

Virus Detection Acceleration

Analyzing Performance: Different Methods of Payload Scans for
Malicious Signatures

Supervisor: Alon Rashelbach

About Us



Itai Benyamin

Computer Engineering
Student



Idan Baruch

Computer Engineering
Student

Introduction

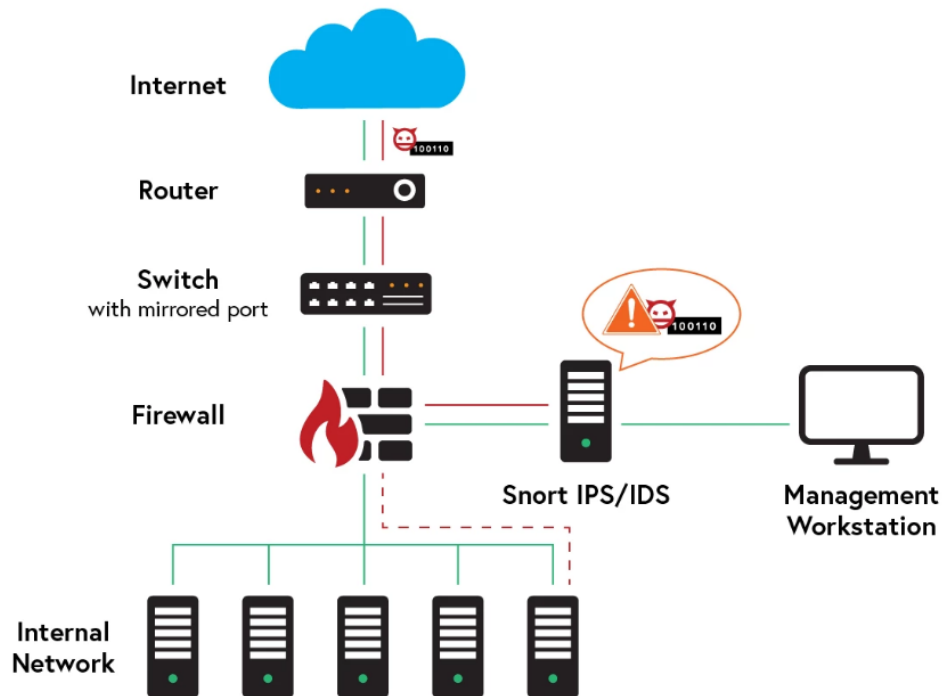
The Need for Fast and Secure Networks

- ▶ Internet traffic increases exponentially
- ▶ Modern networks must fulfil:
 - Robust Security
 - Fast Response Time

Tools Offer Robust Security

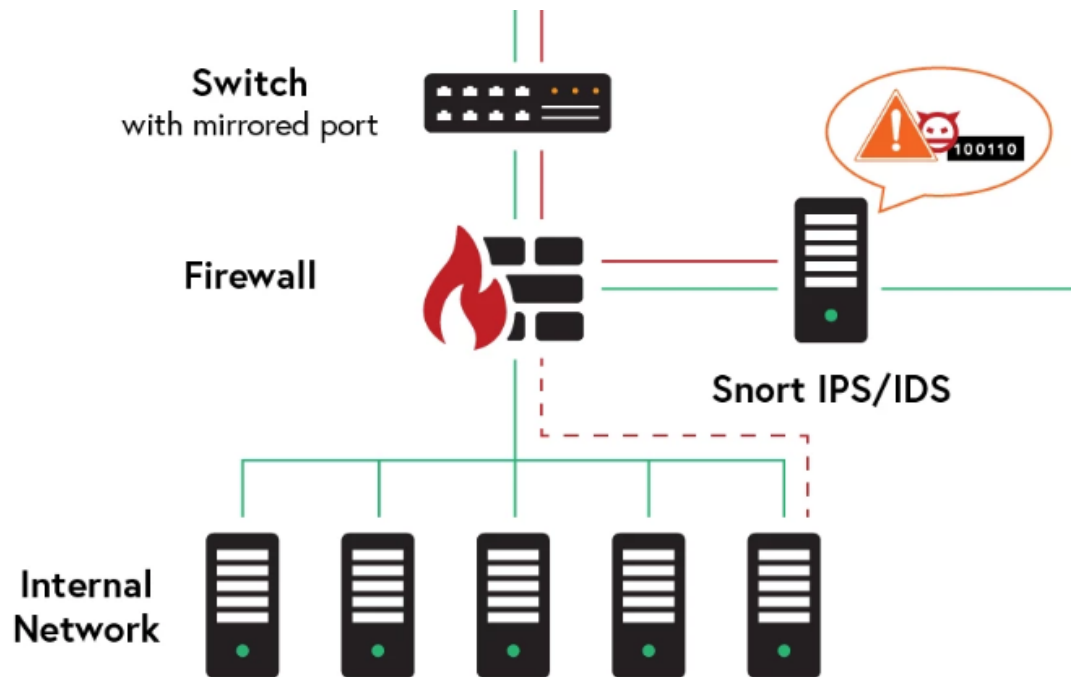
- ▶ Intrusion Prevention Systems (IPS)
- ▶ Deep Packet Inspection (DPI)
 - Uses complex Regular Expressions (regex)

Simple Snort Network Topology



Challenges Introduced by DPI

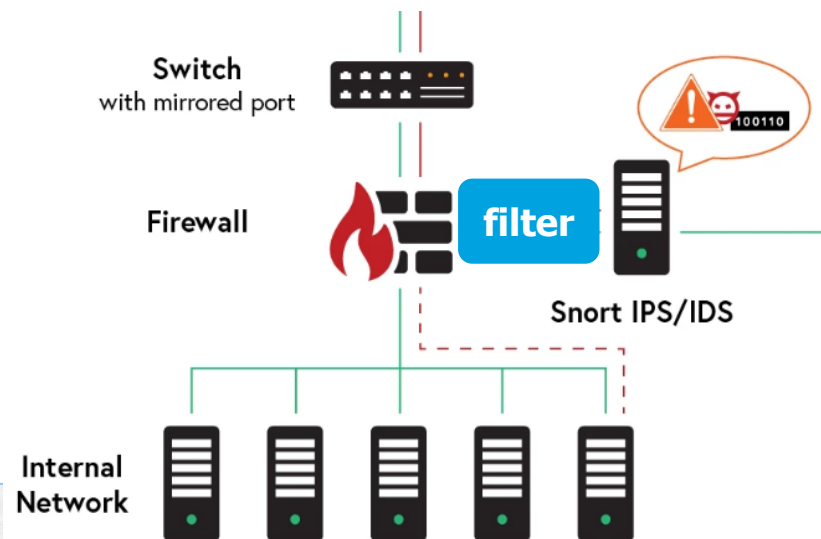
- ▶ Time consuming process
- ▶ Potential Bottleneck to the system



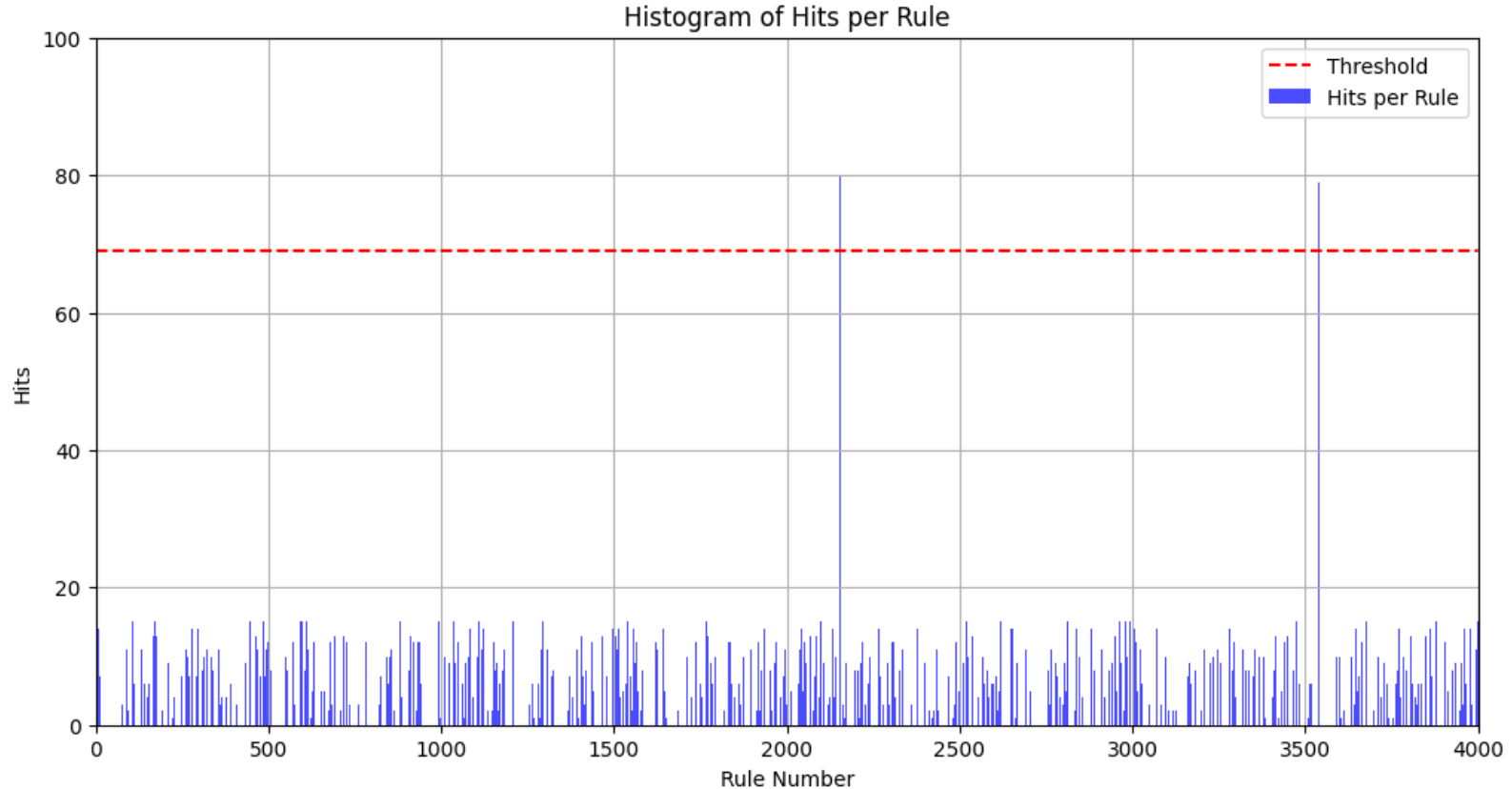
Motivation

What Do We Aim to Achieve

- **Faster matching** using exact string matching as a preliminary classifier to determine which rules should proceed to the DPI stage
- **Robustness** – the early classification stage does not harm the robustness of the overall system
- **Storage efficient solutions** minimizing data structure sizes and discussing the best storage-efficient solutions



How Do We Aim to Achieve it







Case-Study: Snort

What is Snort?

- ▶ Snort is an open-source IPS capable of real-time traffic analysis and packet logging.
- ▶ Snort has three primary uses (modes):
 - Packet Sniffer (like tcpdump)
 - Packet Logger (for network traffic debugging)
 - NIPDS (Network Intrusion and Prevention Detection System)

SNORT 101




Global Commands

Display version:
Snort -V
Snort -version

Do not display the version banner:
Snort -q

Use specific interface:
Snort -i eth0

Sniffer Mode



Verbose mode:
Snort -v

Display link-layer headers:
Snort -e


Display data payload:
Snort -d

Display full packet details in HEX:
Snort -X

Multiple flag usage. Display all packet details:
Snort -eX

Sniff "N" number of packets:
Snort -v -n 10

Logger Mode



Default log path :
/var/log/snort

Use alternative log path:
Snort -v -l /home/username/Desktop

Log in ASCII format:
Snort -v -K ASCII


Read snort files:
Snort -v -r snort.log

Read "N" number of packets:
Snort -v -r snort.log -n 10

Filter packets with "Berkeley Packet Filters" (BPF):
Snort -v -r snort.log tcp
Snort -v -r snort.log 'udp and port 53'

Default Log path ->
/var/log/snort


PCAP Processing



Process single pcap file:
Snort -c /etc/snort/snort.conf -q -r file.pcap -A console

Process multiple pcap files:
Snort -c /etc/snort/snort.conf -q -pcap-list="file1.pcap file2.pcap" -A console

IDS/IPS Mode



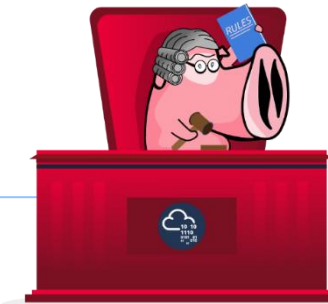
Use configuration file:
Snort -c /etc/snort/snort.conf

Test instance and configuration file:
Snort -c /etc/snort/snort.conf -T

Disable logging:
Snort -c /etc/snort/snort.conf -N

Run Snort in background:
Snort -c /etc/snort/snort.conf -D

Snort Rules



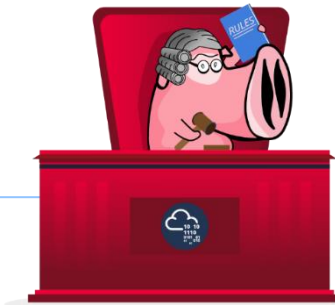
- ▶ Snort uses **rules** to find packets that match against them and generates alerts for users
- ▶ Rules have two main parts:
 - Rule Header
 - Rule Options

```
alert icmp any any <> any any { msg: "ICMP Packet found"; reference:CVE-XXXX; sid:1000001; rev:1; }
```

Diagram illustrating the structure of a Snort rule:

- Rule Header:** `alert icmp any any <> any any`
 - Source:** `any`
 - Destination:** `any`
- Rule Options:** `{ msg: "ICMP Packet found"; reference:CVE-XXXX; sid:1000001; rev:1; }`

Snort Rules

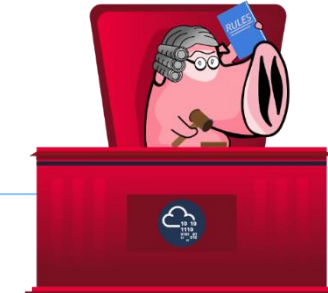


► Rule Header:

- Action
- Protocol
- Source (IP, port)
- Direction (orientation of traffic)
- Destination (IP, port)



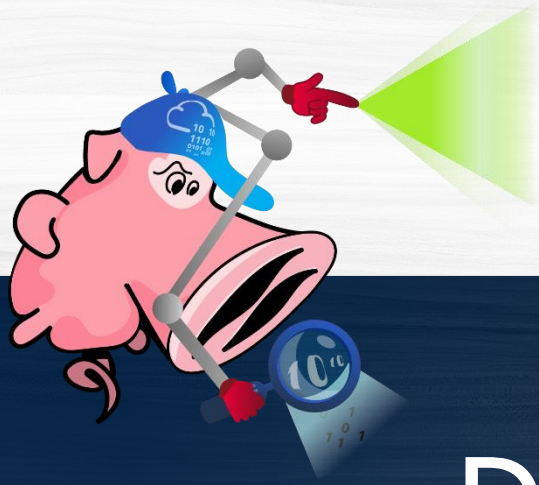
Snort Rules



► Rule Options:

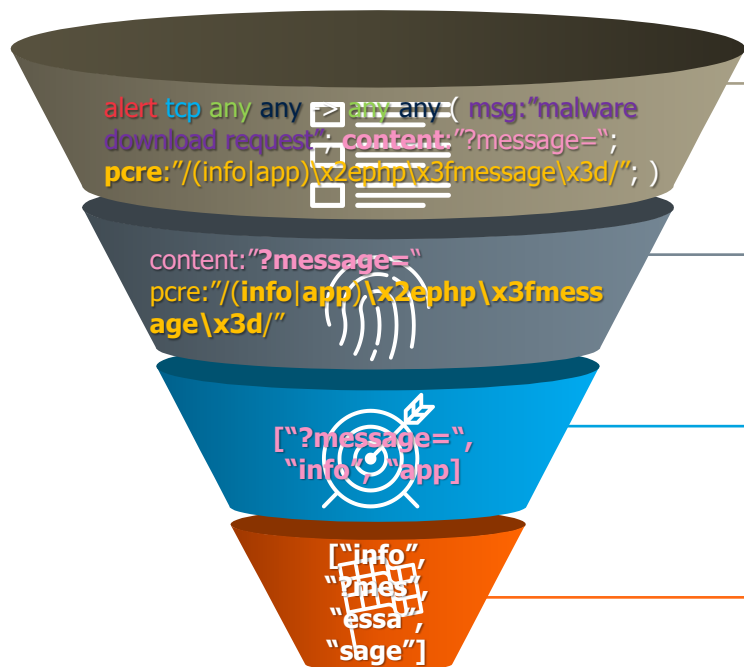
- General Rule Options (message logged, **sid**, reference, # of revisions)
- Payload Detection Options (**content**, **pcrc**)
- Non-Payload Detection Options
- Post-Detection Rule Options





Data Processing

Data Processing: From Data to Information



Rules

Predefined detection guidelines to identify specific network traffic patterns

Signatures

Unique identifiers within a rule that help to investigate and detect specific patterns in the payload data packets

Exact Matches

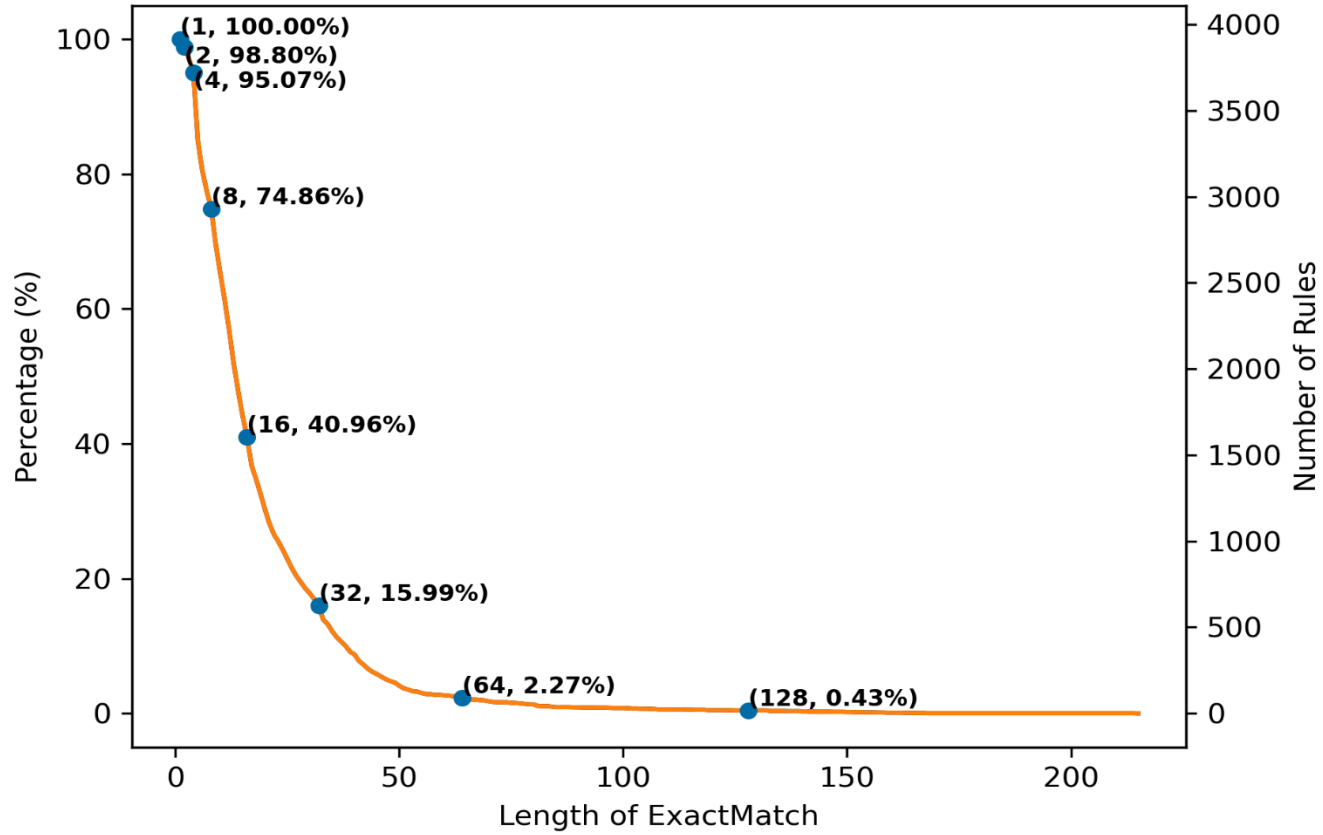
Deterministic components of signatures, removing any wildcards or elements that allow for multiple interpretations or ambiguity

Substrings

Segments or tokens extracted from an exact match, with specified length and gap between each other

Rules that remained after ExactMatch length threshold

(out of all rules with signatures)





Data Insertion

General Data Structure

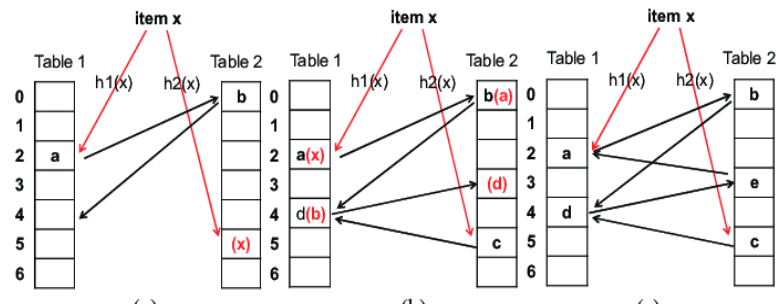
- ▶ Entries are pairs of Exact Matches and SIDs assigned to them.
- ▶ The Key is the Exact Match
- ▶ The Value is a pointer to List containing all relevant SIDs.

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

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

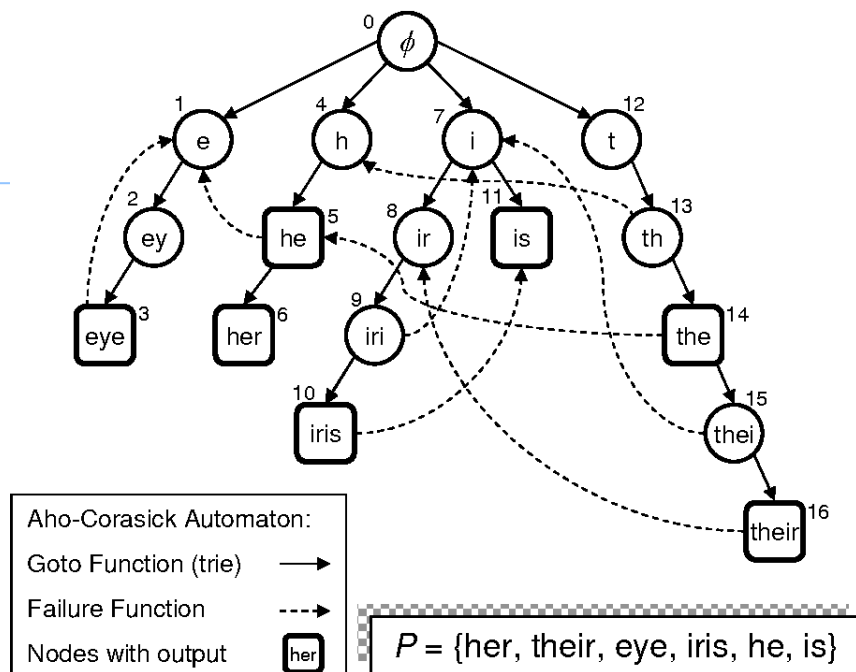
Cuckoo Hash-Table

- ▶ A cuckoo hash table is a data structure that uses multiple hash functions to resolve collisions.
- ▶ Lookup – $O(1)$
- ▶ Insertion – $O(n)$
- ▶ Examining 4 variations of entries:
 - Substrings of Length= $\{4,8\}$ with Gap= $\{1,2\}$
- ▶ We used efficient's libcuckoo implementation



Aho-Corasick

- ▶ Aho-Corasick algorithm is a string-matching technique that efficiently searches for multiple patterns simultaneously in each text by constructing an automaton from the patterns
- ▶ Lookup – $O(n+k)$
 - N is the length of the text searched
 - K is the number of patterns found
- ▶ Examining 8 variations of entries:
 - Each one with threshold in $[1, 8]$
- ▶ We used [cjpgdev's cpp implementation](#)



Aho-Corasick

- ▶ Aho-Corasick algorithm is a string-matching technique that efficiently searches for multiple patterns simultaneously in each text by constructing an automaton from the patterns
- ▶ Lookup – $O(n+k)$
 - N is the length of the text searched
 - K is the number of patterns found

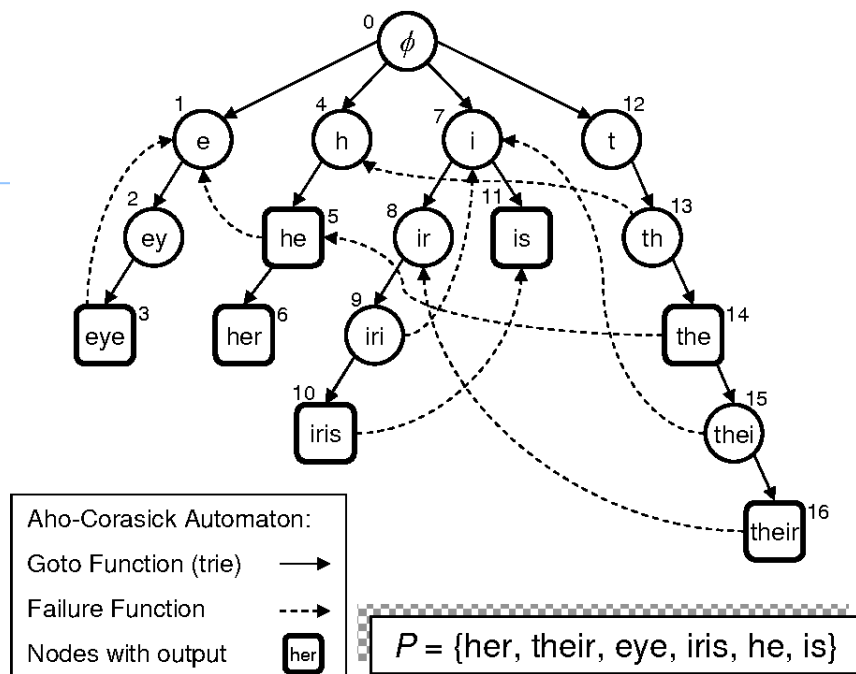
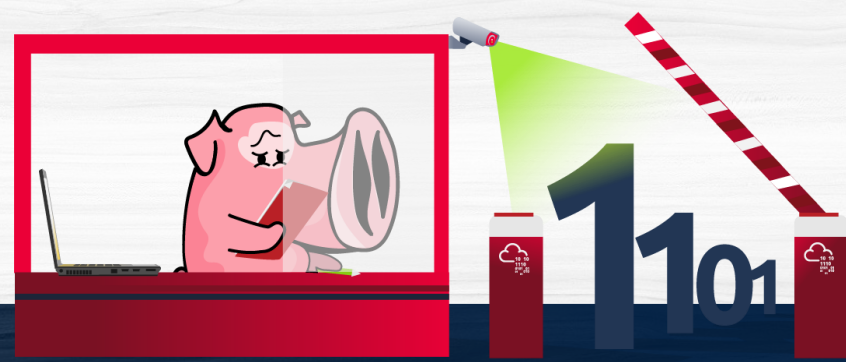


Table 3.3: Required storage for each transition in the Aho-Corasick GOTO table.

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



End-to-End Testing

End-to-End Testing

- ▶ Generated tests including signatures of suspicious traffic
- ▶ Each test is generated out of a specific PCRE – designed to match **only** against this PCRE's rule (“the **validated** rule” of the test)
- ▶ Rule triggered from the test payload are either the validated rule or false positives.

End-to-End Testing

- ▶ 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.

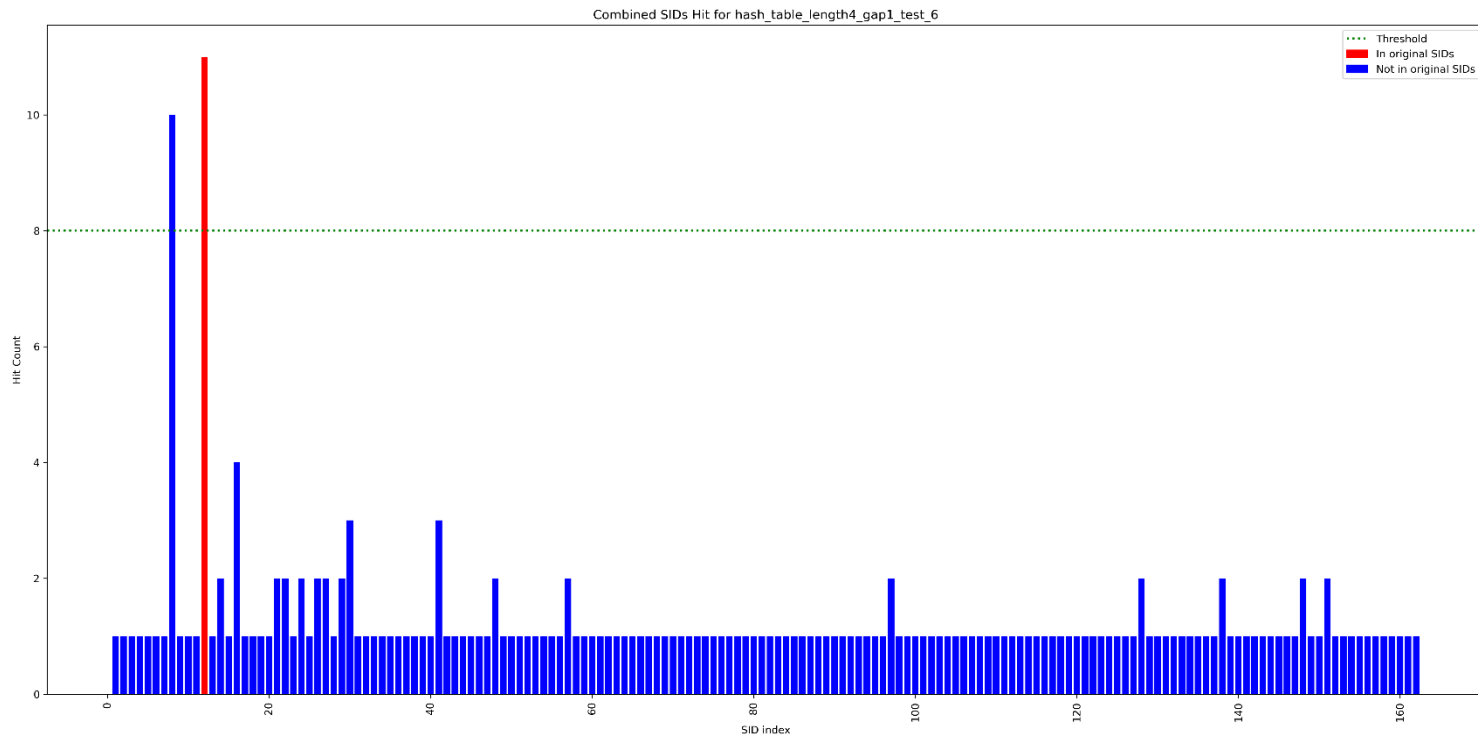
End-to-End Testing

- ▶ To lower the false positive rate a **trigger threshold** is applied during the tests
- ▶ The threshold filters rules below a certain “hit” parameter
 - A hit in the Cuckoo Hash is a match against a sub-string
 - A hit in the Aho-Corasick Automaton is the length of the pattern matched

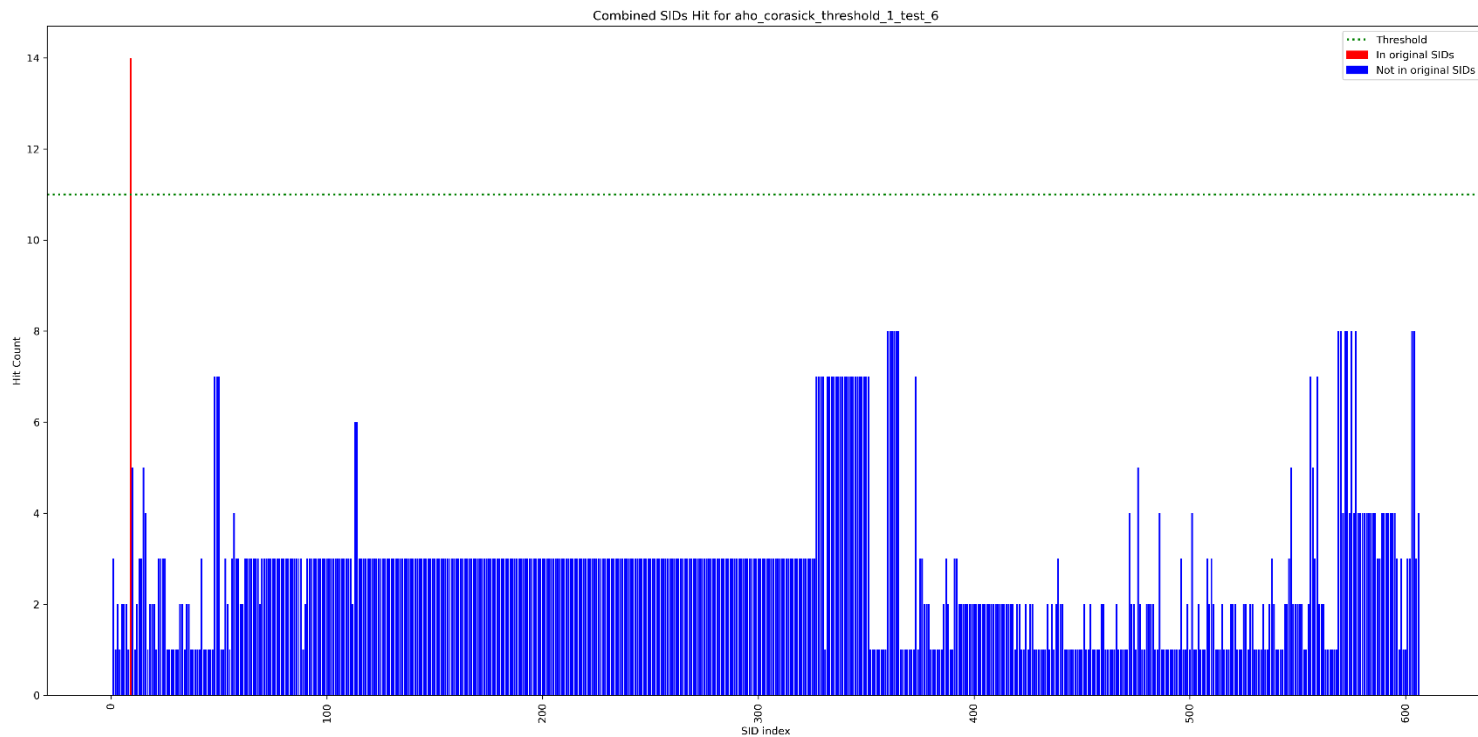
End-to-End Testing

- ▶ The search tests ran on each data structure variation, giving scores based on the following parameters:
 - Memory used by the data structure
 - Success rate (% of the real rules detected)
 - False Positive rate (% of the non-real rules out of all rules matched)

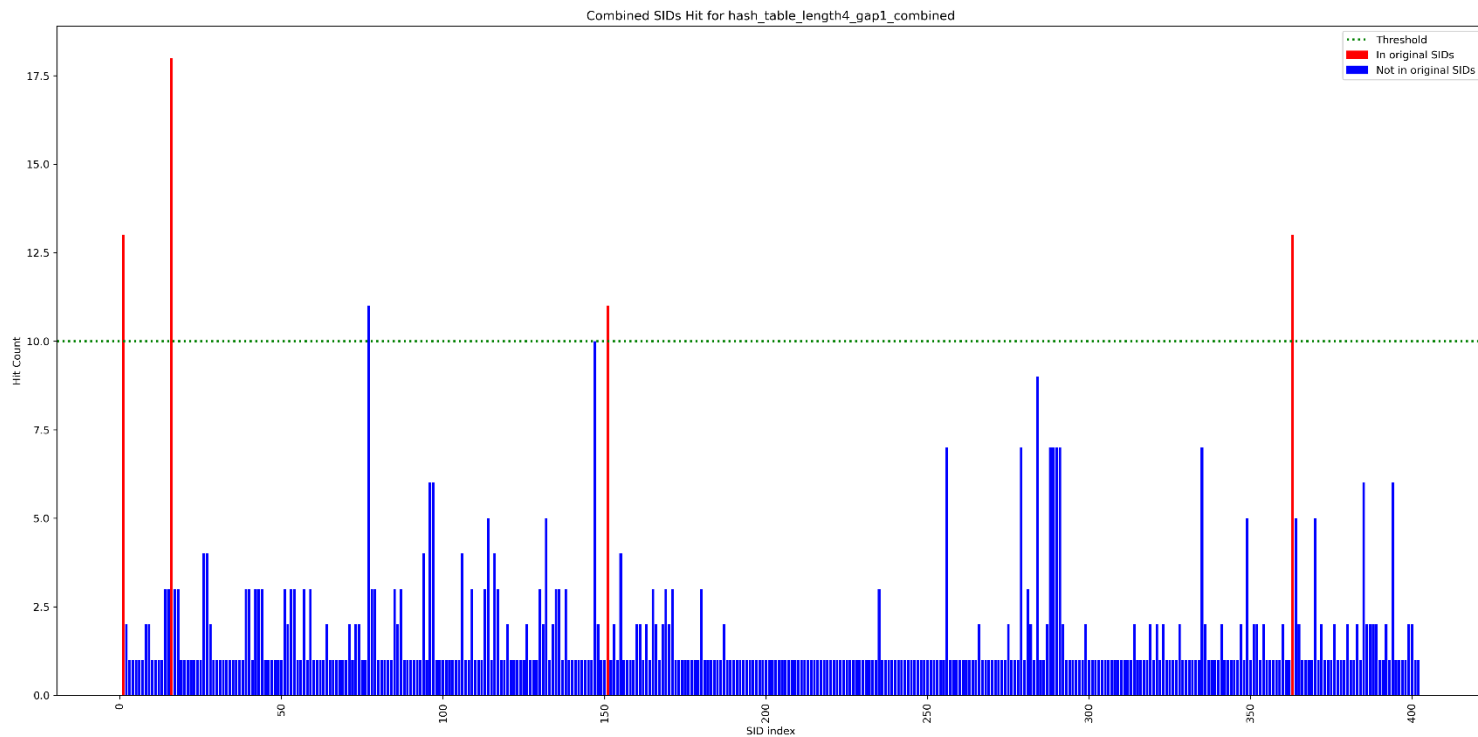
End-to-End Results (Cuckoo Hash)



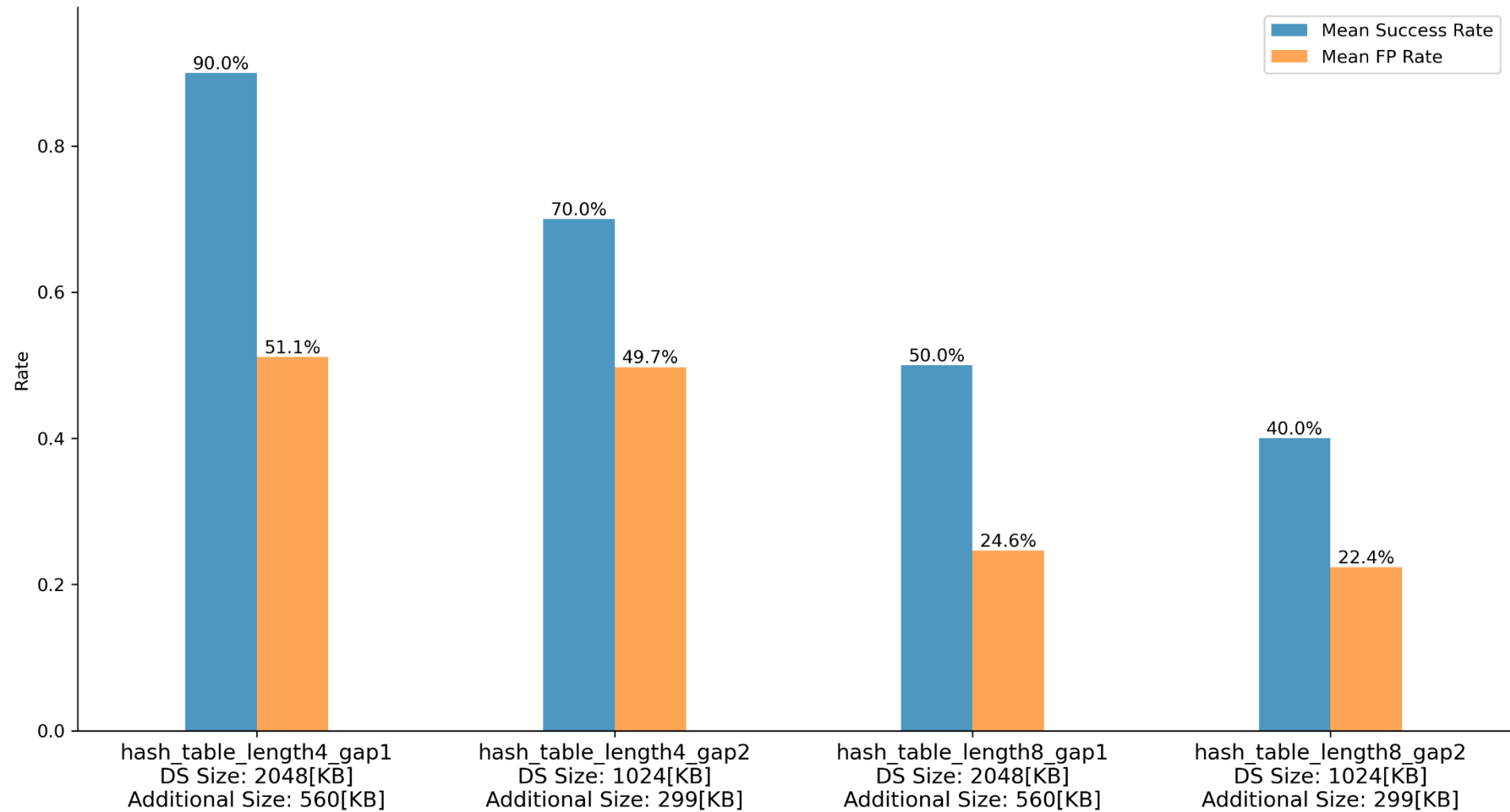
End-to-End Results (Aho-Corasick)



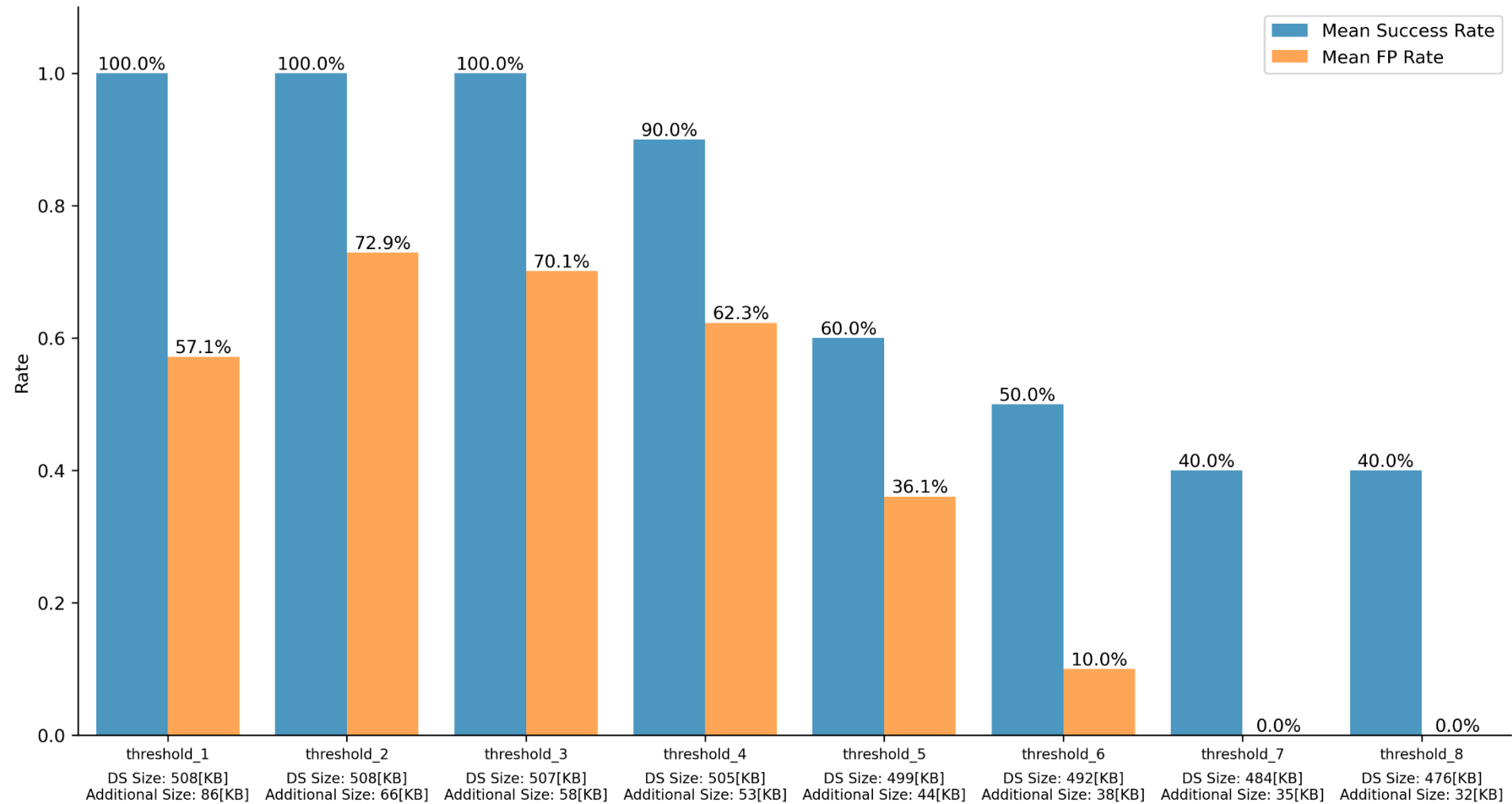
End-to-End Results



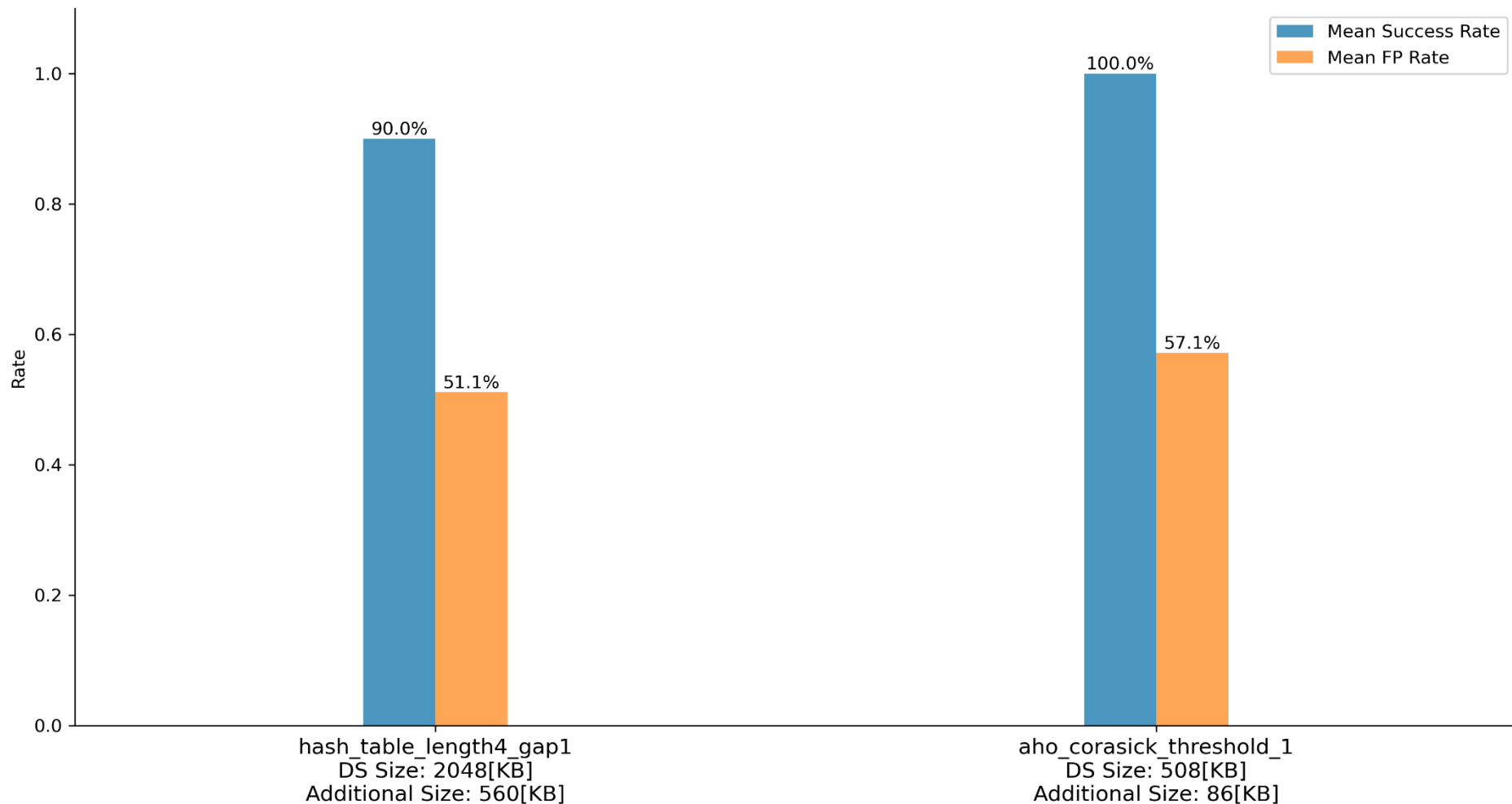
Hash Table Statistics



Aho-Corasick Statistics



Best Data Structures in term of Mean Success Rate and Mean False Positive Rate



IBLT

Recall

- ▶ Entries are pairs of Exact Matches and SIDs assigned to them.
- ▶ The Key is the Exact Match
- ▶ The Value is a pointer to List containing all relevant SIDs. (Up until here implemented as a Linked-List)

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

IBLT

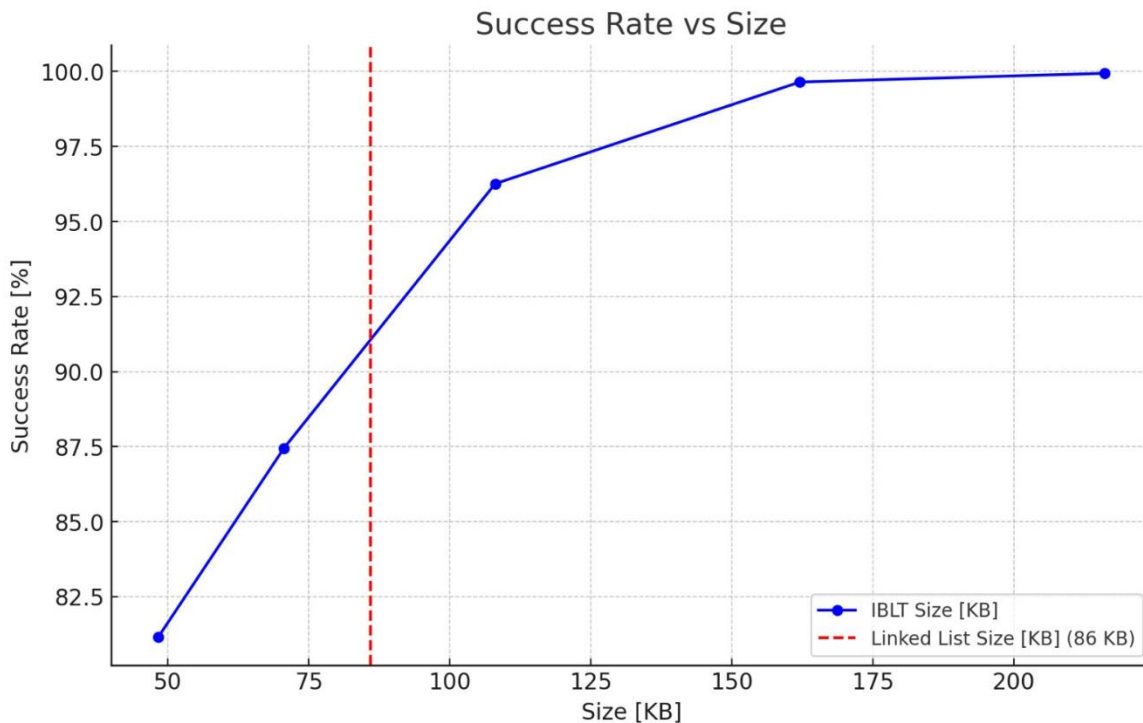
- ▶ IBLT – Invertible Bloom Lookup Table is a probabilistic data structure that allows efficient set reconciliation and element recovery.
- ▶ We can use IBLT in the storing of SIDs which associate with every Exact match:

'info' \rightarrow {1, 4, 10}

- ▶ This way we can save memory, however we risk with failure when retrieving relevant SIDs from the IBLT.

	count	key_sum	val_sum	hash_sum
0	2	0x0c151c030b	0b0100	0b0010
1	2	0x0602111c00	0b1001	0b1101
2	1	0x6d656c6f6e	0b0111	0b1001
3	1	0x6772617065	0b1010	0b0110
4	3	0x6b677d736e	0b1110	0b0100

Invertible Bloom Lookup Table





Summary & Conclusions

Important Insights

- ✓ **Parsing**
- ✓ **Cuckoo Hash**
- ✓ **Aho Corasick**
- ✓ **IBLT**
- ✓ **End-to-End**

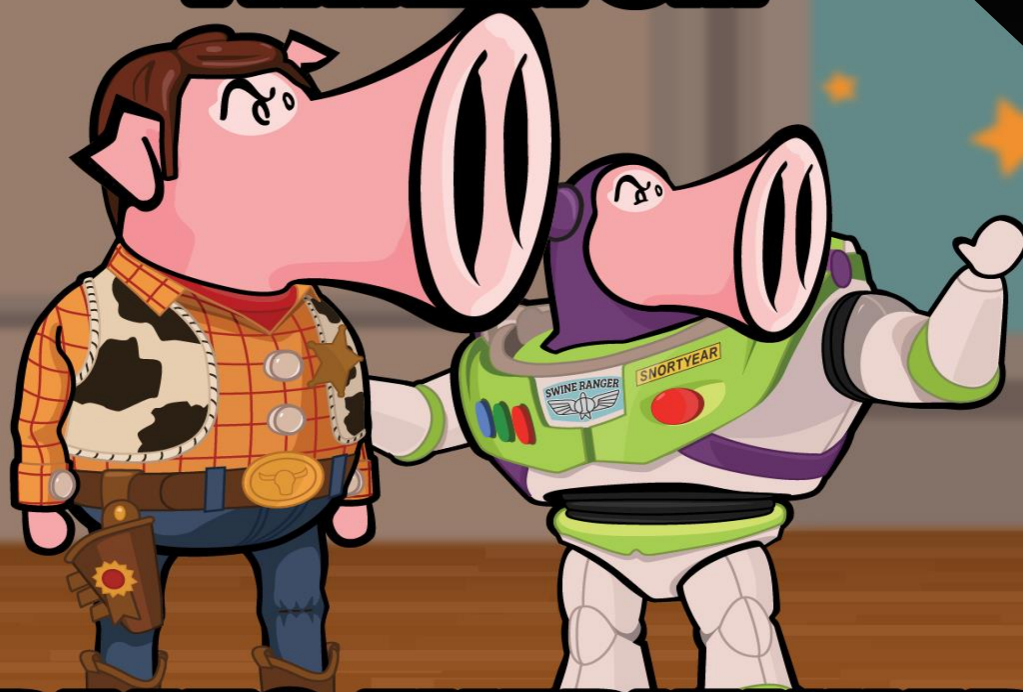
Achievements

- ✓ **Proven Feasible Solution**
- ✓ **Completely Generic Script** – our simulation can receive a .rules snort files and a test file and our script will provide all relevant results for this data (if they are in a required structure).
 - ✓ **This can help with future work.**
- ✓ **Our own Pull-Requests for relevant repositories.**

Suggested Future Work

- ✓ Use a larger Snort rulesets
- ✓ Find a more robust benchmark to test the preliminary classifier
- ✓ Create an evaluation of the overall system (considering latency and robustness):
 - ✓ DPI only
 - ✓ DPI with preliminary classification

THREATS...



**Thank you
for listening!**

THREATS EVERYWHERE