries must have a fixed size. This means that the exact matches of varying lengths cannot be inserted to this data structure as is. To resolve this issue, **sub-strings** of fixed length L and with parsing gap of G are extracted from the exact matches, as seen in Figure 3.2.
This project uses the *libcuckoo* (12) library for testing, going over four different variations of the data structure with changing sub-strings' lengths and gaps:

- Sub-strings of length $L = 4$ and gap $G = 1$
- Sub-strings of length $L = 4$ and gap $G = 2$
- Sub-strings of length $L = 8$ and gap $G = 1$
- Sub-strings of length $L = 8$ and gap $G = 2$

The choice for the L, G was made based on the rule coverage results, which are discussed later on.

3.5.2 Aho-Corasick Automaton

The Aho-Corasick algorithm (13) is a string-matching technique that efficiently searches for multiple patterns simultaneously in a given text, by constructing an automaton from the patterns. This is in contrast to the cuckoo hash table, where only one match can be found at a time, and the parsing of the text is necessary to find multiple patterns. The lookup time of the Aho-Corasick automaton is $O(n + k)$, where n is the length of the text searched, and k is the number of patterns found. While it may seem as a much larger order in comparison to the one of the cuckoo hash table lookup, this complexity is calculated over lookup in an entire text and of multiple patterns. Achieving the same using the cuckoo hash table would require parsing the text into m smaller sub-strings and matching each sub-string in $O(1)$, resulting in an effective search time of $O(m)$.
Aho-Corasick allows the insertion of strings in varying lengths, making the insertion of exact matches into this data structure rather easy. Moreover, there is no need to set an insertion threshold, since exact matches of any length can be stored. This makes the Aho-Corasick a promising candidate for a data structure.
This work uses the *cjgdev* :: *aho_corasick* (14) library for testing. To fulfill this project's needs, additional functionalities were implemented to the mentioned library as a pull-request (15).

3.5.3 Theoretical Size Minimization

To fully utilize the benefits of large scale matching hardware, as discussed in Section 2.3.4, the data structure of the preliminary classifier should consume minimal memory. Non space efficient designs would not be beneficial to use on small hardware applications. For that, theoretical methods to minimize the space consumption of the data structure, are suggested. Usually, small hardware applications do not use 64 bits architectures, meaning that for this work a 32 bits architecture can be assumed. This assumption reduces the calculated space of pointers, from 8 Bytes to 4 Bytes.
The Aho-Corasick could be theoretically implemented using a GOTO table, where every transition edge in the automaton is converted into a space-efficient cell in the GOTO table. This can be seen in Table 3.3, presenting the theoretical sizes of potential transitions as cells in the GOTO table. Based on this table, all following calculations of the Aho-Corasick space usage are done by the theoretical size: $num\_of\_transition\_edges \cdot (3 + 1 + 3 + 4)[Bytes]$.

Table 3.3: Required storage for each transition in the Aho-Corasick GOTO table.

State Number	Char	Next State	Pointer to SIDs list
3 Bytes	**1 Byte**	**3 Bytes**	**4 Bytes**

3.5.4 Secondary Structure

The secondary structure contains a list of rules that are triggered by a certain exact match. This can be implemented using the naive solution of a linked list, or using a more sophisticated storage approach such as the Invertible Bloom Lookup Table (IBLT) (16).
IBLT is a probabilistic data structure that allows efficient set reconciliation and element recovery. We can use IBLT in the storing of SIDs which associate with every exact match. This way we can save memory, however we risk with failure when retrieving relevant SIDs from the IBLT.
For the theoretical calculations of the additional space needed, Py-IBLT (17) library was used for testing.

3.6 End to End Testing

To test the feasibility of the preliminary classifier, as well as the two different data structures suggested, a series of ten tests are generated to compare the robustness, false positive rate, and storage-efficiency of each candidate. Each test is generated out of a specific PCRE pattern, creating a payload with a suspicious signature designed to match **only** against this PCRE's SID. This SID is referred to as the **validated** SID of the test. SIDs that are triggered from the test payload are either the actual rule ($if sid == validated\_sid$) or false positives ($if sid\ != validated\_sid$). The false positive rate is calculated as the number of false positive SIDs out of all SIDs triggered in this test. It is calculated as a mean on all tests. The success rate is defined by a mean on whether or not the validated SID was detected by the preliminary classifier. To lower the false positive rate a trigger threshold (explained in Section 3.3.2) is applied during the tests. The threshold filters rules below a certain "hit" parameter, where a hit in the cuckoo hash is a match against a sub-string, and a hit in the Aho-Corasick automaton is the length of the pattern matched. The threshold is determined per test, evaluating the highest hit-rate.

4 Results and Discussion

4.1 Rule Coverage with Exact Matches

In order to implement the early classifier suggested in this work, the non-ambiguous exact matches are extracted from Snort's community ruleset. In Figure 4.1 the results of applying different values of insertion threshold, introduced in Section 3.3.1, are shown.

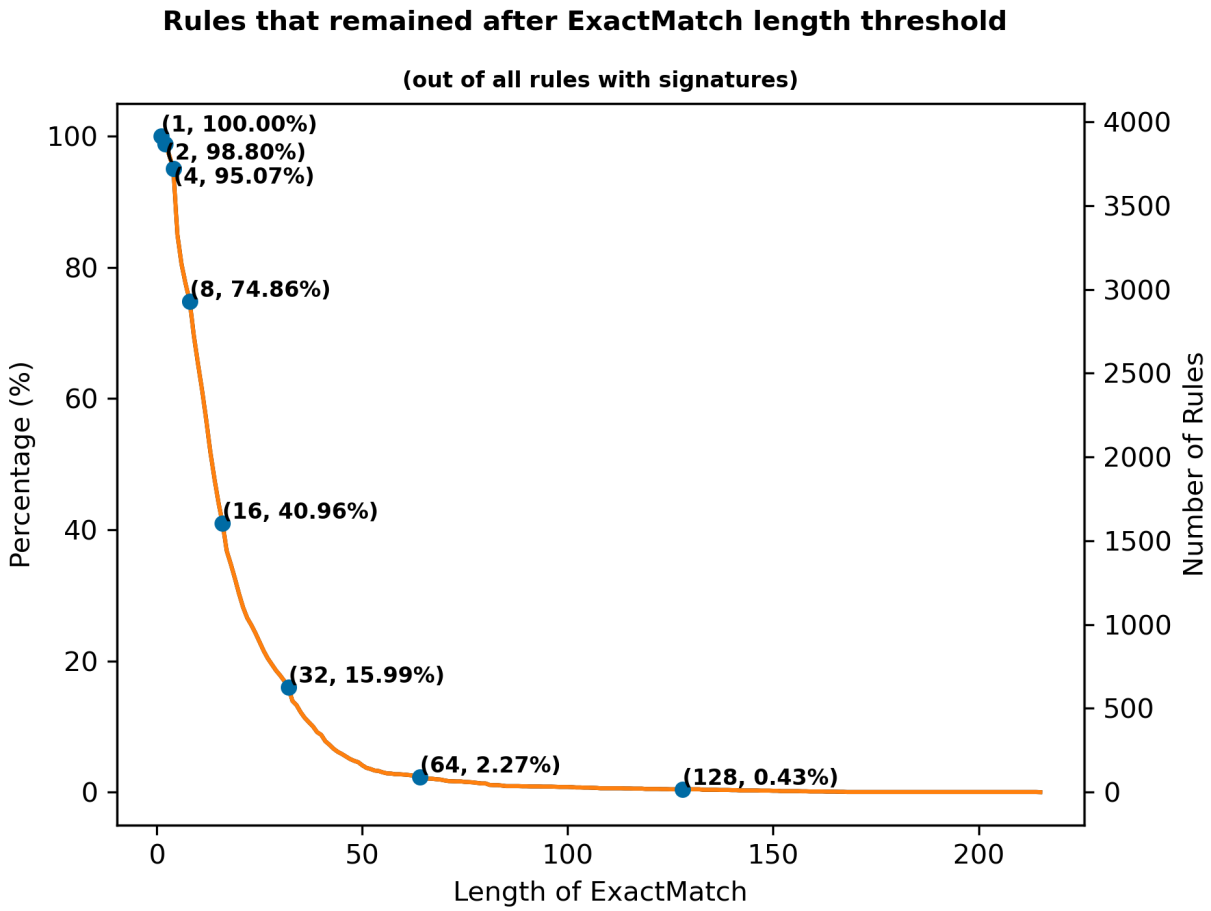

Figure 4.1: Percentage of rules remained as a function of the applied insertion threshold on the exact matches length in characters.

The graph starts at the value of the minimal insertion threshold of 1. The minimal threshold demands that the length of the exact matches have to be at least one character to be inserted to the classifier's data structure ($len(exact\_match) \geq 1$). For this minimal threshold, all exact matched extracted are inserted to the data structure. It can be seen that this exhibits a 100% coverage of the rules in the DPI's ruleset. That means that Algorithm 1 manages to extract non-ambiguous strings from **every** PCRE signature of the ruleset.
Increasing the length threshold, results in an exponential drop in rules coverage. This is due to the fact that less exact matches are inserted into the data structure, representing less amount of the overall rules, as presented in Table 3.2.

For threshold lengths above 8 the data structure is not covering for above 25% of the rules. This means these 25% of rules, which is about 1,000 rules, has to be examined by the time-consuming DPI stage. Therefore, it is deducted that the range of valid insert thresholds should be between 1 and 8. Exceeding this range makes the early classifier redundant, as the reduction offered by the preliminary classification is not high enough.

4.2 End to End Results

4.2.1 Individual Tests Results

After performing the end-to-end tests, the results of the individual tests for each data structure candidate can be discussed.
Figure 4.2 shows the amount of hits triggered for each SID that matched the performed test, using cuckoo hash table.

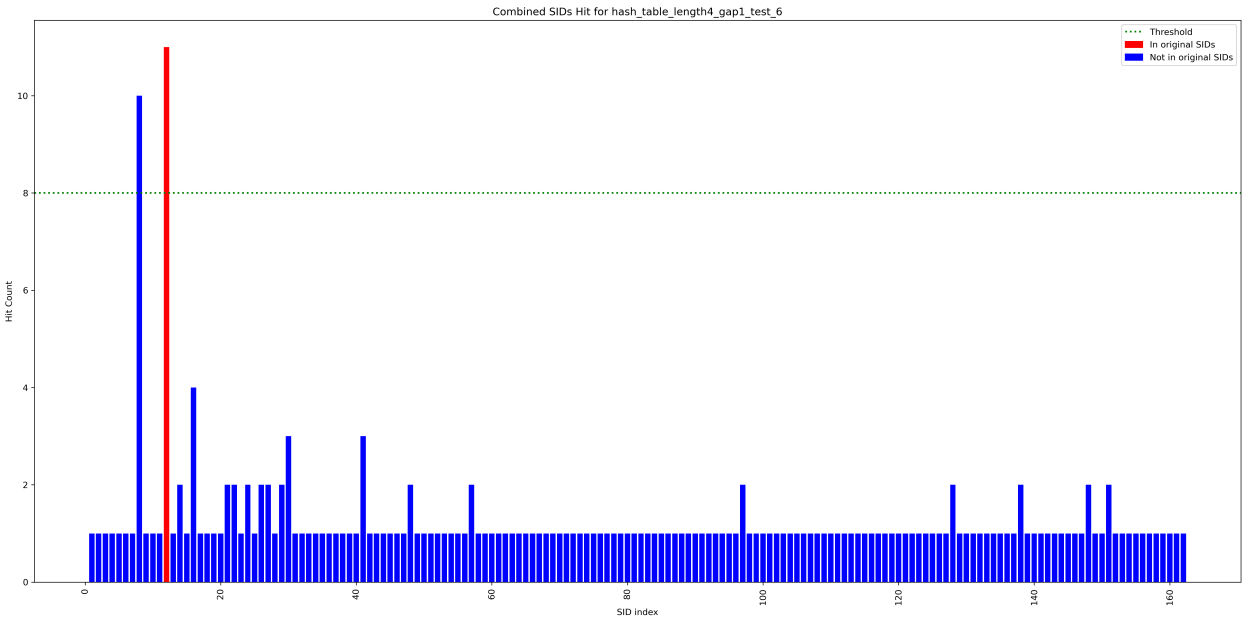

Figure 4.2: Test result on cuckoo hash table, using sub-strings of $L = 4$, $G = 1$, and trigger threshold of 8.

The x-axis indicates the normalized indices of SIDs which matched to the suspicious signature test. The height of the bar states the amount of triggers of the corresponding SID. The red bar indicates the validated SID, which should match against the test. The blue bars are other SIDs, which should not be passed to the DPI stage by the early classifier.
The graph shows vividly that the validated SID, in red, exhibits the highest number of hits. Moreover, there are many other SIDs, that were also triggered, but at much less volume except one outlier. The trigger threshold then can be applied. Setting it to 80% of the highest hit-rate, most non-validated SIDs can be filtered, and only the wanted validated SID, as well as one outlier, remain. This results in a successful test, detecting the validated

SID. However, another SID surpasses the trigger threshold, resulting a false positive rate greater than zero.
The same test was applied on a Aho-Corasick automaton. Figure 4.3 shows the amount of hits triggered for each SID that matched the performed test.

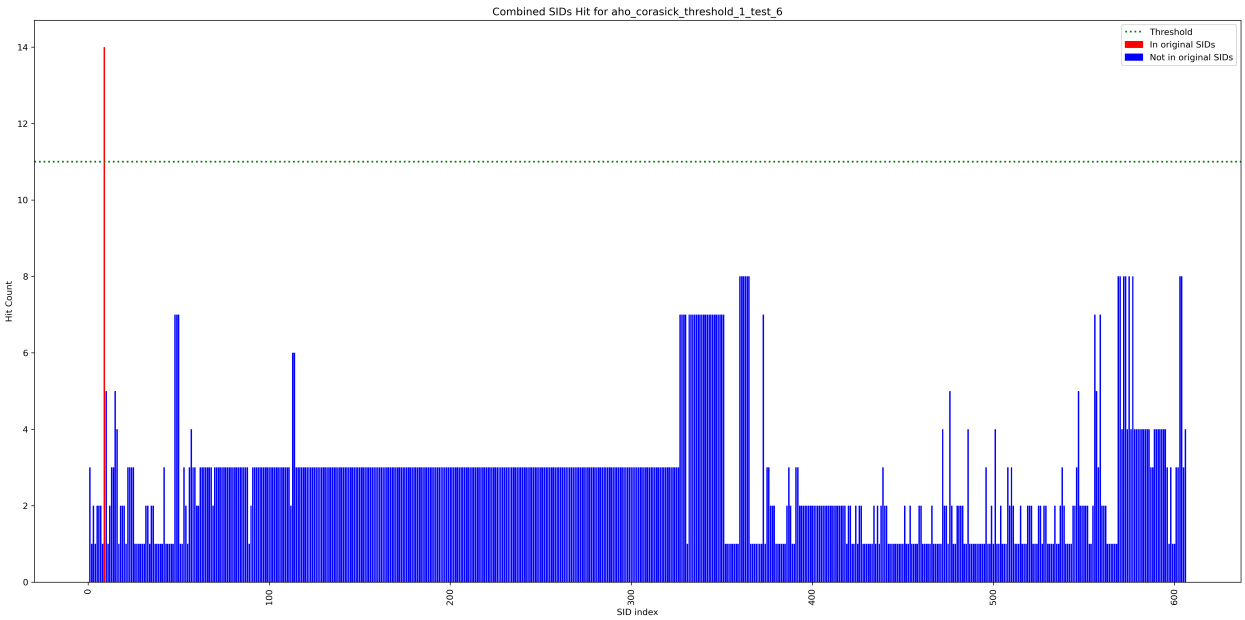

Figure 4.3: Test result on Aho-Corasick automaton, using insert threshold of 1 and trigger threshold of 11.

The graph shows again that the validated SID is the highest peak. The noise margin increased as the total number of SIDs triggered is higher in this example. However, none of these SIDs surpassed the trigger threshold. This is the best possible outcome, making this test have a 100% success rate and a 0% false positive rate.
Figure 4.4 maps a number of different tests that were applied on the cuckoo hash table. This graph illustrates the classifier's ability to detect multiple validated-SIDs at once. However, as the height of the detected validated SIDs varies, the trigger threshold has to be chosen accordingly.

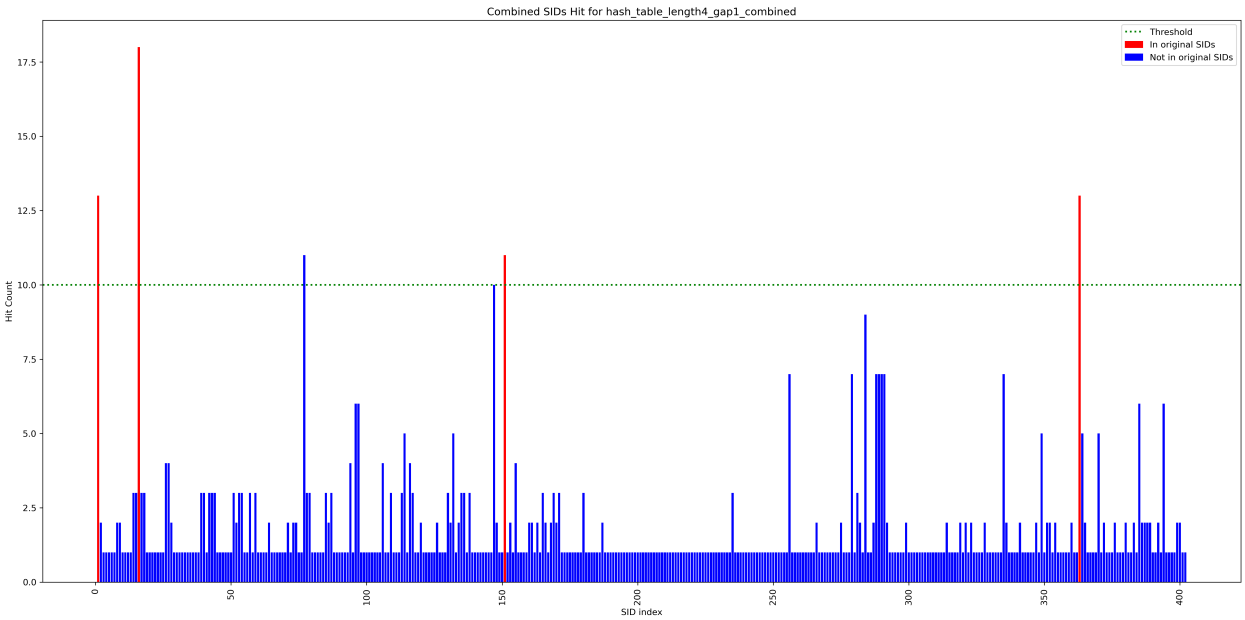

Figure 4.4: Combined test results of four different individual tests on the cuckoo hash table, using sub-strings of $L = 4$, $G = 1$, and trigger threshold of 8.

4.2.2 Mean Test Results

Many individual tests on each data structure can be evaluated to compare the performances of the candidates and determine the most promising one. In Figure 4.5 different variations of the cuckoo hash table, using four different sub-string set-ups, are evaluated.

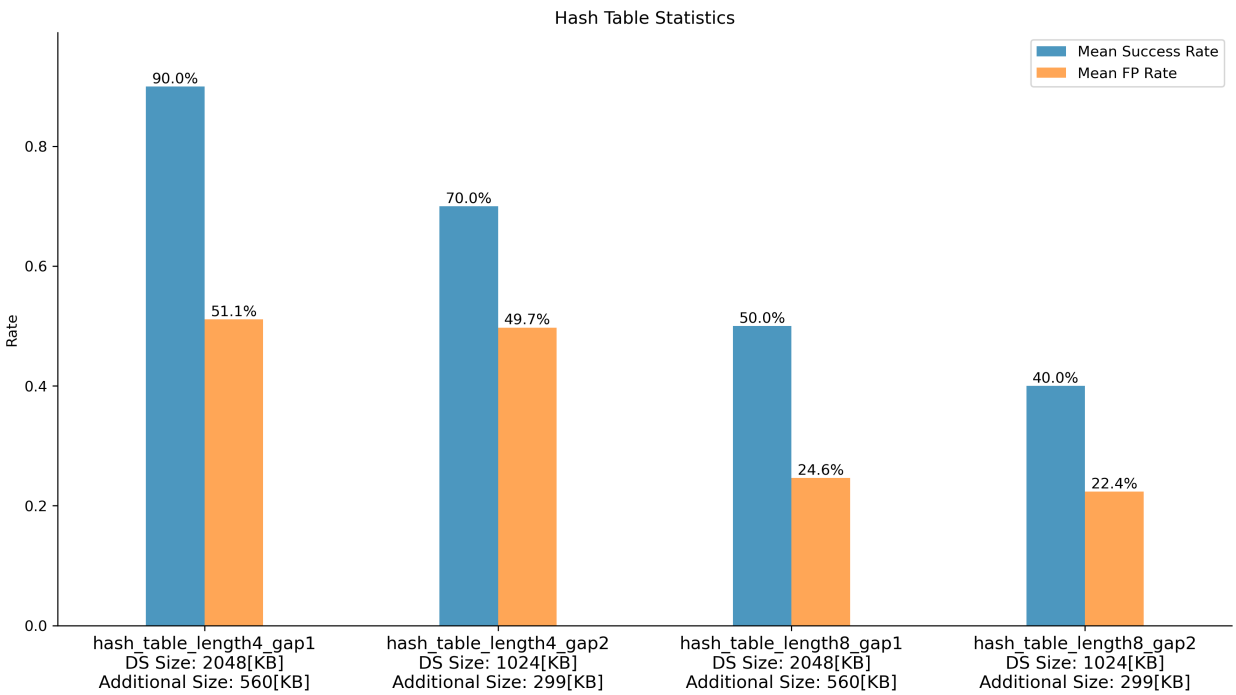

Figure 4.5: Mean of individual test results of different variations of the cuckoo hash table.

In each position of the x-axis is a different variation of the cuckoo hash table. The two bins for each position state, in blue, the mean success rate over the entire test series, and

in orange, the mean false positive rate. For rising L or G both the success rate and the false positive rate decrease. The goal is to find the best variation, in which the success rate is the highest, the false positive rate is the lowest, and the data structure size is the smallest. However, not all of these parameters are equally important. The success rate affects the robustness of the early classifier and is weighted the most with 75% of the total score given. The false positive rate affects the speed of the system, since it determines how many irrelevant matches falsely proceed to the time-consuming DPI stage. To minimize the false positive rate, a 25% weight is given to the complement of it.
The best variation of the cuckoo hash table, determined by the mentioned calculation logic, is the one of the $L = 4$, $G = 1$ sub-strings.
Similar to the previous evaluation of the cuckoo hash table, in Figure 4.6 different variations of the Aho-Corasick, using insert thresholds in range $[1, 8]$, are evaluated.

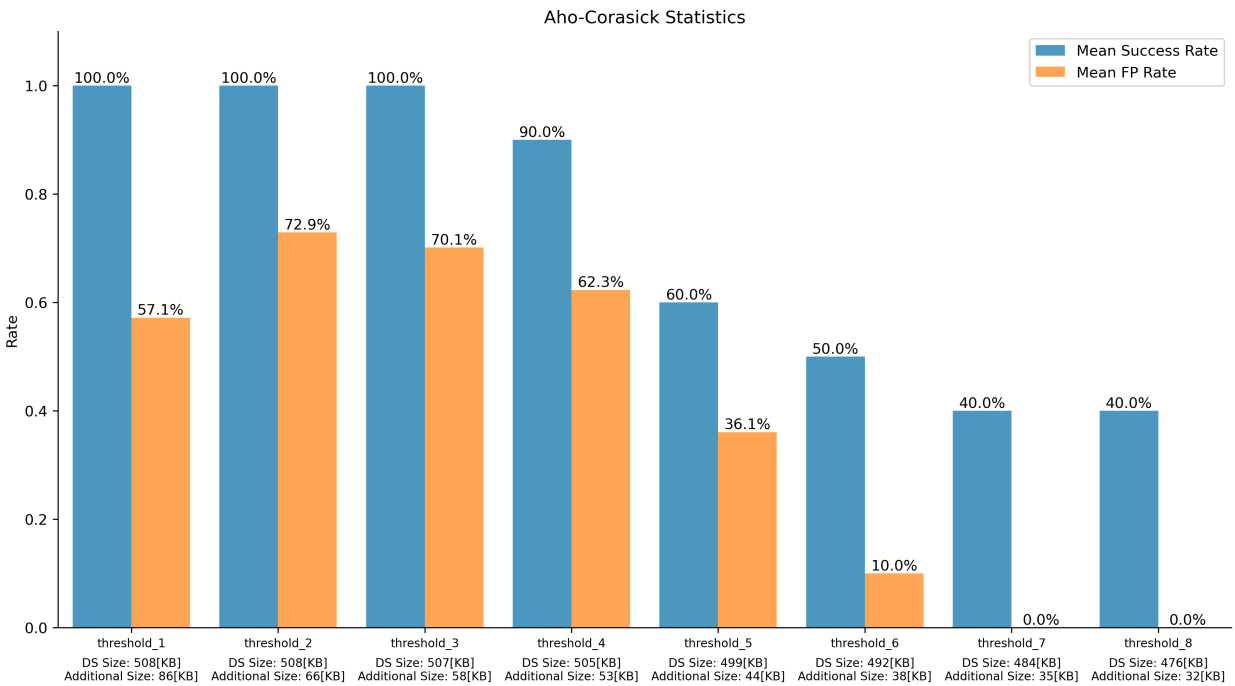

Figure 4.6: Mean of individual test results of different variations of the Aho-Corasick automaton.

Here, eight different variations are shown, each time setting a different insertion threshold before constructing the data structure. The graph shows the same trend of decreasing the insertion threshold causes the both values to decrease as well. There is only one exception, the false positive rate of the first variation with insertion threshold $= 1$, does not follow the observed trend. This behaviour is induced by the trigger threshold, and the exact dependencies should be investigated further. In general, the best evaluation score is given to this insertion threshold of one.
Comparing the best variations of both systems results in Figure 4.7.
In this comparison between the data structures, the weighted score of the Aho-Corasick data structure is higher. Therefore, the Aho-Corasick can be depicted as the preferred candidate

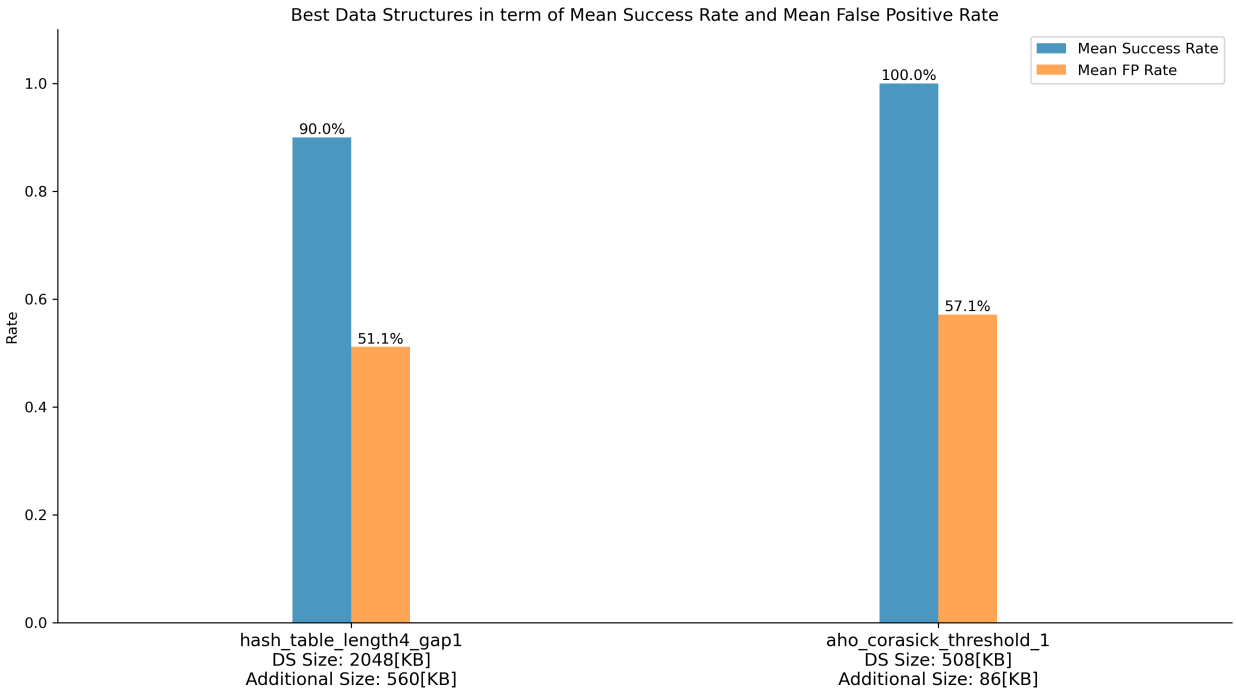

Figure 4.7: Mean of individual test results of the chosen variations of the cuckoo hash table (left) in comparison to the chosen variation of the Aho-Corasick automaton (right).

for this application. On top of that, the better space-efficiency of the Aho-Corasick adds to the already higher weighted score it has over the cuckoo hash table.

4.3 IBLT

Throughout this project, the SIDs list for each exact match is assumed to be implemented as a simple linked list. Under this work's assumption, a linked list containing n SIDs could be implemented using $2n$ [Bytes]. In this naive implementation, each SID ranges between 0 and $10^6 - 1$ as its valid range. This allows the usage of 32 bits integer to save this parameter. The pointer to the next item of the list takes another 32 bits, under this work's assumption of 32 bits architecture. In total, for a list containing n SIDs, the most simple and naive linked list implementation requires the following storage.

$$(32 + 32) \cdot n \text{ [bits]} = 8n \text{ [Bytes]}$$

In an attempt to reduce that, Bloom Lookup Tables (IBLTs), explained in Section 3.5.4, are used to store the SIDs. IBLT is a good candidate for the implementation of the secondary data structure only if it manages to achieve a high success rate for the List Entries operation, restoring the SIDs list. Each cell in the IBLT consists of:

- Count: 8 bits (sufficient for small collisions)

- Key sum: 32 bits (XOR of item values)

Usually, an IBLT cell also includes a value sum, however, the SIDs are only stored as keys, so there is no need of values assigned to them. Under these assumption, a cell in the IBLT is composed of $8 + 32 = 40$ bits. For high success rate for the List Entries operation, it is common to use at least $2n$ cells. Thus, the total size of the IBLT is calculated as follows:

$$\text{Total Storage} = \text{Number of cells} \times \text{Cell size} \tag{4.1}$$

$$\text{Total Storage} = 2n \times 40 \text{ bits} = 80n \text{ bits} = 10n \text{ Bytes} \tag{4.2}$$

The actual optimal size may vary depending on the specific characteristics of the data and the hash functions used.
The IBLT offers a trade-off between space efficiency and success rate for particular use cases. In Figure 4.8, the limit set by the naive implementation versus the performances is illustrated.

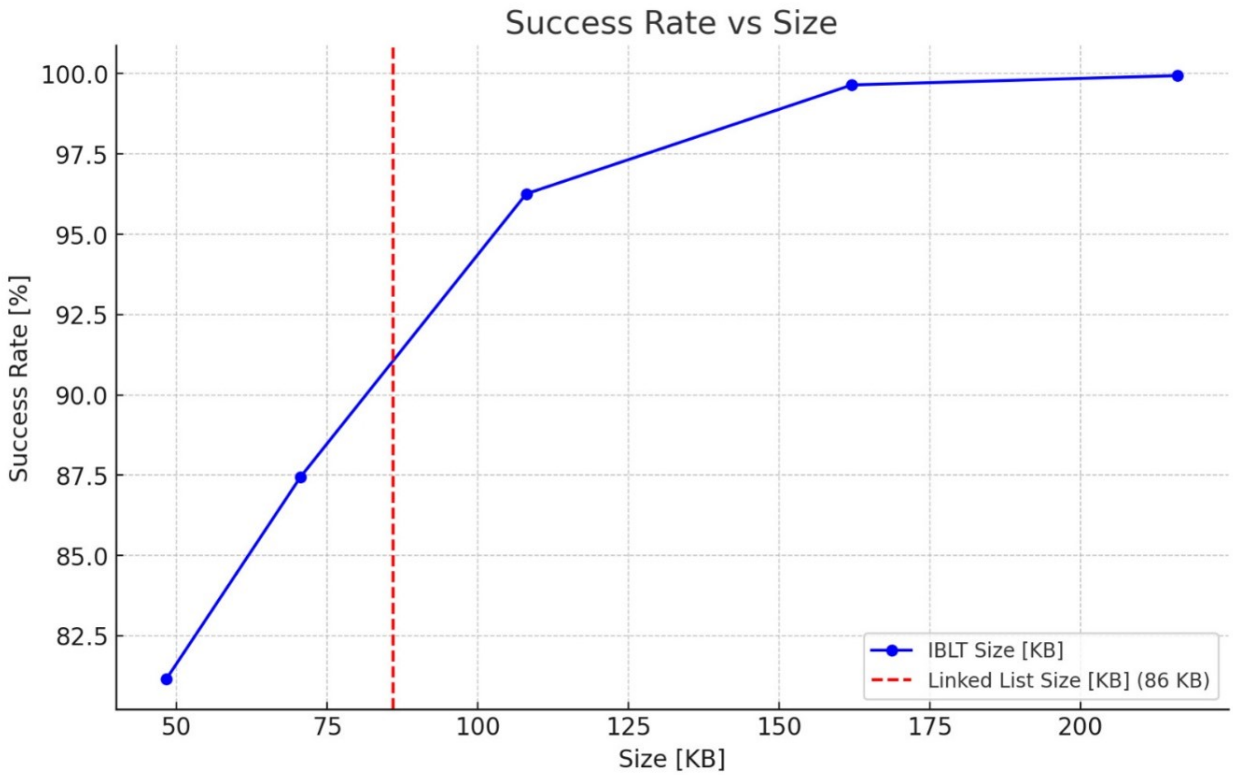

Figure 4.8: IBLT List Entries operation success rate versus size in [KB]. The red dashed line indicated the limit on store-efficiency set by the naive linked list implementation.

The success rate of restoring the SIDs list stored in the IBLT rises as the IBLT size increases. The red dashed line, at $86[KB]$ indicates the naive linked list implementation size, storing the entire set of SIDs for this work. This is calculated by taking the theoretical total storage of each implementation of the secondary data structure, replacing n with the actual value

of SIDs that are stored. Since the naive implementation guarantees 100% SIDs restoration, there is no use of IBLT implementations above this size. Therefore, the IBLT can only be used to offer a trade-off between SIDs restoration rate and overall size of the secondary data structure.

5 Summary and Outlook

In this work, the feasibility of a preliminary payload detection classifier was suggested, as a solution for the bottleneck of time-consuming DPI complex regex scanning. For this reason, Snort's community ruleset was parsed, using a generic and fully automated designated script. The exact matches extracted from the ruleset proved to be robust, in a sense that they covered for the entire SIDs in the original ruleset. The insertion threshold is then deducted to the range of $[1, 8]$ since for higher values the SID-coverage is not sufficient for reducing the scale of the DPI processing time. Two candidates were compared for the data structure, namely, the cuckoo hash table and the Aho-Corasick automaton. Performing individual end-to-end tests, showed a high feasibility of the suggested early classifier for both examined data structures. These tests used ten randomly generated suspicious payloads, including SIDs to detect. A trigger threshold was used to assist lowering the false positive rates. The exact determination of this threshold requires further investigation. However, this project found that using a constant threshold in relative to the highest hits result, gave a detection success rate of 100%, as well as low false positive rate greater than zero for the cuckoo hash table, and zero for the Aho-Corasick presented test.

This work evaluated the many variations of the two candidates for the data structure using the mean of these individual test results. The evaluation score was calculated using a 0.75 weight for the success rate and a 0.25 weight for the complement of the false positive rate. The two best performing candidates from each of the cuckoo hash table and the Aho-Corasick automaton were chosen using this score. The promising candidate of the cuckoo hash was the $L = 4$, $G = 1$ variation, and the promising candidate of the Aho-Corasick automaton was the one using the minimal insertion threshold of one.

The Aho-Corasick automaton with minimal insertion threshold was selected as the best performing candidate for the early classifier, out of an evaluation on all candidates introduced. Using this candidate, a robustness of zero negative rate, i.e., 100% success rate, was achieved. This, at a cost of 57.1% mean false positive rate, and a total of 594[KB] for this data structure.

Lastly, an implementation of an IBLT, as a way to store the SID lists, was evaluated. Compared to the naive implementation of a linked list, the IBLT only offered a trade-off between space-efficiency and SIDs restoration robustness.

For future works it is recommended to use a larger Snort ruleset, such as the paid subscription ruleset offered by Snort. This should be a rather easy task, as this project's generic algorithm was proven to deal with any Snort ruleset. Moreover, there is a need of a more robust benchmark to test the preliminary classifier. Since this project proved the feasibility of such classifier, it is suggested to continue evaluating the system's performance, considering latency and robustness tests. Ideally, these tests will evaluate the performances of the DPI as a standalone system, compared to the DPI with this work's suggested preliminary classifier.

Ultimately, the usage of the exact match preliminary classifier will greatly reduce the time-consumption induced by running the DPI on many rules. Thus, solving the bottleneck of today's IPS detection methods.

Bibliography

[1] Haircutfish, "Tryhackme snort: Task 9 snort rule structure, task 10 snort2 operation logic: Points to remember, & task 11 conclusion," 2022.

[2] R. Cox, *Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...)*. 2007.

[3] R. Cox, *Implementing regular expressions*. 2011.

[4] Scott Crosby, *Denial of Service through Regular Expressions*. 2003.

[5] Snort, "Snort website," n.d.

[6] Snort, "Payload detection options," n.d.

[7] Snort, "Pcre," n.d.

[8] Xiang Wang, "Introduction to hyperscan," 2017.

[9] A. Rashelbach, I. de Paula, and M. Silberstein, "Neurolpm - scaling longest prefix match hardware with neural networks," in *56th Annual IEEE/ACM International Symposium on Microarchitecture*, (New York, NY, USA), pp. 886–899, ACM, 2023.

[10] Niels Lohmann, "Json for modern c++," 2023.

[11] P. Rasmus and F. R. Flemming, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[12] Goyal et al., "A high-performance concurrent hash table," 2022.

[13] A. V. Aho and M. J. Corasick, "Efficient string matching," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[14] Christopher Gilbert, "C++ implementation of the aho corasick pattern search algorithm," 2018.

[15] "Revised c++ implementation of cjgdev's aho corasick algorithm," 2024.

[16] M. T. Goodrich and M. Mitzenmacher, "Invertible bloom lookup tables," in *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pp. 792–799, IEEE / Institute of Electrical and Electronics Engineers Incorporated, 2011.

[17] Jesper Borgstrup, "Py-iblt library," 2014.