Lightweight Portable Intrusion Detection System for Auditing Applications

- Implementation and evaluation of a lightweight portable intrusion detection system using Raspberry Pi and Wi-Fi Pineapple

Carl Nykvist
Martin Larsson

Supervisor: Ali Hassan Sodhro
Examiner: Andrei Gurtov
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enskilda kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsmann i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Carl Nykvist
Martin Larsson
Abstract

The goal of this thesis was to develop, deploy and evaluate a lightweight portable intrusion detection system (LPIDS) over wireless networks. The LPIDS was developed by adopting two different string matching algorithms: Aho-Corasick algorithm and Knuth–Morris–Pratt algorithm (KMP). The LPIDS was implemented and tested on the hardware platforms Wi-Fi Pineapple and Raspberry Pi. To evaluate and test the LPIDS as well as the algorithms, performance metrics such as throughput, response time and power consumption are considered. The experimental results reveal that Aho-Corasick performed better than KMP throughout the majority of the process, but KMP was typically faster in the beginning with fewer rules. Similarly, Raspberry Pi shows remarkably higher performance than Wi-Fi Pineapple in all of the measurements. Moreover, we compared the throughput between LPIDS and Snort. It was concluded that the throughput was significantly higher for LPIDS when most of the rules do not include content parameters. This thesis concludes that due to computational complexity and slow hardware processing capabilities of Wi-Fi Pineapple, it could not become suitable IDS in the presence of different pattern matching strategies. Finally, we propose a modification of Snort to increase the throughput of the system.

Keywords: IDS, LPIDS, KMP, Raspberry Pi, Aho-Corasick, Wi-Fi Pineapple
Acknowledgments

The authors of this thesis would like to do the following acknowledgements:
Ali Hassan Sodro and Andrei Gurtov, thank you for great academic supervision and excellence in answering our questions during our thesis work.
Justin Gratto, Daniel Lester and Secure State Cyber, thank you for hardware tools, great support and the possibility to be able to complete this work.
Linus Sjöström, thank you for the valuable review and feedback.
Contents

Abstract iii
Acknowledgments iv
Contents v
List of Figures vii
List of Tables ix

1 Introduction 1
1.1 Motivation ........................................... 2
1.2 Research questions ................................... 2
1.3 Main contributions ................................... 2
1.4 Delimitations ........................................ 3
1.5 Secure State Cyber ................................... 3
1.6 Thesis Outline ....................................... 3

2 Background 4
2.1 Intrusion Detection Systems ........................ 4
2.2 Hardware ........................................... 7
2.3 Protocols ............................................ 8
2.4 Cryptography ....................................... 12
2.5 Algorithms and Data Structures ..................... 14
2.6 Related Work ....................................... 18

3 Method 23
3.1 Buildroot ........................................... 23
3.2 Implementation ..................................... 24
3.3 Comparative Analysis ............................... 32

4 Results 36
4.1 Throughput ......................................... 36
4.2 Content Matching Time ............................... 37
4.3 Response Time ...................................... 37
4.4 Rule Filter Performance .............................. 37
4.5 Snort vs LPIDS ..................................... 38
4.6 Power consumption .................................. 40
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Intrusion detection systems classification.</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>IPv4 header.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>TCP header.</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>UDP header.</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>ICMP header.</td>
<td>11</td>
</tr>
<tr>
<td>2.6</td>
<td>IEEE 802.11 data frame format.</td>
<td>12</td>
</tr>
<tr>
<td>2.7</td>
<td>Stream cipher.</td>
<td>13</td>
</tr>
<tr>
<td>2.8</td>
<td>Block cipher.</td>
<td>13</td>
</tr>
<tr>
<td>2.9</td>
<td>R-Tree.</td>
<td>15</td>
</tr>
<tr>
<td>2.10</td>
<td>R-Tree minimum bounding rectangles.</td>
<td>15</td>
</tr>
<tr>
<td>2.11</td>
<td>Trie example.</td>
<td>17</td>
</tr>
<tr>
<td>2.12</td>
<td>Goto function.</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>The initialization of the LPIDS.</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>The main packet capture loop of the LPIDS.</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>TCP reassembly functional diagram.</td>
<td>28</td>
</tr>
<tr>
<td>3.4</td>
<td>IPv4 parsing.</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>IEEE 802.11 frame.</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>IEEE 802.11 frame type distribution.</td>
<td>34</td>
</tr>
<tr>
<td>3.7</td>
<td>Protocol distribution.</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>Histogram of the packet data used for evaluation.</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Throughput comparison per number of rules.</td>
<td>36</td>
</tr>
<tr>
<td>4.2</td>
<td>Content matching per number of rules.</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Response time per number of rules.</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>R-tree and Linear search comparison.</td>
<td>38</td>
</tr>
<tr>
<td>4.5</td>
<td>Throughput per number of rules with content parameter on Raspberry Pi 3 Model B+.</td>
<td>39</td>
</tr>
<tr>
<td>4.6</td>
<td>Workload distribution of Snort with varying number of rules with a content parameter.</td>
<td>39</td>
</tr>
<tr>
<td>4.7</td>
<td>Content matching on Snort between AC-BNFA and AC-FULL.</td>
<td>40</td>
</tr>
<tr>
<td>4.8</td>
<td>Workload distribution of Snort with respect to number of rules.</td>
<td>41</td>
</tr>
<tr>
<td>4.9</td>
<td>Power consumption per throughput.</td>
<td>41</td>
</tr>
<tr>
<td>4.10</td>
<td>LPIDS workload distribution per number of rules.</td>
<td>42</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of packet analysis in LPIDS.</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview of packet analysis in Snort.</td>
<td>44</td>
</tr>
</tbody>
</table>
6.1 Proposed modification of Snort using R-trees.
List of Tables

2.1 Failure function .................................................. 17
2.2 Success function .................................................. 18
3.1 Supported rule formats .......................................... 26
3.2 Throughput values ................................................. 33
1 Introduction

As internet technologies advance, the focus on security increases immensely. Because of the rapidly increasing network technology, there is an increased need for security in every domain. The ongoing development and adoption of Internet of Things (IoT) in both our homes and workplaces are relying on secure, trustworthy and privacy-preserving infrastructures [39]. As due to increase in number of attacks there is a dire need of security auditing [13]. Identifying these attacks is crucial to prevent information from being compromised as well as unauthorized access. There are opportunities to detect intrusion attempts and network-based attacks by using an intrusion detection system (IDS).

An IDS collects and controls activities in a system. It can also detect suspicious activities by applying a fixed rule set (signature detection) or statistical-based methods using various types of machine learning (anomaly detection) [60]. One of the most popular examples of an IDS is Snort, which is open-source software for Linux and Windows. It is essential to prioritize system security, minimize vulnerabilities and prevent the system from intrusion by attackers [8].

This thesis adopts two popular pieces of hardware such as, Wi-Fi Pineapple and Raspberry Pi. Wi-Fi Pineapple contains tools for wireless penetration testing. These tools can perform reconnaissance, man-in-the-middle attacks and logging. Raspberry Pi is a single-board computer used for various applications for example, home automation and robotics.

When the number of rules in a portable IDS increases, its efficiency and performance decreases. This is particularly important to consider in hardware with limited processing capability. To maintain efficiency, the IDS requires suitable data structures and algorithms to process the observed network traffic efficiently. The key component in an IDS is the pattern matching algorithm which scans the captured network packets for a set of predefined patterns. The most commonly used algorithm for this purpose is the Aho-Corasick multi-pattern matching algorithm [29] [61].
This thesis aims to create a Lightweight Portable IDS (LPIDS) suitable for both Wi-Fi Pineapple and Raspberry Pi. It also presents a comparative analysis to investigate the effectiveness between Wi-Fi Pineapple and Raspberry Pi in terms of throughput, power consumption and response time. The implementation of the LPIDS has been done using the two different pattern matching algorithms: Aho-Corasick algorithm and Knuth–Morris–Pratt algorithm (KMP). As previously stated, Aho-Corasick is a multi-pattern matching algorithm, capable of matching multiple patterns simultaneously, which is beneficial for a larger set of patterns. KMP is a single-pattern matching algorithm which only searches for one pattern at a time.

1.1 Motivation

The Common Vulnerabilities and Exposures (CVE) lists had more than 15000 reported security vulnerabilities in 2018 [13], demonstrating the increasing necessity for monitoring against threats through an IDS. An IDS is observed as an increasingly important component in enhancing the security of a (wireless) network, in light of the increasing complexity of attacks in the wild. An IDS is usually subject to the debate of a cost-benefit analysis, meaning that the efficiency of these protections is of high importance and whatever can be automated is of the highest benefit to a customer. The advantage with a portable IDS compared to being permanently integrated into a router is that it may be used by security consultants when performing security audits for different clients. Additionally, a portable device may be used in more and various locations (e.g. trains, buses and airplanes) where its size has advantages over larger fixed-placement IDSs.

1.2 Research questions

This thesis intends to answer the following questions:

- Can the throughput, power consumption and response time of an IDS increase by applying a single-pattern matching algorithm such as KMP, as an alternative to the multi-pattern matching algorithm Aho-Corasick?

- Is a lightweight portable IDS viable on embedded devices such as Wi-Fi Pineapple and Raspberry Pi in terms of throughput, power consumption and response time?

- How does LPIDS compare to Snort in terms of throughput on a Raspberry Pi?

1.3 Main contributions

There are three key contributions of this thesis as follows.

First, an efficient and lightweight intrusion detection system (LPIDS) is proposed. Second, the LPIDS was developed, implemented, tested and evaluated using Aho-Corasick and KMP on two different hardware platforms: Wi-Fi Pineapple and Raspberry Pi. Third, a comparative analysis of proposed LPIDS is done in terms of network metrics such as throughput, power consumption and response time with regard
to their counterparts. Additionally, the proposed LPIDS is suggested for consultants while performing security audits.

1.4 Delimitations

This master’s thesis will only use 10,000 rules when testing the LPIDS in terms of the metrics due to the limitation of time and hardware capacity. The LPIDS handles wireless traffic and cannot handle HTTPS traffic because it requires the private key to decrypt the traffic. Additionally, it only runs the tests on one large packet capture file and not multiple files. The implemented system supports monitoring of networks using WEP, WPA-PSK and WPA2-PSK but not the newer WPA3 standard or Enterprise versions of WPA. Finally, the system was created to be run in one thread to achieve the best results on Wi-Fi Pineapple as it has a single-core processor.

1.5 Secure State Cyber

Secure State Cyber, for which this thesis was performed, is a company that has specialized in information security since 2005. They provide solutions in information security for companies, organizations and authorities. Their customers consist of both companies in the private sector as well government agencies such as authorities and defence agencies.

1.6 Thesis Outline

This thesis starts with Chapter 1, motivating the problem and explaining the main contributions. It also presents the three research questions that are going to be answered at the end of the thesis. Chapter 2 provides a background of all the theory and concepts the reader needs to understand. This includes basics of intrusion detection systems, hardware specifications, cryptography used in the LPIDS, network protocols and the theory behind the algorithms and data structures. In Chapter 3, the reader is presented with explanations of how the LPIDS was developed and the different parts of it. At the end of this chapter, the comparative analysis is explained, meaning how the algorithms and LPIDS will be tested in terms of throughput, power consumption and response time. Chapter 4 presents the acquired results from the different tests measured on both hardware platforms. The discussion in Chapter 5 analyses the results and evaluates the method. It also discusses the ethical and societal aspects related to the work. Lastly, Chapter 6 answers the research questions as well as describing possible ideas for future work.
2 Background

In this chapter, we present background about the different technical concepts that the reader is required to understand as well as other related work to this paper that has been done in the field.

2.1 Intrusion Detection Systems

An intrusion is a number of actions that attempt to compromise the integrity, confidentiality or availability of a system [21]. Usually, an intrusion is done by an unauthorized person aiming to obtain access and compromise a system. Intrusion detection is a security technology that aims to determine if a computer system is being attacked. It provides mechanisms to gather and analyze information on the network or host in order to identify potential security anomalies and alert the administrator. There may be different purposes of IDSs; one might detect web attacks by only considering HTTP requests, while a system intended to monitor traffic to the FTP server might only consider bruteforcing of login credentials to the FTP server. All IDSs have a common general definition of "intrusion" as an unauthorized usage, or misuse, of a computer system. An IDS also provides forensic information to administrators and organizations which may be used to discover the origins of an attack. This makes intrusion detection systems an essential component of a security system [44]. An IDS has the capability to identify the following attempts [35]:

- HTTP attacks against web servers
- Port scans
- Inside hacking on the operating system
- Additional checks for ports opened through firewall intentionally or unintentionally
2.1 Intrusion Detection Systems

2.1.1 Basics of an IDS

Since there are many different IDSs used, a standard model has been established called Common Intrusion Detection Framework (CIDF). CIDF defines a set of components which forms an IDS. These boxes include event generators (E-boxes), analysis engines (A-boxes), storage mechanisms (D-boxes) and countermeasures (C-boxes). The objective of E-boxes is to provide information about events to the rest of the system, which is required for other boxes to conclude security events. The output of E-boxes can be complex events or low-level network events. A-boxes analyze the information generated from E-boxes and extract the relevant information. The function of D-boxes is handling storage mechanisms as well as making the data available at a later time to the system’s operators. C-boxes define what to do if an attack is identified and product output in reaction to events. This can be, for example, sending an alarm, shutting down a server or TCP connections. C-boxes allow the IDS to prevent further attacks from occurring after an initial attack has been detected [44].

In terms of accuracy of an IDS, there are four possible states for each potential activity. The four possible states are true positive state, true negative state, false positive state and false negative state. A true positive state is when an IDS alerts for an attack that is actually an attack. A true negative state is when a user does something that is allowed and the IDS does not alert. An example could be if a user with permission logins to a FTP server, and the IDS does not alert. False positive is an alert triggered mistakenly by an IDS reporting an intrusion but in reality it is just a normal network traffic. This is sometimes called a false alarm rate. False negative state is a missed attack by the IDS which can cause serious damages into the network. If this happens, the security professional or administrator of the IDS has no idea if an attack occurred [15, 22].

A classification of IDSs are shown in Figure 2.1.

There are two broad types of IDS: host-based intrusion detection system (HIDS) and network-based intrusion detection system (NIDS) [9].

2.1.2 Network-based Intrusion Detection System

NIDS detect intrusions in network data by inspecting both incoming and outgoing traffic. It monitors all the raw network traffic for all devices on the network. NIDS performs well at distinguishing attacks that concern low-level manipulation of the network but cannot detect if, for example, unauthorized users make changes to the network.
2.1. Intrusion Detection Systems

A NIDS works by examining the content of the packets on the network by parsing them, analyzing the protocols used and extracting the relevant information from them [17].

2.1.3 Host-based Intrusion Detection System

A HIDS only has sensors located within the host or operating system, analyzing the internals of a computer system. An example of what a HIDS may detect are files that have been modified from a previous state where these files should not be modified at all. It can also analyze what applications are used and what files are accessed at the host level. However, a HIDS needs to be configured on every host and can be vulnerable to Denial-of-Service attacks (DoS). This makes managing HIDS complicated [44].

2.1.4 Different Approaches for Detection and Analysis

The two most common traditional intrusion detection approaches are misuse detection and anomaly detection. Misuse detection uses knowledge about patterns in the data flow that correspond to known attacks in the form of signatures. This approach is useful when detecting known attacks but not useful for identifying unknown attacks since, naturally, signatures have not yet been written for them. Another aspect of misuse detection is that any mistakes in the signatures written will increase the false alarm rate, making the IDS less effective [17, 47].

To implement misuse detection, pattern matching or rule-based techniques are usually used. Pattern matching is mostly used in NIDS where patterns are created, matched, and identified based on the content of a network packet. The advantage of using pattern matching is the fast search because the pattern is only matched in the extracted protocol fields, which reduces the search space for patterns [17, 47].

Another approach of misuse detection is rule-based techniques where a set of rules are matched against network traffic data. If any network data traffic deviates from the rules, an intrusion is reported. For instance, a rule can be written to alert the administrator if someone tries to log in with an incorrect password more than five times [17, 47].

Anomaly detection detects intrusions by classifying the data in terms of normal or anomalous. A model consists of the four phases: data collection, normal system profile, anomaly detection, and response. Data collection is the training phase where normal user activities are gathered over some time. After data has been gathered, normal user profiles are created from this data. Anomaly detection decides which traffic deviates from the normal system profiles and which traffic should be detected as malicious. If an intrusion has occurred, the response component reports the intrusion [9, 17, 47].

Anomaly detection is advantageous over misuse detection in its ability to detect unknown attacks. However, anomaly detection increases high false alarm rates. Also, if malicious traffic is within the defined baseline, the IDS does not alert to it [9, 17, 47].
2.1.5 Snort

One popular example of a NIDS is the open-source tool Snort. Snort uses misuse detection, where the administrator creates the rules. Rules can also be submitted to Snort by members of the open-source community. The first step for Snort when a packet arrives is to decode it by determining which protocols the packet contains. While the decoder investigates the packet headers, it checks if any headers deviate from the rules.

Snort is a signature-based IDS, meaning it evaluates specific patterns to detect attacks according to a set of given rules. The rules are created by an administrator to detect different types of attacks and probes. An example rule is the following:

```plaintext
alert tcp 192.168.1.14 any -> 111.111.1.1 80
    (msg : “incoming TCP packet”;
    content : “payload string”;
    sid : 220312;
)
```

In this rule, it detects if the IP address 192.168.1.14 sends a TCP packet through port 80 to the IP address 111.111.1.1. The message option is managed by the administrator and will be printed if the NIDS detects this rule. The sid option is a unique number for each rule [53].

In Snort, string matching is the computationally expensive operation. The string matching routines in Snort account for up to 70% of total execution time and 80% of instructions executed on packet capture files [7].

2.2 Hardware

In this section, we present the background of the two different hardware platforms used to run the LPIDS.

2.2.1 Wi-Fi Pineapple

Wi-Fi Pineapple is a purpose-built wireless auditing platform for network penetration testing [62]. In its Tetra configuration, it includes dual integrated dual-band (2.4GHz-5GHz) radios, an Atheros AR9344 system on chip with 64MB RAM and 2GB NAND flash. The default operating system is the Linux based OpenWrt which provides a minimal implementation of the C standard library called uClibc as well as uClibc++ for the C++ standard library. The basis of the platform is the PineAP Suite, which contains a range of penetration testing tools and an API for users to create their modules and tools. PineAP includes modules for creating rogue access points for man-in-the-middle attacks, and various flooding and deauthentication attacks for DoS.

2.2.2 Raspberry Pi

Raspberry Pi is a tiny computer that can run many different operating systems. It contains a motherboard, RAM, USB ports, Ethernet port as well as HDMI support. In
this thesis, the Raspberry Pi 3 Model B+ is used. This specific model was released in 2018 and some of the specifications are the following [46]:

- Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz
- 1GB LPDDR2 SDRAM
- 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN
- Micro SD port

2.2.3 Network Interface Controller modes

The Wireless Network Interface Controller (WNIC) is a piece of computer hardware which connects a computer to a computer network. To monitor the data arriving on a wireless channel, a packet sniffer listens to data that arrives at the WNIC. The WNIC can be set in different modes depending on the purpose of the packet sniffer, but the three most used modes are managed mode, promiscuous mode and monitor mode. Managed\(^1\) mode is used when a node connects to a network. Promiscuous mode is used when sniffing all the packets that are destined or not destined to a controller device, after connecting to an access point. The promiscuous mode is possible to use in both wireless and wired networks and is especially useful when analyzing packets with Wireshark or tcpdump. Lastly, monitor mode performs passive capturing for a wireless network without associating to the network itself. In this thesis, the IDS uses monitor mode because it does not require to connect to any network, but instead, only want to listen to the data that is sent through it [6].

2.3 Protocols

In this section, the five major protocols and techniques used are described to explain what different fields they use.

2.3.1 IPv4

IPv4 is the standard protocol and most used nowadays. An IPv4 packet contains a header section and data section, where the header contains multiple fields as shown in Figure 2.2.

![IPv4 header](https://linux.die.net/man/8/iwconfig)
The important fields for this report are the total length, flags, protocol, source address and destination address. Total length is the length of the datagram, which is the entire packet, in bytes including the header and the data section. This field is required to contain between 20 and 65,535 bytes. The flags field is a three-bit field used to control fragments. Fragmentation occurs when the IP packet traverses a network which has a maximum transmission unit (MTU) smaller than the size of the datagram. The bits are defined as the following:

- Bit 0: Reserved, must be zero
- Bit 1: Don’t Fragment (DF)
- Bit 2: More Fragments (MF)

DF is set when a datagram cannot be fragmented and if this happens, it drops the packet and sends back an ICMP Destination Unreachable message. MF is set when the current datagram is a fragmented packet in a larger fragment, which means that this datagram is not the last fragment.

The protocol field varies depending on what protocol it is. For example, if it contains 0x01, the ICMP protocol is used while 0x06 maps to TCP. Lastly, the source address is the IP address of the sender while the destination address is the IP address of the receiver.

2.3.2 TCP

TCP is one of the main protocols used in the transport layer of the protocol stack. It provides reliable delivery of packets between applications that communicate on an IP network. TCP is connection-oriented meaning that before one application process can send data to another, the two processes must first agree on a handshake. This guarantees that the transmitted data will arrive in the proper sequence. The advantage with TCP is the reliable data transfer, which guarantees that the byte stream sent is the exact byte stream received.

The TCP header consists of multiple fields shown in Figure 2.3.

```
<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgement Number</td>
<td></td>
</tr>
<tr>
<td>Offset</td>
<td>Reserved</td>
</tr>
<tr>
<td>TCP Flags</td>
<td>Window</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent Pointer</td>
</tr>
<tr>
<td>TCP Options(variable length, optional)</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 2.3: TCP header.

The fields in focus for this report in the above figure are source port, destination port, sequence number, acknowledgement number and flags. Source port is port of the sender and destination port is the port of the receiver. The 32-bit sequence number is used to identify each byte of data sent between computers in order for it to be reconstructed later on. This makes sure packets can be restructured even if a packet loss has occurred. The 32-bit acknowledgement number is sent by the TCP
endpoint to inform the sending host that the data has been received successfully. The sequence and acknowledgement number are the two most important fields in the TCP header, and used to implement a reliable data transfer service. There are then some TCP flags in the header, but the main four used are the ones listed below.

- **SYN** - synchronizes sequence numbers to initiate the TCP connection
- **ACK** - acknowledges received data
- **RST** - resets the connection
- **FIN** - terminates the connection

### 2.3.3 TCP segmentation offload

TCP segmentation offload is a technique implemented in modern network interface cards (NIC) to reduce the overhead associated with transmission and reception on the host CPU. The overhead originates from a relatively small maximum transmission unit (MTU) defined in the Ethernet protocol of 1500 bytes, which requires packets larger than 1500 bytes to be segmented into multiple packets. These segments are then reassembled on the receiver end into the original packet. In order to offload the CPU from the process of segmenting and reassembling larger packets and consequently reduce the number of I/O operations to the NIC, these operations are accelerated in hardware on the NIC. This allows for the CPU to send packets larger than the MTU to the NIC. In Linux, the maximum packet size that can be sent to NIC offload is 64kB. The transmission and reception offloading are known as large send offload (LSO) and large receive offload (LRO) respectively.

### 2.3.4 UDP

UDP is another main protocol at the transport layer, which makes the computer system able to send messages to other hosts on the IP network. UDP is a connectionless protocol, meaning there is no handshake between the sender and receiver before sending a segment. An example of a protocol that uses UDP is Domain Name System (DNS). The advantages with using UDP is that it is faster because it does not require the three-way handshake. Another reason why DNS uses UDP is because DNS requests are generally small and they therefore fit inside an UDP segment. Another advantage with UDP is that it has only 8 bytes of overhead compared to TCP which has 20 bytes overhead in every segment.

The packet structure of an UDP header does not contain as many fields as TCP or IPv4 header, but the fields are shown in Figure 2.4.

<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>Checksum</td>
</tr>
</tbody>
</table>

Figure 2.4: UDP header.

The length field specifies the length of the UDP header and UDP body. The checksum is an optional field in IPv4 which may be used for error-checking of the header.
and data. This means the checksum determines if the bits within a UDP segment has been altered during the transfer from the source to the destination [43].

2.3.5 ICMP

ICMP is an error-reporting protocol network devices, e.g. routers, used to send error messages or other information when network problems prevent delivery of IP packets. An example is when the message Destination Network Unreachable is encountered in a FTP session. This message means that an IP router was unable to find a path to the host for the packet, and therefore sent the ICMP message instead to the client [27].

The ICMP packet is encapsulated in an IPv4 packet, which consists of a header and a body. The ICMP header structure is shown in Figure 2.5.

<table>
<thead>
<tr>
<th>Type</th>
<th>Code</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.5: ICMP header.

The type field contains the ICMP type for the different control messages. These are indicated by a value where, for example, type 3 indicates Destination Unreachable and type 11 indicates Time Exceeded. The code field gives additional context information to the type field [40].

The well-known ping program works by sending an ICMP type 8 code 0, which is the code for echo request message to a specific host. The host then sends back a type 0 code 0 which is the code for an ICMP echo reply. Attackers can easily send IP datagrams to a host and flood the servers with a deluge of ICMP packets. It is therefore important to have defense mechanisms to protect against this, and an IDS is one example of a defense mechanisms [27].

2.3.6 IEEE 802.11 protocol

IEEE 802.11 protocol is a standard for communication on wireless networks. For transmission of data in the physical layer, management and control of wireless links, 802.11 uses three different frames: management, control and data. Management frames are used to deliver authentication and association. One example of such management frames is the use of beacon frames, which are broadcasted regularly by the access point. These frames contain the Service Set Identifier (SSID) and information about the capabilities of the access point. The association frames are important in order to associate a Wi-Fi station with an access point [5, 34].

The control frames assist with the delivery of data frames as well as controlling the access to the wireless medium. Examples of control frames are Acknowledgement (ACK) frames, Request to Send (RTS) frames and Clear to Send (CTS) frames. RTS and CTS are primarily used to minimize the risk of collisions by reserving the transmission channel to make sure a data frame can be transmitted.

The data frames carry the packets from for example web pages and files in the frame body. An 802.11 data frame is shown in Figure 2.6.

---

2 An SSID is an ID used for naming wireless networks
2.4 Cryptography

In this section, the theory behind the cryptography used in wireless network communication is explained.

2.4.1 EAPOL authentication

Extensible Authentication Protocol (EAP) Over LAN (EAPOL) is a network authentication used in IEEE 802.1X. EAPOL operates on the link layer and these packets are used for 802.1X authentication when a device (supplicant) is connecting to an AP(authenticator). There are always four packets split up with different purposes and the four-way handshake is explained below [12, 20, 34]:

Before the first message, the supplicant generates a Pairwise Master Key (PMK) which is derived from the SSID and passphrase of the AP in an HMAC function. The authenticator derives the same PMK. The second requirement needed before communication commences is that the supplicant generates a random sequence Station Nonce (SNonce) and the authenticator generates a random sequence AP Nonce (ANonce). The authenticator sends the first EAPOL message encapsulated with the ANonce. With this nonce, the supplicant generates a Pairwise Transient Key (PTK) through a pseudo-random function in the format below:

\[
PTK = \text{PRF}(\text{PMK}, \text{“Pairwise key expansion”})
\]

\[
\| \text{min} (\text{AA, SPA})
\]

\[
\| \text{max} (\text{AA, SPA})
\]

\[
\| \text{min} (\text{ANonce, SNonce})
\]

\[
\| \text{max} (\text{ANonce, SNonce})
\]

The Frame Control is a 16-bit field containing flags depending on which type of frame it is, e.g. management frame or data frame. In the data frame, there are usually three addresses instead of two, which usually are source and destination address, but the third address in this frame comes from the infrastructure networks. There may also be a fourth address which is only present in data frames. Flags in the Frame Control field specify the role of the three different addresses. If the 802.11 frame is sent from the access point, address one contains the MAC address of the destination Wi-Fi device, address two contains the MAC address of the access point and address three is the MAC address of the server that sent the 802.11 data frame [10, 34].
The PTK is divided into Key Confirmation Key (KCK), Key Encryption Key (KEK) and a Temporary Key (TK). Then, the Message Integrity Code (MIC) is calculated with KCK from the PTK. When the MIC is calculated, the supplicant sends this with the SNonce to the authenticator in message two. The authenticator derives the PTK and re-computes the MIC on its side. It also computes the Group Temporal Key (GTK), which is a sequence number used to detect potential replay attacks on the client. The authenticator then sends message three with the MIC and GTK. Lastly, the supplicant verifies the MIC using the PTK and sends back an ACK to the authenticator with the MIC calculation. When all these messages are completed, the PTK can be used as the session key and unicast data can be transmitted [12, 20, 34].

2.4.2 RC4

RC4 is a symmetric stream cipher which is used in WEP and WPA. A general figure of a stream cipher is shown in Figure 2.7.

![Figure 2.7: Stream cipher.](image)

In a stream cipher, a keystream is generated by a stream of pseudorandom characters. This means it creates a new key depending on what has arrived in the message and therefore, the key is not the same every time. The keystream is able to encrypt messages with different length. Each plaintext symbol is encrypted one at a time. The circle in Figure 2.7 is usually an exclusive-or (XOR) operation. Since RC4 is a symmetric stream cipher, the same algorithm is used for both encryption and decryption. However, RC4 has been shown to be unsecure and not used in the newer IEEE 802.11 protocol versions [32, 18].

2.4.3 AES

Advanced Encryption Standard (AES) [55, 14] is a symmetric block cipher where one block is encrypted at a time. This is shown in Figure 2.8. Block ciphers may be faster than stream ciphers depending on the area of use but they do not have to be.

![Figure 2.8: Block cipher](image)

In AES, the block size is 128 bits while the length of the key is either 128, 192 or 256 bits. The encryption process consists of several rounds where each round contains
2.5. Algorithms and Data Structures

four sub-processes. The number of rounds depends on the length of the key, but for a 128-bit key length, AES uses ten rounds of these four processes explained below [55, 14].

1. SubBytes - If the key length is 128 bits, AES will treat it as 16 bytes and these bytes will be substituted by a substitution-box (S-box). This is used to hide the relationship between the key and the ciphertext.

2. ShiftRows - In this step, all bytes in each row are shifted by a certain offset to avoid every column of plaintext only affecting the same column of ciphertext.

3. MixColumns - The input bytes are confused where each of the input bytes affects all of the output bytes. This step provides diffusion, meaning every byte of the ciphertext depends on every byte of the plaintext.

4. AddRoundKey - In the last process of a round, the first key is XORed with each byte of the state. The reason this step is mandatory is that the block cipher is dependent on the key, otherwise, decryption is trivial because key material is never used.

AES is included in WPA2 security for wireless networks. Since AES is a symmetric block cipher, the same algorithm is used for both encryption and decryption. The decryption process consists of the four sub-processes but is done in the reverse order compared to the encryption process [55, 14].

2.5 Algorithms and Data Structures

2.5.1 Spatial data structures

Spatial data structures are data structures optimized for storing querying data represented by objects in multi-dimensional spaces. The structure consists of tuples representing spatial objects, and the queries in spatial data structures are called spatial queries. The benefit of using spatial data structures is that they allow representing objects as geometric data types, e.g. points and rectangles.

Geographic data is commonly stored and queried in spatial data structures for finding points of interest within a certain distance of a given point.

These data structures are not limited to storing geometric information but can also represent any data characterized by a single point in space or an n-dimensional volume.

In this thesis, the rules which the LPIDS uses are represented as a four-dimensional hyper-rectangles based on both the source and destination IP-address ranges and port ranges. This enables efficient filtering of rules which are relevant for a given packet, which in turn is represented as a four-dimensional point in the spatial data structure.

2.5.2 R-trees

An R-tree is a data structure used to store and access spatial data. The trees can organize any-dimensional data such as rectangles or points [19, 37].
An R-tree consists of root nodes, internal nodes and leaf nodes, as shown in Figure 2.9. The nodes in the tree represent a fixed-size collection of minimum bounding rectangles (MBR) which encloses the respective child node MBRs. Figure 2.10 is an example of MBRs corresponding to the nodes in the tree. The dotted lines are root nodes or internal nodes and the solid lines are leaf nodes which contain the data for the given spatial region. The root node holds a pointer reference to the largest region. The internal node contains a set of rectangles and maps pointers to child nodes. Leaf nodes store actual data by having rectangles of spatial objects or pointer to objects.

Searching in R-trees commences from the root node and is similar to B+-trees. First, each rectangle in a node is required to be checked to observe if it overlaps the search rectangle. If the rectangles overlap, the corresponding child node has to be searched. The searching is done recursively until all the overlapping nodes have been traversed. The tree is usually in a form that allows the search algorithm to ignore irrelevant regions of the tree and only examine data near the search area [19, 37].

Insertions in R-trees are done by recursively traversing the existing tree searching each traversed node for rectangles which encapsulate the new spatial object. When
a leaf node is reached, the new object is inserted. If the leaf node is full, the leaf is
split into two leaf nodes. This split and insertion of new nodes propagate upwards in
the tree and can cause parent nodes to split as well, all the way up to the root node.
The procedure of splitting overflowed nodes is the primary factor which influences
the search performance in the tree. Ideally, the optimal splitting procedure would
minimize the amount of overlap between rectangles as this would reduce the number
of sub-trees which are required to be visited. An optimal splitting procedure should
also retain the balance of the tree. However, finding this split configuration requires
an exhaustive search of all split combinations. Therefore, a heuristic is typically used
to determine an acceptable compromise between search and insertion efficiency. In
Guttman’s paper [19] a heuristic is proposed where the rectangles which require the
least enlargements are chosen. Brakatsoulas et al. [11] proposed another heuristic
using k-means clustering to discover an acceptable splitting solution. This heuristic
resulted in a more expensive insertion procedure but significantly reduced the search
time compared to the least enlargement heuristic.

2.5.3 Pattern Matching

Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt algorithm (KMP) [24] is a well-known string searching al-
gorithm, improving on the naive string searching algorithm where each character
input search-string $m$ is compared to each character in the input text $n$. The KMP
algorithm approach is to create a failure function for each character in $m$, defining
the number of characters which can be skipped during a failed partial match in $n$.
This technique results in no "backing up" while iterating $n$, i.e., no character in $n$ will
in the worst case be compared more than once and will, therefore, result in a linear
time complexity with respect to the size of $n$. There will be an added time complexity
when constructing the failure function which also has a linear time complexity
with respect to the size of the search string $m$, resulting in overall time complexity of
$O(\text{size}(n) + \text{size}(m))$.

Tries

A trie is a data structure used to represent a collection of strings by making it easier
to retrieve a string from a collection of strings [2]. Tries are useful when applications
are retrieval-based where the length of the keys varies. The performance of a trie
is dependent on the length of the key instead of the number of keys itself as it is in
search of trees [36]. An example trie with the keys Ball, Bat, Bal, Be, Cat and Rat are
shown in Figure 2.11. Each path from the root node to the leaf node corresponds to
one word. The character $\$ $ indicates the termination of a string, making sure no prefix
of a word can be a word itself.

Searching for a key $K$ in a trie $T$ begins at the root node, which is the branch node.
The branch node is a collection of fields pointing to a branch node or to information
node. An information node contains the keys. The first character of the key $K$ is ex-
tacted and moved down to the field corresponding to that character. This process is
done for all characters in the key and the path can easily be traced when the searching
2.5. Algorithms and Data Structures

is done. Searching for a prefix of B in Figure 2.11 would return 4 values: Ball, Bal, Bat and Be, while searching for a prefix of C would return the only word Cat.

**Aho-Corasick pattern matching algorithm**

Aho et al. [3] invented a string-searching algorithm which locates all occurrences of a finite set of keywords in a string of text. The algorithm is based on a finite state machine constructed from a set of given keywords $K$. Transitions between the states are defined by a goto-function $g(i,c)$, a fail-function $f(i)$ and a success-function $s(i)$ for all states $i$ in the state machine. The goto-function $g(i,c)$ defines the state transition to be made given a state $i$ and an input character $c$. The fail-function $f(i)$ defines the state transition to be made if the current character is not defined in the goto-function $g(i,c)$. Finally, the success-function $s(i)$ declares the states, which represents a pattern match.

Figure 2.12 and Tables 2.1 and 2.2 show an example of an Aho-Corasick state machine goto-, fail- and success-functions respectively for keywords $K = \{/bin/sh, /bin/dash, /bin/bash\}$.

![Trie example](image)

**Figure 2.11: Trie example.**

![Goto function](image)

**Figure 2.12: Goto function.**

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(i)$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 2.1: Failure function**

The state machine is typically represented as a trie data structure which enables constant time complexity when traversing the goto-function. The trie is extended
from the regular definition with suffix and dictionary transitions representing the fail-
and success-function respectively. These transitions create cycles within the goto-
function state graph to be used when an input symbol either does not match or is a
successful pattern match.

The complexity of constructing the state machine given a set of keywords $K$ is
equivalent of constructing a trie, that is, $O(|K| \cdot L)$ where $L$ is the average size of the
keywords. Since each transition in the state machine can be done in constant time, the
complexity of processing an input string of size $n$ is of complexity $O(n + \max(K))$.

In Snort, there are two different types of state machines in Aho-Corasick. The first
one is Deterministic Finite Automata (DFA), which requires exactly one transition of
a state machine to discover the correct state when examining an input character in
the text stream. The contrast to DFA is Non-Deterministic Finite Automata (NFA),
which can require more than one transition of a state machine to find the correct
state. The advantage with DFA is the capability of processing data faster than NFA,
but the disadvantage is that it has a more complex construction and structure of the
state table matrix compared to NFA [33].

Alternative Aho-Corasick representations
In order to adapt to various constraints provided by both hardware and software, al-
ternative approaches of representing the Aho-Corasick state machine are often used.
These representations are primarily focused on improving memory access patterns,
for improving efficiency and minimizing the memory footprint [29, 51]. Xu et al. [63]
evaluate the space-time tradeoff in the context of an IDS using a 2D state-transition
table against a basic state-transition with the goto-function represented as a trie and
the success- and failure-function as backlinks within the trie. Their results show an
increase in memory usage using a state-transition table in comparison to a basic im-
plementation but an increase in throughput from 15MB/s to 120MB/s in the best
case.

2.6 Related Work

2.6.1 IDS on Embedded Systems
Surendra et al. [31] conducted a study where they implement a Raspberry Pi based
Honeypot that simulates vulnerabilities and attracts attackers. For example, they use
IDS to detect attacks in the form of anomaly behaviour. Following that, they use the
Honeypot to gather as much information as possible about the attacker and in the
future remove security loopholes. They conclude that a Raspberry-Pi Honeypot is a
cost-effective mechanism to improve network security.
2.6. Related Work

The paper by Kyaw et al. [28] compares two open-source intrusion detection systems on a Raspberry Pi 2 and focuses on the performance and efficiency aspects. They compare Snort IDS and Bro IDS and conclude that Snort IDS puts less loads on the CPU for similar performance, as well as the packet loss rates, which were 4 and 19% respectively. The random access memory (RAM) usage was 23% for Snort and 6% for Bro. Both systems could detect simulated attacks, e.g. SYN Flood and MITM attacks. They also found out that a Raspberry Pi 2 cannot handle all network traffic because it could crash due to limited RAM when there are high network loads sent to the IDS, and this is the reason the packet loss rate was high according to the authors.

Sforzin et al. [49] evaluated Raspberry Pi’s performance while running Snort and if it can be served in a distributed system such as IoT. The evaluation was done by doing several tests with different rule sets of Snort and varying network traffic in multiple trace files. By measuring the workload of the device during the tests, the measurements showed that the Raspberry Pi could run Snort without the CPU being overwhelmed and without the memory reaching 100% utilization. The results motivate this thesis since Raspberry Pi could run Snort which is a pretty heavy open-source tool with many functionality options, it is likely a Wi-Fi Pineapple with similar hardware may be able to run a lightweight IDS making it a feasible solution.

2.6.2 Wireless IDS

Alipour et al. [5] present an anomaly-based intrusion detection system for wireless networks based on behavioural analysis. To generate a model for the network, the system observes the network during regular operation and extracts a sequence of consecutive state-machine transitions of the 802.11 protocol to form n-grams used for intrusion prediction. During runtime, the IDS monitors the real traffic sessions using the trained model to detect abnormal behaviour. The system is evaluated and compared to similar systems using a predefined rule set to detect intrusion. Due to these predefined rules, the other systems are not able to detect any zero-day attacks which the behavioural analysis based detection was able to. However, the 0.1% false positive error rate of the proposed system was higher than all other IDS tested.

In Kolias et al. [25] paper, an empirical evaluation of different statistical methods and feature extraction to detect malicious behaviour in wireless networks is conducted. In order to perform this evaluation, a dataset containing both intrusive and non-intrusive 802.11 traces were collected. The dataset is intentionally tailored towards intrusion detection methods and is made publically available. Several different methods to detect malicious behaviour on a network are investigated and compared in terms of accuracy, false positive rate and training time. In conclusion, the authors find that the most efficient method in this context is random decision forests.

The authors briefly discuss the advantages of statistical methods in an IDS but do not evaluate them in the context of misusing detection systems which other research deduces to be a superior alternative for detecting known attacks [5]. This thesis aims to evaluate and use a hybrid detection system using both rules-based and statistical-based methods to detect intrusion.

Mohammed et al. [45] developed a packet sniffer on the Linux platform to collect network packets being able to differentiate network attacks using network states and distribution of packets. The packet sniffer used the library libpcap in C to capture
packets and was set to promiscuous mode to receive all packets even though some may not be intended for this intrusion detection system. The authors discuss what filter can be used in the packet sniffer to be able to handle a large amount of network traffic to make sure that it does not lose packets. Their proposed solution to this is to put a filter called Low Pass Filter (LPF) early in the packet-processing chain inside the PF_PACKET protocol where the socket can receive packets directly from the network card driver. The filter then decides which packets should be discarded or not. The authors lastly discuss ways to detect the presence of a packet sniffer tool and explain both the ARP detection technique and SNMP monitoring.

Fayssal et al. [16] presents a Wireless Self Protection System (WSPS) which includes features such as monitoring the wireless network and anomaly analysis to detect wireless attacks. The system combines information from both the physical layer and link layer and is capable of monitoring the wireless network on multiple channels. To validate their proposed system, they conducted experiments with wireless attacks such as beacon and probe request flood attacks. The results from the tests showed that their system had a false positive rate of 0.1209% and a detection rate of 99%.

Sun et al. [56] provides an overview of intrusion detection techniques in mobile ad hoc networks (MANETs) and wireless sensor networks (WSN). They also survey the existing intrusion detection techniques for these two network types. In MANETs, it is difficult to distinguish between false positives and real intrusions because most dynamics in MANETs are caused by mobility. According to the authors, this may lead to less effective IDS. For both MANETs and WSNs, there exist key distribution and management schemes which may help to prevent DoS attacks and secure routing protocols. The authors conclude that IDS has become a significant security mechanism for both MANETs and WSNs.

Tsakountakis et al. [58] discusses major wireless attack categories concerning IEEE 802.11 family networks. They also propose a WIDS module to tackle these attacks as well as evaluating them. The major wireless attack is DoS according to the authors. They mention that there is no method to detect other attacks except DoS, but that a WIDS could identify them without preventing them.

2.6.3 IDS evaluation

Thongkanchorn et al. [57] investigates the performance of three different intrusion detection systems: Snort, Suricata and Bro, which are set up on a system running CentOS v5.0. To evaluate these systems and compare them, they used various attacks such as Scan port attack, FTP attack and SNMP attack. The results were then analyzed according to the performance metrics CPU utilization, the number of packets lost and the number of alerts. The results indicated that for each IDS, the CPU usage and the number of packets lost were low when TCP traffic was used. For UDP traffic, Suricata had the highest CPU utilization. The authors concluded that the results for each IDS vary depending on which type of simulated attack there is.

Alhomouda et al. [4] compared the performance of Snort and Suricata on three different platforms: ESXi (virtual machine), Linux 2.6 (Ubuntu10.10) and FreeBSD v8.1. The authors compared the efficiency and performance by measuring the percentage of packet drop and how often the IDS alerts on traffic. The conclusions drawn
are that Snort performs better than Suricata in terms of packet drops and speed. Meanwhile, for high traffic rates, Suricata performs best on Linux while Snort is best implemented on FreeBSD.

Paulauskas et al. [38] investigate the performance of Snort version 2.8.0 implemented on a Linux system. The report analyzes how the hardware affects the performance of Snort and also provides recommendations on how to maximize the performance on the hardware when running Snort. The authors concluded that hardware and techniques for logging alerts are the main factors that affect the performance. This work used tcpdump and libpcap as we also do but does not implement its intrusion detection system and compares with Snort.

2.6.4 Spatial Data Structures for Packet Filtering

Huang et al. [23] proposed an R*Tree based Bitmap Intersection algorithm by using spatial utilization of R*Tree and the fast matching speed of the intersection to improve the speed of the R*tree in each node to avoid having an $O(n)$ search speed. They also discussed the improvement of this algorithm using multiple R*Trees and Bloom Filter. The authors also evaluate the proposed algorithm in terms of the number of memory accesses and memory usage. They concluded that the algorithm consumed 300 KB of memory space and the use of multiple R*Trees improved the memory usage by 30%.

Srividhya et al. [54] conducted a comparative analysis of R-trees and R+-trees based on spatial querying of real estate data. The purpose of the study was to create an efficient mechanism for ranking real estate with respect to proximity to points of interest such as airports, hospitals and markets. The empirical results showed a logarithmic time complexity with respect to the number of objects stored in the data structures for both R-trees and R+-trees with a slightly lower constant factor for the R+-trees.

2.6.5 IDS pattern matching

Tuck et al. [59] examines string matching algorithms for intrusion detection. They provide modifications to the Aho-Corasick string matching algorithms to reduce the amount of memory required and improve its performance on hardware implementations by increasing the throughput. Their results from the modified Aho-Corasick reduced the memory required for the string sets used in NIDS by up to a factor of 50 while improving performance by more than 30%. The authors also presents techniques to enchance the worst-case performance of Aho-Corasick algorithm. According to the authors, Aho-Corasick is the only string matching algorithm that has deterministic worst-case lookup times and a friendly data structure to use.

Gurtov et al. [26] uses Host Identity Protocol (HIP) to implement a Wi-Fi authentication system that allows HIP clients to connect to a single HIP relay in a city-wide Wi-Fi system. This new distributed authentication architecture should secure wireless communications against DoS and Man-in-the-Middle (MitM) attacks. HIP offers end-to-end security and resistance to these attacks. The authors also observe that packet matching time on a Wi-Fi AP firewall takes $O(n)$ time in the worst case, and
$O(1)$ in the best case. They also suggest that an approach for efficient rule matching is using tries or ternary trees, and in the LPIDS, tries are used.

Abbes et al. \cite{1} proposed a method combining a novel protocol analysis approach with traditional pattern matching to improve the performance of pattern matching when examining for attack signatures. They applied a decision tree which is adaptive to the network traffic characteristics, resulting in less number of false positives. Their method presents an extra cost for decoding the protocol, but the overhead of preprocessing is balanced by a faster detection, immunity against evasion attempts and significant search space reduction. This method is better when the number of rules increases.
3 Method

In this section, the proposed LPIDS is explained in detail. This includes cross-compilation, setup, initialization and information on how to run the module with specific rule formatting requirements.

3.1 Buildroot

The default configuration of the Wi-Fi Pineapple platform does not include a C/C++ compiler. In order to deploy the LPIDS on the system, cross-compilation for the Wi-Fi Pineapple architecture was necessary. However, setting up the required environment for compilation on a host, i.e. the machine which is performing the compilation, is an intricate process. There exist several tools which simplifies the process and for this thesis, Buildroot was used.

Buildroot is an open-source tool created under the name uCLinux which was made as an effort to port the Linux kernel to microprocessors without a memory management unit [52]. The project was split into two different projects, uClibc and Buildroot, with Buildroot focusing on simplifying the distribution of the Linux kernel on embedded systems.

In summary, Buildroot aids the user in generating the following components for cross-compilation:

- Cross-compiler toolchain targeting a specified architecture. The toolchain typically includes the compilers gcc and g++ for C and C++ respectively. Further it includes the linker ld.

- A target architecture Linux filesystem.

[https://buildroot.org/](https://buildroot.org/)
3.2 Implementation

The implementation of the LPIDS is divided into six parts as listed below.

- Setup and initialization of the network interface used in the monitoring
- Interpreting user-defined detection rules
- Protocol parsing
- Decryption
- Rules-based filtering
- Pattern matching of packet payloads

The system was implemented in C++17 using the standard C++ template library (STL) to the largest extent possible. Some functionality of STL had to be replaced with functions from the C standard library due to limited support from the Wi-Fi Pineapple platform. The functionality mainly concerns `std::filesystem`, `std::stringstream` and `std::fstream`. The filesystem library provides facilities for performing operations on the file systems, which includes directories, paths and files. Stringstream provides functionality for formatted I/O operations. Fstream has capabilities of creating files, writing information to files and reading information from files.

Compilation of the source code for evaluation was done using GCC-MIPS version 7.3.0 for the Wi-Fi Pineapple and GCC-ARM version 8.1.0 for the Raspberry Pi. Additionally, the highest level of code optimization was used, that is, the optimizer flag `-O3` was set. There are however possible drawbacks of using the highest level of optimization as it includes aggressive unrolling of loops which could lead to a degradation in performance, especially in hardware platforms that feature a smaller size instruction cache. During the initial tests, no such degradation was observed and therefore, the highest level of optimization was used during the remainder of the tests.

3.2.1 Setup and Initialization

The initialization attempts to switch a given network interface to monitor mode and prepares the underlying driver for network capture. Upon successful initialization, the device observes the network for beacon broadcasts containing the required information such as the access point’s MAC-address, SSID and channel frequency. These

beacons are broadcasted on all available channel frequencies defined by 802.11 standards. Hence, it is possible for a client to identify information about the available access points without knowledge of the frequencies being used. The beacons sent by the access points are used by the LPIDS to acquire the MAC-address and channel frequency based on a given target SSID. These steps are illustrated in Figure 3.1. Once acquired, the LPIDS monitors all traffic which is either sent or received by the target access point. At this stage, the LPIDS is fully configured and initialized. Figure 3.2 illustrates the main packet capture loop of the LPIDS from capturing the packets until a potential alert occurs.

![Figure 3.1: The initialization of the LPIDS.](image)

![Figure 3.2: The main packet capture loop of the LPIDS.](image)
3.2. Implementation

3.2.2 Rule Parsing

Upon startup of the LPIDS, the configuration and rules are read, syntactically verified and parsed. After the rules have been parsed, they are inserted into an R-tree with respect to the source and destination IP-address as well as source and destination port, forming a hyper-rectangle spatial object. If any of the source or destination IP addresses are inverses of a defined IP range that is declared in the configuration file, they will be split into multiple rectangles in the R-tree. The reason for splitting the rectangles is that inverses of a dimension in the R-tree cannot be represented as a rectangle and it is, therefore, necessary to insert multiple rectangles to represent this shape.

After all the rules have been parsed, the rules which have a content parameter\[^{5}\] are used to build the Aho-Corasick state machine. Later on, this is used for pattern matching within a received packet.

Rule Configuration

The rules defined for the LPIDS are placed in a rule directory similar to Snort. The LPIDS supports different actions, protocols, IP address format, ports and operators. The supported ones are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Actions</th>
<th>Protocols</th>
<th>IP addresses</th>
<th>Ports</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>alert</td>
<td>ICMP</td>
<td>$EXTERNAL_NET</td>
<td>80</td>
<td>==</td>
</tr>
<tr>
<td>log</td>
<td>HTTP</td>
<td>$HOME_NET</td>
<td>1:65535</td>
<td>=</td>
</tr>
<tr>
<td>pass</td>
<td>TCP</td>
<td>192.168.0.0/16</td>
<td>1000:</td>
<td>&gt;=</td>
</tr>
<tr>
<td></td>
<td>UDP</td>
<td>any</td>
<td>any</td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td>IP</td>
<td>any</td>
<td>any</td>
<td>&lt;=</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>!=</td>
</tr>
</tbody>
</table>

Table 3.1: Supported rule formats.

Instead of writing multiple rules for the different IP addresses, the LPIDS can parse some defined variables, e.g. **EXTERNAL_NET** and **HOME_NET**, which is a subnet of IP addresses. These subnets are defined in the LPIDS configuration file. In this file, it is also possible to define **EXTERNAL_NET** as all IP addresses that are not included in the **HOME_NET** by defining

\[
\text{ipvar EXTERNAL_NET }!\$HOME_NET
\]

It is also possible to utilize `portvar` for example to define **HTTP_PORTS**. By having this functionality, the amount of configured rules can be decreased, which makes it easier for the LPIDS to progress and filter.

The rules also contain different parameters depending on what content should match the packet data and what message should be displayed. An example rule that works in this LPIDS is the following one:

\[^{5}\]Content parameter is a keyword used to search for specific content in packet payload and trigger alert based on that data.
3.2. Implementation

```plaintext
alert http $EXTERNAL_NET any -> $HOME_NET any
(
    msg      : "Hacker is here!";
    dsize    : > 100;
    type     : "response";
    content  : "Bootstrap";
)
```

The `dsize` parameter is the length of the payload data meaning that this rule matches if the payload contains the word `Bootstrap` and if the payload response is larger than 100 bytes. The rule will only alert if all the parameters are satisfied. Another available parameter is the flag parameter 6. The flag parameter only alerts if it matches both the SYN and FIN flag for example. Inside the content parameter, it is possible to write which hex bytes it should match, for example placing the hex bytes of shellcode 7 to detect if someone is endeavouring to exploit a binary. An example of how the content parameter may be configured as shown below.

```plaintext
content : |90 90 90 E8 C0 FF FF FF|/bin/sh
```

If this parameter is in a rule, it would only alert if that exact hex byte sequence is present in the packet bytes followed by the string `/bin/sh`. The vertical bar character (|) toggles between hex bytes and ASCII, which makes it easier to parse the rule and interpret the contents of this parameter.

### 3.2.3 TCP Reassembly

As discussed in Section 2.3.3, due to the MTU of the Ethernet protocol and similarly to the 802.11 protocol, TCP packets larger than 1500 bytes need to be segmented before being transmitted on the network. In effect, in order for the LPIDS to correctly interpret larger packets, it requires a software implementation equivalent to large receive offloading on NICs.

Figure 3.3 demonstrates an overview of the TCP reassembly implementation in LPIDS. The procedure begins by observing the headers in the TCP packet. If the header has both the SYN and ACK flags set, it means the packet is the second step in the TCP three-way handshake, and a new TCP stream is initiated and stored. If either the FIN or RST flags are set, it signifies the end of the stream and the reassembled packet stream is processed as a single packet by the IDS. Finally, if none of the flags are set it means the incoming packet should be a part of an ongoing TCP stream. If such a stream is stored in the internal state, the payload is appended to the stream based on the relative sequence numbers of the packet to maintain the order of the payload. Note that in all outcomes, the packet is still processed as a single TCP packet denoted as "stateless" in the rule sets.

Some application layer protocols embeds multiple application messages within a single TCP stream, e.g. HTTP. When reassembling TCP targeting such protocols there

---

6 Flag parameter is a keyword used to discover which flag bits are set inside the TCP header of a packet

7 Shellcode is a piece of assembler code used as the payload in the exploitation of a software vulnerability.
exists additional information in the payload on how the packets are segmented. In the case for HTTP, this information is signified by the HTTP Content-Length header.

![TCP reassembly functional diagram.](image)

**Figure 3.3: TCP reassembly functional diagram.**

### 3.2.4 Protocol Parsing

The parsing of the different protocols is done by examining the different fields and layers of packets in Wireshark. It works by reading the bytes in the packet from top to bottom, and if some bytes are not of interest or mandatory to parse a protocol, the program can skip reading those. When parsing IPv4 packets, the first step is to determine the protocol used. Figure 3.4 demonstrates the fields used in an IPv4 packet in Wireshark.

![IPv4 parsing.](image)

**Figure 3.4: IPv4 parsing.**

By knowing that the length of an IPv4 header always is 24 bytes, it is possible to calculate where fields are located. The length of the header field is important for the rest of the parsing. The number inside the parentheses for the protocol field is unique per protocol. This means the hex byte is read, and with that information, it is possible to determine which protocol is used. Similarly, when parsing the TCP protocol, the length of the header field is always 20 bytes, and with this information it is possible to parse the different fields required, for example, TCP flags.

Similarly, for HTTP parsing, the program first checks if it has received all of the streams and that it is completed. If so, the stream processing commences. An
3.2. Implementation

HTTP message is defined by having specific headers where the first four characters are HTTP. Once the HTTP request method has been parsed, it then parses all the headers in the request. The Content-Length header indicates the length of the request or response body. Therefore, this header is parsed to know when the HTTP stream terminates, which is essential information for the TCP reassembling. The Content-Encoding header indicates which encoding has been applied to the body. The supported HTTP encoding for the LPIDS is gzip, which is a standard compression method for HTTP requests and responses.

The parsing of IEEE 802.11 is done by first parsing the header. Figure 3.5 displays part of an IEEE 802.11 frame.

<table>
<thead>
<tr>
<th>IEEE 802.11 Beacon frame, Flags: ........C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type/Subtype: Beacon frame (0x0008)</td>
</tr>
<tr>
<td>.... .. = Version: 0</td>
</tr>
<tr>
<td>.... 00.. = Type: Management frame (0)</td>
</tr>
<tr>
<td>1000 .... = Subtype: 8</td>
</tr>
<tr>
<td>Flags: 0x00</td>
</tr>
<tr>
<td>.000 0000 0000 0000 = Duration: 0 micros</td>
</tr>
<tr>
<td>Receiver address: Broadcast (ff:ff:ff:ff:ff)</td>
</tr>
<tr>
<td>Destination address: Broadcast (ff:ff:ff:ff:ff)</td>
</tr>
<tr>
<td>Transmitter address: Cisco-LL_B2:b2:55 (00:0c:41:82:b2:55)</td>
</tr>
<tr>
<td>Source address: Cisco-LL_B2:b2:55 (00:0c:41:82:b2:55)</td>
</tr>
<tr>
<td>BSS Id: Cisco-Li_B2:b2:55 (00:0c:41:82:b2:55)</td>
</tr>
</tbody>
</table>

Figure 3.5: IEEE 802.11 frame.

The important part of this frame is the frame control field indicating one of the types: control, management or data. The control and management frames contain beacon frames, probe responses and acknowledgement frames, which are not of interest to our LPIDS. Therefore, if these frames arrive, the program skips reading them. The frames parsed and processed are the data frames. This is done by first checking what frame control field the packet contains. If it is data or Quality of Service (QoS), fields such as ethernet source and destination address are extracted. Afterwards, if a passphrase has been provided, and all four EAPOL packets have arrived, the LPIDS decrypts the data packets.

3.2.5 Decryption

If a password for the target access point is provided, the LPIDS calculates the pairwise master key to be used for decrypting packets on the network. Then the LPIDS decrypts the EAPOL packets according to the formulas explained in Section 2.4.1. As mentioned previously, the EAPOL is a four-way handshake. To parse these packets, the LPIDS first checks if the key descriptor in the 802.1X Authentication frame is 0x02, which is the identifier for an EAPOL RSN key. This verifies the use of the EAPOL protocol. Following that, the key information value is extracted to differentiate between the four different EAPOL packets. This is shown in the listing below.

```cpp
1 if (key_descriptor == 0x02) {
2     uint16_t key_information = Endian::be_to_host
```
3.2. Implementation

(p.get<
uint16_t
>();

switch(key_information)
{
  case EAPOL_1:
    client->process_eapol_1(static_cast<WPAEncryption>
    (key_information & 0x7), nonce, flags);
    break;
  case EAPOL_2:
    client->process_eapol_2(nonce, flags);
    break;
  case EAPOL_3:
    client->process_eapol_3(p.get_buffer() + eapol_begin,
    eapol_length, mic, flags);
    break;
  case EAPOL_4:
    client->process_eapol_4(flags);
    break;
}

On line 7 in the listing above, the three least significant bits are extracted because these contain what cryptographic algorithm has been used. With this information, depending on if it is AES or RC4 that has been used, we can determine if the traffic is WPA-2, WPA or WEP. The process_eapol_1 method is then used to create a new state for every client and store the intercepted nonce generated by the access point. In process_eapol_2, the nonce generated by the client is stored. Finally, in process_eapol_3, the MIC sent from the access point is intercepted and the LPIDS uses the previously intercepted nonces to calculate the PTK for the connection. The PTK is verified by comparing the computed MIC with the MIC that was intercepted from the access point.

3.2.6 Rule Based Filtering R-trees

The parsed rules are stored in R-trees with respect to the specified protocol and potential encapsulated protocols (e.g. TCP being encapsulated in an IP packet) they are defined for. Once the LPIDS has extracted the protocol, source and destination IP-addresses and ports from a captured packet, it queries the corresponding R-tree for the protocol using the IP-addresses and ports in the packet. This code is shown in the listing below.

```cpp
void ids_rule_check_common(std::list<RulePtr>& rules,
                        const ip_address& src, const ip_address& dst, uint16_t port_src,
                        uint16_t port_dst, Rule::Protocol protocol){
  auto& ruleset = g_rules[protocol];

  uint32_t point[] =
  {
```
3.2. Implementation

```c
ip_address_to_uint32(src),
ip_address_to_uint32(dst),
port_src,
port_dst
};
```

```
14
15
ruleset.Search(point, point, [&rules](const RulePtr& r){
    rules.push_back(r);
    return true;
});
```

The rules which match these parameters are then further tested for optional constraints for detection such as TCP flags and payload length.

3.2.7 Content Matching

The final step in matching a captured packet is to match the content parameters of the rules which remain after the previous filtering. Each rule which satisfies all other constraints is matched against the packet payload. The payload is processed in the Aho-Corasick state machine to determine if any of the rules' content parameters match the payload. If the specified content is found within the packet payload, a user-defined action is performed, e.g. outputting a message to the command line. In this thesis, an open-source implementation of the Aho-Corasick algorithm was used. The state transition function is implemented as an interleaved array, increasing the data locality and thus the performance of the algorithm. The packet pattern matching source code is presented in the listing below.

```c
MEMREF text;
text.ptr = reinterpret_cast<const char*>(payload.data());
text.len = payload.size();

acism_scan(g_state_machine, text,
    [](int strnum, int, void* ctx){
        MEMREF pattern = reinterpret_cast<MEMREF*>(ctx)[strnum];
        Rule* r = reinterpret_cast<Rule*>(pattern.value);
        r->trigger();
        return 0;
    }, const_cast<MEMREF*>(g_patterns.data()));
```

For all of the rules, it extracts the parsed content. If the rule has a content parameter, it creates the state machine for the KMP algorithm. It then checks for the first match and triggers if it matches. Therefore, it does not care how many matches there are, only if there is one match or not. If the rule does not have any content parameter,
it is expected to trigger because all other checks have been passed, for example, protocol and IP checks. The content matching routine for the KMP algorithm is shown in the listing below.

```cpp
for (const auto & r : rules){
    auto content = r->get_compiled_content();
    if (!content.empty()){
        kmp::pattern<std::basic_string<uint8_t>::iterator>
        kmp(content.begin(), content.end());
        if (kmp.match_first(payload.begin(), payload.end())){
            r->trigger();
        }
    } else{
        r->trigger();
    }
}
```

### 3.3 Comparative Analysis

#### 3.3.1 Throughput

The first metric to be evaluated for LPIDS was the throughput of the system. That is, how many packets the system can process per unit time for a varying number of detection rules. The throughput characteristics of the system are of high importance since it could be possible to essentially perform a DoS attack on the system in order to bypass detection. A system with a higher throughput would decrease the probability of such an attack to some extent.

To calculate the throughput, the management frames and control frames are not included in the total number of packets. The throughput is calculated as follows:

$$T = \frac{\text{Number of data packets}}{\text{Sampling interval}}$$  \hspace{1cm} (3.1)

#### 3.3.2 Power Consumption

The power consumption is calculated using a USB-multimeter to approximate the power used during the LPIDS evaluation. The voltage and current values are extracted when the IDS runs for 30 seconds. The power is then calculated using Equation 3.2, where $I$ is the current and $V$ is the voltage.

$$P = I \cdot V$$  \hspace{1cm} (3.2)

This is then tested by applying different throughput values. The different throughput values used for both hardware components are shown in Table 3.2. The range of throughput values is different between the devices due to the maximum capacity of the respective platforms.
3.3. Comparative Analysis

<table>
<thead>
<tr>
<th>Wi-Fi Pineapple</th>
<th>Raspberry Pi</th>
</tr>
</thead>
<tbody>
<tr>
<td>300 P/s</td>
<td>1000 P/s</td>
</tr>
<tr>
<td>350 P/s</td>
<td>1500 P/s</td>
</tr>
<tr>
<td>400 P/s</td>
<td>2000 P/s</td>
</tr>
<tr>
<td>450 P/s</td>
<td>2500 P/s</td>
</tr>
<tr>
<td>500 P/s</td>
<td>3000 P/s</td>
</tr>
<tr>
<td>550 P/s</td>
<td>3500 P/s</td>
</tr>
<tr>
<td>600 P/s</td>
<td>4000 P/s</td>
</tr>
<tr>
<td>650 P/s</td>
<td>4500 P/s</td>
</tr>
</tbody>
</table>

Table 3.2: Throughput values.

3.3.3 Response Time

The response time is calculated as the meantime between when the packet is sent and when an alert has been triggered. It is the meantime between each interval, for example between 0 and 500 rules there is one meantime, and between 500 and 1000 rules, there is another meantime.

3.3.4 Rule Filter

The LPIDS uses R-trees to enhance the performance of filtering rules based on the headers of the incoming packets. To evaluate the implementation and the potential advantages of using spatial data structures for packet filtering, tests were done using both R-trees and linear search for filtering.

3.3.5 Snort Comparison

To obtain a comparison of our proposed LPIDS on the Rasberry Pi hardware, a performance evaluation against the open-source IDS Snort version 3 was conducted. However, there are a few key differences in the features and implementation details in Snort in relation to LPIDS which limits the possibilities for objective and fair comparisons.

The main difference is that Snort is a NIDS and does not offer functionality for wireless intrusion detection. While there is additional computation required for WIDS such as decryption and handling of encryption states, it can be observed as negligible with respect to scaling of the system, i.e. how the amount of rules affects system performance.

In order to perform the same tests on Snort as on LPIDS, the packet capture files used for evaluating LPIDS are made compatible with Snort by decrypting, stripping the 802.11 control and management frames and replacing the headers with 802.3 Ethernet headers.

Snort also implements rule filtering differently from LPIDS. According to the user manual\footnote{Snort user manual: https://snort.org/downloads/snortplus/snort_manual.pdf}, Snort evaluates rules in a two-step process including fast pattern search and a full evaluation of a rule signature. This two-step process means the following: Once a packet is received, the payload is matched to the content parameters of the
given rules using Aho-Corasick. This step is referred to as the fast pattern search. The second step is the full evaluation of rule signatures, which is a linear search of the rules which either passed the pattern matching in the previous step or did not have a content parameter. Snort then checks if the remaining rules satisfy the other constraints such as IP-addresses and ports.

### 3.3.6 Packet Capture File and Rules

To generate data and rules used in tests, Python scripts have been created. In order for the rule file to simulate a realistic scenario, the following five protocols were included: IP, ICMP, TCP, UDP and HTTP. The IP range varies to generate different subnets. The ports are also randomly generated between 1 and 65,535. The content parameter is generated by a random sequence of ASCII letters combined with digits.

A Python script to disable different rules has also been created in order to generate different intervals of rules to plot graphs.

The packet capture file used in this thesis has a size of 38 MB and contains varied types of network traffic. The wireless traffic is encrypted with WPA-2. The distribution of the IEEE 802.11 frame types is demonstrated in Figure 3.6. The sum of the management, control and data frames is 115,116 packets which are the total number of packets in the packet capture file. The protocol distribution from the packet capture file is shown in Figure 3.7.

![Trace file frame types](image)

**Figure 3.6: IEEE 802.11 frame type distribution**

In the following chapter, the results of the evaluation of LPIDS are presented. To obtain a better understanding of what throughput requirements are necessary for a system in order to process all captured packets, the histogram of the packet capture file is presented in Figure 3.8. In the worst case, an IDS is required to process 2500 packets per second, which are reflecting a file download during the capture of our evaluation packet file.
3.3. Comparative Analysis

Trace file data packet types

2382 (5.96%) UDP
36880 (92.35%) TCP
525 (1.31%) HTTP
149 (0.37%) Others

Figure 3.7: Protocol distribution

Trace file packet histogram

Packets

Time [s]

Figure 3.8: Histogram of the packet data used for evaluation.
In this section, the results of the comparative analysis are presented and displayed along with graphs for the different metrics measured on the different hardware.

4.1 Throughput

The throughput defined in Chapter 3.3.1 is displayed in Figure 4.1. The Wi-Fi Pineapple has lower throughput in the range 400 to 900 data packets per second compared to Raspberry Pi, which has a range between 2500 to 6000 data packets per second.

The maximum throughput on Wi-Fi Pineapple for Aho-Corasick measured approximately 905.79 packets per second, while the maximum throughput for KMP were 883.73 packets per second. On the Raspberry Pi, the maximum throughput for Aho-Corasick were 6227.43 packets per second, while it was 6173.05 packets per sec-

![Figure 4.1: Throughput comparison per number of rules.](image-url)
4.2 Content Matching Time

Figure 4.2 shows the average time for pattern matching, meaning how long it takes on average to match the content of one packet. Therefore, the best value is the lowest value, which on Wi-Fi Pineapple for Aho-Corasick is 37.64 µs, which happens at 1500 rules. The best value for KMP is 33.90 µs which occurs in the commence at 500 rules. The point of intersection of the two lines on the Wi-Fi Pineapple is approximately 600 rules and 35 µs.

On the Raspberry Pi, the best value for Aho-Corasick is 13.16 µs at 500 rules, while the lowest value for KMP is 10.0 µs at 500 rules. The intersection point of the two lines is approximately 750 rules and 13.50 µs.

4.3 Response Time

Figure 4.3 displays the response time for both pieces of hardware. The lowest response time for Aho-Corasick is 170.83 µs at 1000 rules, while the lowest for KMP is 134.30 µs at the beginning at 500 rules. Because the lines are very similar, the response time does not depend on which algorithm is used.

The lowest response time on the Raspberry Pi for Aho-Corasick is 23.93 µs at 500 rules, while it is 24.14 µs for KMP at 500 rules as well. As the y-axis shows the delay in microseconds, the difference between Aho-Corasick and KMP on the Raspberry Pi is minimal, confirming that the response time is not dependent on an algorithm.

4.4 Rule Filter Performance

The comparison of using R-trees and linear search for filtering packets based on IP-addresses and ports are shown in Figure 4.4. On both hardware platforms we observe
4.5 Snort vs LPIDS

Figure 4.5 shows the throughput comparison between Snort and our developed LPIDS on the Raspberry Pi when either Aho-Corasick or KMP is used. As observed in the figure, Snort performs better when the number of rules containing content parameter increases. The LPIDS remains nearly constant with Aho-Corasick and slightly decreasing with KMP. The point of intersection between Snort and KMP is approximately at 9200 rules with a content parameter corresponding against a throughput around 2200 data packets per second. The point of intersection between Snort and Aho-Corasick occurs at approximately 9600 rules with a content parameter corresponding to a throughput of 3214 data packets per second. Figure 4.6 shows the variation in the workload distribution with respect to the number of rules which have a linear increase in time with respect to the number of rules with both search techniques. The evaluation does indicate an increase in performance using R-trees on both hardware platforms.

![Graphs showing response time and lookup time](image-url)
content parameter. We observe that the percentage of the workload occupied by full evaluation rapidly increases with fewer content parameters and becomes the dominant part of the computation when 9000 out of 10000 rules has a content parameter.

![Figure 4.5: Throughput per number of rules with content parameter on Raspberry Pi 3 Model B+.

![Figure 4.6: Workload distribution of Snort with varying number of rules with a content parameter.](image)

Figure 4.7 displays a comparison between the content matching on Raspberry Pi using two different Snort algorithms: Aho-Corasick Binary Non-Deterministic Finite Automata (AC-BNFA) and Aho-Corasick Full. These two algorithms available in Snort are compared to Aho-Corasick and KMP implemented in LPIDS. As shown in the figure, AC-BNFA is less efficient than the others except KMP in the end when...
many rules are matching. KMP is the only one that exhibits a decreasing performance when the number of rules matching increases, while the other algorithms show nearly constant behaviour. It should be noted that these values reflect the average time spent matching patterns against packet payloads with a certain amount of rules loaded. In Snort, all loaded patterns are matched against the payload while in LPIDS, some patterns have already been filtered by IP addresses and ports and thus have potentially fewer patterns being searched.

Figure 4.8 show how the different computation parts in Snort varies with the number of rules loaded. Since the generated test rule set is randomly generated, there are no payloads matching any of the given rules which is an ideal rule set for Snort and thus no full evaluations are needed. The content matching computations show a constant behaviour with respect to the number of rules. The dominant factor for the performance is protocol parsing which occupies more than 50% of the computational workload.

![Graph showing content matching on Snort between AC-BNFA and AC-FULL.](image)

**Figure 4.7**: Content matching on Snort between AC-BNFA and AC-FULL.

### 4.6 Power consumption

Figure 4.9 displays a comparison of the power consumption between Wi-Fi Pineapple and Raspberry Pi using Aho-Corasick and KMP algorithm. On Wi-Fi Pineapple, KMP behaves more constant compared to Aho-Corasick behaving more inconsistent with low and high peaks. On Wi-Fi Pineapple, the lowest power for Aho-Corasick is 3.76 W corresponding to 450 data packets per second. The lowest power for KMP is 4.79 W corresponding to 300 data packets per second.

There are more similarities between the algorithms on the Raspberry Pi, which only differs a little during the throughput of 3000 and 3500 packets per second. The lowest power for Aho-Corasick is 2.34 W at 1000 data packets per second, while the lowest power for KMP is 2.37 W at 1500 data packets per second.
4.7 System Scaling

The scaling of the combined system with respect to the number of rules is shown in Figure 4.10. The five major influences on the system performance are rule filtering, content matching, protocol parsing and decryption. Figure 4.10 shows how these parts of the system scale with respect to the number of rules loaded into the system. The area marked as “other” is the combined computation workload of the parts not explicitly listed such as packet I/O. During these tests, the Aho-Corasick algorithm was used for content matching. On both platforms we observe that the R-tree filter is the main influence on system scaling as the number of rules increases. The other parts of the system remain constant. At the maximum number of rules tested, the rule independent computations, i.e. protocol parsing and decryption, occupies around 50% of the workload.
4.7. System Scaling

![Graphs showing LPIDS workload distribution per number of rules for different devices: Wi-Fi Pineapple and Raspberry Pi 3 Model B+](image)

Figure 4.10: LPIDS workload distribution per number of rules.
5 Discussion

In this chapter, the data from the tests are analyzed. It also discusses and criticizes different aspects of the method presented in Chapter 3.

5.1 Results

The result of the tests was presented in Section 4. The test data consisted of 10,000 rules, which were at intervals of 500 rules between each step. This number of rules was considered to be sufficient to draw conclusions about the different algorithms on both hardware platforms. For replicability and consistency, all evaluation data from the graphs were tested using a packet capture file instead of live tests when capturing network traffic. The same packet capture data was used in all tests.

5.1.1 Throughput

When analyzing the throughput of the different algorithms, there are several aspects worth discussing. Firstly, on both hardware platforms, the throughput exhibits the same decreasing behavior as the number of rules increase. These are the expected results because when the number of rules increases, the algorithm has larger R-trees to process and therefore, more information to check if it matches against the traffic. In the beginning when there are fewer rules, it is not clear which algorithm has higher throughput. After 2000 rules, Aho-Corasick has higher throughput than KMP on the Wi-Fi Pineapple. On the Raspberry Pi, Aho-Corasick has continuously higher throughput after 1000 rules.

Secondly, KMP has similar throughput to Aho-Corasick on both pieces of hardware in the beginning when there was not a large number of rules. Thirdly, the difference in throughput between the algorithms on Raspberry Pi is much more illustrative, making it reasonable to assume that it could be because Aho-Corasick is just faster when the hardware is more powerful.
5.1. Results

Lastly, the throughput is considerably higher on the Raspberry Pi, which is expected due to the different hardware capabilities between Wi-Fi Pineapple Tetra and Raspberry Pi 3 Model B+. When comparing Aho-Corasick and KMP implemented on the LPIDS with Snort in terms of throughput per number of rules with the content parameter, there are significant differences in the graph. Snort scales up better than LPIDS, especially with respect to the number of content-parameter rules. The reason for this is that Snort is built around a two-step process where the first step is a fast pattern search, run in $O(1)$ time, which is the key to Snort’s speed. Without a content match, Snort cannot build a fast pattern tree to help cut down on the number of rules evaluated. The second step of the filter process in Snort is the full evaluation, which is more costly because it requires to filter for example IP addresses, protocols and ports. This makes it very slow when there are many rules without a content parameter. Compared to our LPIDS, it consists of one R-tree for each protocol and filters out the source and destination IP addresses as well as source and destination ports. The R-tree runs in $O(\log N)$ in the best case which makes it filter quickly.

Figure 5.1 and 5.2 demonstrates a simplified overview of the steps of detection in LPIDS and Snort respectively. The main difference, which is reflected in our results, is the early filtering of rules using R-trees, reducing the number of rules to be fully evaluated and processed based on the payload of a packet. In Snort, the first step of matching a rule signature is to match it against the packet payload using the Aho-Corasick algorithm. If a rule does not have a content parameter, the entire signature evaluated, which reduces the performance significantly as demonstrated in Figure 4.5.

However, the practical relevance for the potential inefficiency presented in Snort is limited. Inspection of the freely available Snort community rules shows that less than four percent of the 3505 available rules do not contain a content parameter.

![Figure 5.1](image1.png)

**Figure 5.1: Overview of packet analysis in LPIDS.**

![Figure 5.2](image2.png)

**Figure 5.2: Overview of packet analysis in Snort.**
5.1. Results

5.1.2 Content Matching

The content matching in Figure 4.2 also has similarities in the way that the lines in both graphs do not differ too much. Aho-Corasick has higher content matching time in the beginning for a few numbers of rules however then KMP increases linearly while Aho-Corasick retains nearly a straight line. The linear behaviour KMP is showing is intended because KMP should behave linear time with the number of rules or patterns. The results show that on Wi-Fi Pineapple, KMP is faster than Aho-Corasick for a small number of rules. However, after that, it is more suitable to use the Aho-Corasick algorithm. Similarly, on Raspberry-Pi, KMP is faster at the beginning up to 750 rules and after that, Aho-Corasick is much faster. A potential reason for this is that KMP is a simpler algorithm than Aho-Corasick and requires less processing in the beginning. The mean time for content matching on Wi-Fi Pineapple is higher than on the Raspberry Pi, and the reason for this is that the processor in Wi-Fi Pineapple Tetra has 533MHz clock rate while Raspberry Pi 3 Model B+ has a 1.4GHz clock rate.

The content matching in Figure 4.7 shows mostly expected results where KMP is linear according to the number of rules, which is intended and explained in Section 2.5.3. According to the Snort user manual, AC-Full should be the fastest algorithm available in Snort but uses the most amount of memory. In our results we observe that it is faster than AC-BNFA, which is consistent with that of the user manual.

5.1.3 Response Time

The observation of the response time is that both of the graphs are expected. The response time for the Raspberry Pi differs very little however in general, they are similar in both graphs. This means that it does not matter which algorithm is used if response time is important because both of them are nearly the same. However, there is a large difference in response time between the hardware platforms itself, which is also expected due to the difference in hardware capabilities. In general, KMP is faster at the beginning, which shows similar behaviour as other graphs where KMP performs better in the beginning compared to Aho-Corasick.

5.1.4 Power Consumption

The results from the power consumption measures in Figure 4.9 shows there is a difference between Aho-Corasick and KMP on the Wi-Fi Pineapple. When the throughput is low, Aho-Corasick utilizes less power on Wi-Fi Pineapple but when the throughput increases, it uses more power than KMP. Therefore, KMP performs better at high throughput values. Aho-Corasick deviates from normal behaviour, and switches between low values and high values. On Raspberry Pi, both of the lines vary from less to more used power than the other for different throughput values. In this case, it is difficult to draw conclusions about this since the multimeter used only provides a very approximative value of the power and voltage. The Wi-Fi Pineapple required three cables to connect and boot it up. The multimeter was only connected through one of those cables and the current values it provided were multiplied by three to calculate the total power consumption, as per Kirchoff’s current law.

---

It is also reasonable that the power consumption on Wi-Fi Pineapple is higher than the power consumption on Raspberry Pi since the Raspberry Pi has better hardware specifications.

5.2 Method

One aim in this study was to discover if it was possible to increase the performance of the LPIDS compared to Snort when using different type of rules, e.g. with or without a content parameter, as well as using different pattern matching algorithms. As discussed and shown earlier, Aho-Corasick and KMP had higher throughput than Snort for the number of rules with fewer content parameters. One important consideration to note is that Snort, in a real-world setting, is not often run with a large number of rules that do not have the content parameter, which is shown by the Snort community rules, containing approximately 3,500 rules\footnote{https://github.com/seemoo-lab/nexmon}. The reliability of the method, however, is consistent. Using our method for configuring many rules matching only IP addresses, ports or protocols will have better performance in terms of throughput. Our tests were run using a packet capture file and not live tests in monitor mode in order to achieve consistency between all tests. There could be a small variation in performance when reading packets from a file versus capturing packets from the network due to a varying amount of input and output operations required. However, the effect of these variations in our evaluations would only result in a constant increase or decrease in performance and not with respect to scaling.

5.2.1 Hardware Platforms

One of the main goals of this thesis was to evaluate the viability of using two different embedded devices as an IDS. In this section, we discuss some of the issues that we found in deploying an IDS on these systems. First, as mentioned in Chapter 3, the Wi-Fi Pineapple platform does not include a compiler or debugger for C/C++, which makes both deployment and debugging of LPIDS a more complicated process. It also uses a minimized variant of \texttt{libc} called \texttt{uClibc} and similarly \texttt{uClibc++} for C++, which limits the availability of common standard library functions, forcing the developer to defer to older programming practices.

Secondly, the default firmware on the Broadcom chipsets used by the Raspberry Pi Model 3 does not support monitor mode. Since LPIDS is designed to operate in monitor mode, it is necessary to modify the firmware on the Raspberry Pi. There exist open-source initiatives\footnote{https://github.com/seemoo-lab/nexmon} which aid users in performing this modification.

5.2.2 Aho-Corasick Algorithm

Using the Aho-Corasick algorithm yielded the best results for the majority of the tests performed on LPIDS.

The disadvantages found using Aho-Corasick is the complexity of the implementation and inflexibility in relation to other algorithms. An implementation following from the theory using a trie as the underlying data structure in non-contiguous memory did not perform well in our tests and were concluded to be caused by inefficient
memory access patterns, degrading cache performance. An implementation using a state-transition matrix based on interleaved arrays increases the data locality, which greatly improves performance but increases the complexity of the implementation.

The Aho-Corasick algorithm also becomes inflexible in terms of inserting and removing patterns from the state machine after creation and requires a complete rebuild in practice. In the LPIDS implementation, a potentially large amount of patterns is eliminated before the pattern matching is performed and removal of patterns which are not eligible for a full signature match would result in a minor performance increase as shown in the evaluation. However, due to the impracticality of addition and deletion of patterns, it is not possible. Although this drawback is negligible for the use in this thesis due to the properties of the Aho-Corasick algorithm where the time complexity, in theory, does not depend on the number of patterns to be matched.

5.2.3 Knuth–Morris–Pratt Algorithm

Evaluations of LPIDS using the KMP algorithm showed expected results with a linear decrease in performance with respect to the number of rules that were loaded. In some cases where the number of patterns to be matched were low, KMP even outperformed Aho-Corasick, which presents the possibility of creating a hybrid solution for pattern matching where a small number of patterns could be searched using KMP and a larger number of patterns could be evaluated by the Aho-Corasick algorithm. This would improve the overall performance of LPIDS when there are only a few number of patterns to be matched.

The main advantage of KMP in an IDS context is the simplicity and flexibility. The implementation of the algorithm is trivial and the initial state-transition table can either be cached for reuse or quickly reconstructed.

5.2.4 Rule Filtering

In the figures showing the behaviour of the rule filtering R-trees, it does not show the best case logarithmic complexity. It is still better than linear search but it still shows a linear behaviour with a lower constant factor than that of linear search. This affects the performance as well, as displayed in Figure 4.4. This could have been improved by using an alternative spatial data structure such as R+-trees. A reason for linear behaviour may be that there are many overlapping rectangles in the R-tree, which degrades the performance. R+-trees could decrease the overlapping rectangles by inserting an object into multiple leaves if necessary, resulting in a search requiring fewer visited nodes compared to an R-tree [48]. This comes to a cost that it takes longer time to build up the tree and more memory is required. In our application, the initial building of the tree is not related with the performance as it is only built one time during initialization and not modified thereafter.

5.2.5 Validity of Study

Data Set

Due to limited time for this project, we did not have time to run the tests with many more rules, different packet capture files and multiple times. If we had this possibil-
5.3. The work in a wider context

It is possible that we could calculate a better mean value of the timings and draw better conclusions from the results. There might be disturbances in the network during the tests and the network condition may vary from day to day, which could affect the data and results. If the tests had been ran multiple times, for example, 10 to 15 times, the reliability of the results would have increased because having disturbances for the same test run on different moments is rare. If someone were to run our LPIDS in the future, this would make it more likely that they would retrieve similar results.

Publicly collected traffic has a high chance of representing general trends in Wi-Fi compared to business systems with more niche user bases. Due to the fact that they are public data, they are also more readily able to be shared with other researchers and departments for additional and validation of results.

The testing can be improved through more real-world traffic capturing and analysis sessions, perhaps by replaying traffic through a bandwidth and latency simulator.

Source criticism

During the thesis, care was taken to utilize peer-reviewed sources as much as possible. The references varied from scientific papers and web material. One of the references in the Section 1.1 was the CVE statistics which may not be the best reference to the number of attacks. This is not a complete list of the number of vulnerabilities, and the 2019 updated list has not been published yet.

Because of limited research and implementations of portable, wireless intrusion detection systems, this topic was interesting to research. A minor criticism of the sources used is that most of the primary sources date before the year 2000. This might suggest that the methods used are out of date. However, most studies in this field build upon Aho-Corasick. The paper for this algorithm was published in 1975. The same principle applies to some of the other references.

Another minor issue concerning the sources was the absence of works related to TCP segmentation offloading. This concept seems to have different names in different papers as well. Consequently, it would have been useful to have more scientific peer-reviewed papers concerning TCP segmentation offloading. The general impression was that overall, the sources used were scientific, suitable and trustworthy.

5.3 The work in a wider context

This study dealt with the collection and handling of network traffic data. A major challenge for public monitoring systems is preservation of users’ personal data. It was of critical importance to not intrude on the user’s privacy when gathering the network traffic data. Therefore, no user identification in any form was used when capturing the traffic. This maintains user confidentiality and privacy, which can be a significant barrier for permission or the perception of live network experiments.

A mobile wireless network was set up instead of a public one, to make sure no sensitive traffic is captured. It is also important to understand that the attacks simulated to test the LPIDS were only created to evaluate the LPIDS and will not be used for any other purpose.
If companies can better utilize the resources they have with different software, then they will not only save money, they will also consume less energy by using a lightweight portable IDS.
Conclusion

6.1 Research questions

Research question 1

Can the performance of an IDS increase by applying a single-pattern matching algorithm such as KMP, as an alternative to the multi-pattern matching algorithm Aho-Corasick?

The results presented in this thesis show that KMP has a higher performance when the number of rules is in the range of approximately 0 and 500. This concludes that Aho-Corasick is more suitable for larger rule sets. The threshold for when each of the algorithms performs better varies across the platforms and in order to maximize performance, would require tuning to find a close to optimal threshold for each algorithm.

Research question 2

Is a lightweight portable IDS viable on embedded devices such as Wi-Fi Pineapple and Raspberry Pi in terms of throughput, power consumption and response time?

According to the results in this paper, Wi-Fi Pineapple does not have enough processing capabilities to act as an IDS. Because of the limited maximum throughput Wi-Fi Pineapple can handle, it will start to drop packets if that throughput is reached. Raspberry Pi is significantly better than Wi-Fi Pineapple, in terms of throughput, response time and power consumption. This means it is more likely to capture more packets compared to Wi-Fi Pineapple and not miss many of them.

Research question 3

How does LPIDS compare to Snort in terms of throughput on a Raspberry Pi?
The goal of this work was to determine if we can get any performance advantages of using a different approach for the pattern search. Aho-Corasick and KMP had higher throughout compared to Snort when the number of rules containing content parameter decreases. In these specific conditions, we observed that our LPIDS scaled better than Snort. Therefore, if you have an IDS with many rules that do not contain a content parameter, the LPIDS is a better choice compared to Snort. The specific algorithm that has the highest performance in terms of throughput is Aho-Corasick, which should be the choice if LPIDS is used.

6.2 Future work

In this thesis it was decided to not implement multithreading support for our proposed LPIDS, as mentioned in Section 1.4. The Raspberry Pi 3 model features a quad-core processor and would have a significant increase in performance if the workload could be divided between them. Lin et al. [30] proposed and implemented a parallel Aho-Corasick algorithm using graphics cards in the context of NIDS pattern matching and observed a marked improvement in throughput. Applying these techniques in LPIDS on the Raspberry Pi would likely increase the performance.

In our results we found that although when LPIDS was run with 10000 rules, the majority of the computation time was spent in decryption and parsing routines. The same observation could be made from the evaluations of Snort. The reason for the large impact of these routines could be due to the smaller computational resources on the embedded devices they were tested on and the impact on systems with more computational resources could be negligible. However, if an IDS is to be deployed on an embedded system similar to those evaluated in this thesis, further investigations should be made in order to decrease the performance impact due to decryption and parsing of incoming packets.

During our comparison with Snort, we observed a decrease in performance correlated with the number of rules which used a content parameter. The reason for this is due to the system design of Snort where in the first step of processing incoming packets, the packet payload is matched against the content parameter in the rule using the Aho-Corasick algorithm. If the rule does not have a content parameter, the first step is skipped and all of the rule constraints are fully evaluated.

We propose modifying Snort’s processing pipeline such that it minimizes the number of rules which skip the first step in the processing by filtering the rules which do not have a content parameter in a spatial data structure, similar to the processing in LPIDS. Figure 6.1 shows an overview of our proposed improvement that can be made in Snort. Arguably the increase in performance of such a modification is small in most practical scenarios as the rules typically do have a content parameter. However, as mentioned in the Chapter 5, 135 out of 3,505 rules in a publicly available rule set do not have such a parameter and would therefore result in considerable overhead for each processed packet.
Figure 6.1: Proposed modification of Snort using R-trees.
Bibliography


