
Effektiv, flerkärnig implementation av IPsec Encapsulating Security Payload protokollet för en Security Association

Mattias Hellsing
Albin Odervall

Supervisor: August Ernstsson
Examiner: Christoph Kessler

External supervisors: Magnus Degerfalk and Michael Lundkvist, Ericsson
Upphovsrätt


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/.

© Mattias Hellsing
Albin Odervall
Abstract

As the mobile Internet traffic increases, the workload of the base stations processing this traffic increases with it. To cope with this, the telecommunication providers responsible for the systems deployed in these base stations have looked to parallelism. This, together with the fact that these providers have a vested interest in protecting their users’ data from potential attackers, means that there is a need for efficient parallel packet processing software which handles encryption as well as authentication. A well known protocol for encryption and authentication of IP packets is the Encapsulating Security Payload (ESP) protocol of the IPsec protocol suite. IPsec establishes simplex connections, called Security Associations (SA), between entities that wish to communicate. This thesis investigates a special case of this problem where the work of encrypting and authenticating the packets within a single SA is parallelized. This problem was investigated by developing and comparing two multi-threaded implementations based on the Eventdev, an event driven programming library, and ring buffer libraries of Data Plane Development Kit (DPDK). One additional Eventdev-based implementation was also investigated which schedules linked lists of packets, instead of single packets, in an attempt to reduce the overhead of scheduling packets to the worker cores. These implementations were then evaluated in terms of throughput, latency, speedup, and last level cache miss rates. The results showed that the ring buffer-based implementation performed the best in all metrics while the single packet-scheduling Eventdev-based implementation was outperformed by the one using linked lists of packets. It was shown that the packet generation, which was done by the receiving core, was the main limiting factor for all implementations. In addition, the memory resources such as the memory bus, memory controller and prefetching hardware were shown to likely be an area of contention and a possible bottleneck as the packet generation rate increases. The conclusion drawn from this was that a parallelized packet retrieval solution such as Receive Side Scaling (RSS) together with minimizing memory resource contention is necessary to further improve performance.
Preface

This thesis was written by two authors. While most of it was co-authored, there are a few sections that we wish to personally attribute.

The presentation of the ring buffer-based implementation was written by Albin Odervall, this includes sections 3.5, 5.4 and subsection Ring Buffer-Based Implementation Evaluation in Section 5.12. Albin also authored the sections regarding implementation and verification of the Encapsulating Security Payload protocol, sections 3.2, 4.6 and 5.1. Further, Albin authored the sections discussing Hyper-Threading, subsection Intel Hyper-Threading Evaluation of Section 4.9 and subsection Hyper-Threading Evaluation of Section 5.8.

The presentation of the Eventdev-based implementations was written by Mattias Hellsing, this includes sections 3.6, 5.5, 5.6 and subsections Eventdev-based Implementation Evaluation and Eventdev-Based Implementation Utilizing Chained Packet Buffers Evaluation in Section 5.12. Mattias also authored the parts of the report addressing the bottlenecks of the application; subsection Bottleneck Evaluation of Section 4.9, subsection Bottleneck Evaluation of Section 5.8 and subsection Extended Bottleneck Evaluation in Section 5.12.

Acknowledgements

We would like to thank our supervisors at Ericsson, Magnus Degerfalk and Michael Lundkvist, for their assistance during our thesis. We would also like to thank our examiner Christoph Kessler at Linköping University. Furthermore, we want to thank our opponents Sara Bergman and Cristian Torrusio.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Preface</td>
<td>iv</td>
</tr>
<tr>
<td>Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Research Questions</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Delimitations</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis Overview</td>
<td>3</td>
</tr>
<tr>
<td>2 Background</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Data Stream Processing</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Parallelism</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Cache-Aware Programming</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Cryptography</td>
<td>12</td>
</tr>
<tr>
<td>2.5 The Data Plane Development Kit</td>
<td>13</td>
</tr>
<tr>
<td>2.6 IPsec</td>
<td>19</td>
</tr>
<tr>
<td>2.7 AES in Galois/Counter mode for ESP</td>
<td>22</td>
</tr>
<tr>
<td>2.8 Related Work</td>
<td>24</td>
</tr>
<tr>
<td>2.9 Analysis of Non-Academic Work on IPsec</td>
<td>29</td>
</tr>
<tr>
<td>3 Implementation</td>
<td>31</td>
</tr>
<tr>
<td>3.1 Application Structure</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Encapsulating Security Payload Implementation</td>
<td>32</td>
</tr>
<tr>
<td>3.3 Single-Threaded Implementation</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Multi-Threaded Architecture</td>
<td>36</td>
</tr>
<tr>
<td>3.5 Ring Buffer-Based Implementation</td>
<td>39</td>
</tr>
<tr>
<td>3.6 Eventdev-Based Implementations</td>
<td>40</td>
</tr>
<tr>
<td>4 Evaluation Method</td>
<td>45</td>
</tr>
<tr>
<td>4.1 Hardware, Software and Compiler</td>
<td>45</td>
</tr>
<tr>
<td>4.2 Packet Generation</td>
<td>47</td>
</tr>
<tr>
<td>4.3 Packet Streams</td>
<td>47</td>
</tr>
<tr>
<td>4.4 Metrics</td>
<td>48</td>
</tr>
<tr>
<td>4.5 Measurements</td>
<td>48</td>
</tr>
<tr>
<td>4.6 Verification of the Encapsulating Security Payload Protocol Implementation</td>
<td>49</td>
</tr>
<tr>
<td>4.7 Configuration Parameters</td>
<td>51</td>
</tr>
<tr>
<td>4.8 Configuration Evaluation</td>
<td>52</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The maximal theoretical speedup given the parallelizable portion of a program and number of processors.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Ring buffer with seven out of eight slots occupied</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Eventdev flow chart</td>
<td>16</td>
</tr>
<tr>
<td>2.4</td>
<td>Packet before and after ESP in transport mode is applied</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Packet before and after ESP tunnel mode is applied</td>
<td>21</td>
</tr>
<tr>
<td>2.6</td>
<td>ESP tunnel mode header</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>ESP payload using AES-GCM-ESP</td>
<td>23</td>
</tr>
<tr>
<td>2.8</td>
<td>AES-GCM nonce using AES-GCM-ESP</td>
<td>23</td>
</tr>
<tr>
<td>2.9</td>
<td>AES-GCM additional authenticated data using AES-GCM-ESP with 32-bit sequence numbers</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Application structure</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Processing stages for an incoming IPv4 packet to be encrypted</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>IPv4 header</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>ESP header</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Processing stages for an incoming ESP packet to be decrypted</td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Eventdev-based implementation structure</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Processor and cache topology generated using likwid-topology -g. The top boxes in each socket shows the thread identifiers for each core. The size of the L1 cache follows below, then the size of the L2 cache. The L3 cache is shared across the whole socket and can be seen in the bottom box in each socket.</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Wireshark ESP SAs preferences</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Core and thread topology when evaluating Hyper-Threading using the ring implementation</td>
<td>53</td>
</tr>
<tr>
<td>4.4</td>
<td>Core and thread topology when evaluating Hyper-Threading using the EventDev-based implementations</td>
<td>53</td>
</tr>
<tr>
<td>5.1</td>
<td>Decrypted ESP packets in Wireshark</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>Throughput of the single-threaded implementation using the IMIX packet stream</td>
<td>57</td>
</tr>
<tr>
<td>5.3</td>
<td>Latency of the single-threaded implementation using the IMIX packet stream</td>
<td>58</td>
</tr>
<tr>
<td>5.4</td>
<td>LLC miss rates of the single-threaded implementation using the IMIX packet stream</td>
<td>58</td>
</tr>
<tr>
<td>5.5</td>
<td>Throughput of the ring implementation using 1 worker core using the IMIX packet stream</td>
<td>59</td>
</tr>
<tr>
<td>5.6</td>
<td>Throughput of the ring implementation using 2 worker cores using the IMIX packet stream</td>
<td>59</td>
</tr>
<tr>
<td>5.7</td>
<td>Throughput of the ring implementation using 3 worker cores using the IMIX packet stream</td>
<td>60</td>
</tr>
<tr>
<td>5.8</td>
<td>Throughput of the ring implementation using 4 worker cores using the IMIX packet stream</td>
<td>60</td>
</tr>
</tbody>
</table>
5.9 Latency of the ring implementation using 1 worker core using the IMIX packet stream ........................................ 61
5.10 Latency of the ring implementation using 2 worker cores using the IMIX packet stream ........................................ 62
5.11 Latency of the ring implementation using 3 worker cores using the IMIX packet stream ........................................ 62
5.12 Latency of the ring implementation using 4 worker cores using the IMIX packet stream ........................................ 63
5.13 Last level cache miss rates of the ring implementation using 4 worker cores using the IMIX packet stream ........................................ 64
5.14 Throughput of the event implementation using 1 worker core using the IMIX packet stream ........................................ 65
5.15 Throughput of the event implementation using 2 worker cores using the IMIX packet stream ........................................ 65
5.16 Throughput of the event implementation using 3 worker cores using the IMIX packet stream ........................................ 65
5.17 Latency of the event implementation using 1 worker core using the IMIX packet stream ........................................ 66
5.18 Latency of the event implementation using 2 worker cores using the IMIX packet stream ........................................ 67
5.19 Latency of the event implementation using 3 worker cores using the IMIX packet stream ........................................ 67
5.20 Last level cache miss rates of the event implementation using 3 worker cores using the IMIX packet stream ........................................ 68
5.21 Throughput of the event-chained implementation using 1 worker core using the IMIX packet stream ........................................ 69
5.22 Throughput of the event-chained implementation using 2 worker cores using the IMIX packet stream ........................................ 69
5.23 Throughput of the event-chained implementation using 3 worker cores using the IMIX packet stream ........................................ 69
5.24 Latency of the event-chained implementation using 1 worker core using the IMIX packet stream ........................................ 70
5.25 Latency of the event-chained implementation using 2 worker cores using the IMIX packet stream ........................................ 71
5.26 Latency of the event-chained implementation using 3 worker cores using the IMIX packet stream ........................................ 71
5.27 Last level cache miss rates of the event-chained implementation using 3 worker cores using the IMIX packet stream ........................................ 72
5.28 Throughput in Mpps of the chosen configuration of each implementation using the IMIX packet stream ........................................ 72
5.29 Throughput in Mbps of the chosen configuration of each implementation using the IMIX packet stream ........................................ 73
5.30 Latency of the chosen configuration for each implementation using the IMIX packet stream ........................................ 74
5.31 Latency distribution of the chosen configuration for the single-threaded implementation using the IMIX packet stream ........................................ 75
5.32 Latency distribution of the chosen configuration for the ring implementation using the IMIX packet stream ........................................ 76
5.33 Latency distribution of the chosen configuration for the event implementation using the IMIX packet stream ........................................ 76
5.34 Latency distribution of the chosen configuration for the event-chained implementation using the IMIX packet stream ........................................ 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.35</td>
<td>Throughput in Mpps of the chosen configuration of each implementation with varying packet sizes</td>
</tr>
<tr>
<td>5.36</td>
<td>Throughput in Mbps of the chosen configuration of each implementation with varying packet sizes</td>
</tr>
<tr>
<td>5.37</td>
<td>Latency of the chosen configuration of each implementation with varying packet sizes</td>
</tr>
<tr>
<td>5.38</td>
<td>Throughput of the chosen configuration of each implementation with copying of packet data enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.39</td>
<td>Throughput of the chosen configuration of each implementation with encryption enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.40</td>
<td>Throughput of the chosen configuration of each implementation with copying of packet data and encryption enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.41</td>
<td>Latency of the chosen configuration of each implementation with copying of packet data enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.42</td>
<td>Latency of the chosen configuration of each implementation with encryption enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.43</td>
<td>Latency of the chosen configuration of each implementation with copying of packet data and encryption enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.44</td>
<td>Last level cache miss rates of the chosen configuration of each implementation with copying of packet data enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.45</td>
<td>Last level cache miss rates of the chosen configuration of each implementation with encryption enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.46</td>
<td>Last level cache miss rates of the chosen configuration of each implementation with copying of packet data and encryption enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.47</td>
<td>Throughput of the chosen configuration of each implementation with Hyper-Threading (HT) enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.48</td>
<td>Latency of the chosen configuration of each implementation with Hyper-Threading (HT) enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.49</td>
<td>Throughput of the chosen configuration of each implementation with reordering enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.50</td>
<td>Latency of the chosen configuration of each implementation with reorder enabled versus disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.51</td>
<td>Throughput of the ring implementation with queue configuration 1-1 using the IMIX packet stream</td>
</tr>
<tr>
<td>5.52</td>
<td>Latency of the ring implementation with queue configuration 1-1 using the IMIX packet stream</td>
</tr>
<tr>
<td>5.53</td>
<td>Throughput of the event implementation with queue configuration 1-1 using the IMIX packet stream</td>
</tr>
<tr>
<td>5.54</td>
<td>Latency of the event implementation with queue configuration 1-1 using the IMIX packet stream</td>
</tr>
<tr>
<td>5.55</td>
<td>Throughput of the event-chained implementation with queue configuration 1-1 using the IMIX packet stream</td>
</tr>
<tr>
<td>5.56</td>
<td>Latency of the event-chained implementation with queue configuration 1-1 using the IMIX packet stream</td>
</tr>
<tr>
<td>5.57</td>
<td>Throughput of the ring implementation with queue configuration 1-1 and copying of packet data disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.58</td>
<td>Throughput of the event implementation with queue configuration 1-1 and copying of packet data disabled using the IMIX packet stream</td>
</tr>
<tr>
<td>5.59</td>
<td>Throughput of the event-chained implementation with queue configuration 1-1 and copying of packet data disabled using the IMIX packet stream</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Core tasks for the ring buffer-based implementation</td>
</tr>
<tr>
<td>3.2</td>
<td>Example packet distribution across 6 cores of a single receive batch</td>
</tr>
<tr>
<td>3.3</td>
<td>Core tasks for the Eventdev-based implementations</td>
</tr>
<tr>
<td>3.4</td>
<td>General configuration of the Eventdev-based implementations</td>
</tr>
<tr>
<td>3.5</td>
<td>Number of events allowed at a time in the Eventdev-based implementations</td>
</tr>
<tr>
<td>3.6</td>
<td>Event port configuration in the Eventdev-based implementations</td>
</tr>
<tr>
<td>3.7</td>
<td>Queue configuration in the Eventdev-based implementations</td>
</tr>
<tr>
<td>4.1</td>
<td>Packet sizes, their overhead, and distribution in the IMIX stream used during evaluation</td>
</tr>
<tr>
<td>5.1</td>
<td>Results of the anti-replay service testing</td>
</tr>
<tr>
<td>5.2</td>
<td>Parameters of the chosen configurations</td>
</tr>
<tr>
<td>5.3</td>
<td>Throughput, latency, and speedup for the top performing configurations compared to the single-threaded implementation during encryption using the IMIX packet stream</td>
</tr>
<tr>
<td>5.4</td>
<td>Throughput, latency, and speedup for the top performing configurations compared to the single-threaded implementation during decryption using the IMIX packet stream</td>
</tr>
</tbody>
</table>
## Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>SISD example</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>SIMD parallelism example</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>MIMD parallelism example</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Race condition example</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Race condition example, thread_1 instructions</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Race condition example, thread_2 instructions</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Padded payload length calculation</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Packet distribution algorithm</td>
<td>39</td>
</tr>
</tbody>
</table>
The traffic volume of the Internet is growing each year [1], this creates a demand for both hardware and software able to handle these heavy loads. As an increasing number of mobile devices connect to the Internet via the wireless networks of telecommunication providers, these providers must handle an increasingly large portion of the traffic. To cope with this, the next generation of wireless telecommunication—5G—is designed to reach data rates 100 times larger, lower latency by a factor of five, and handle data volumes 1000 times greater than what we see today [2]. At the most extreme routing points on the Internet, the traffic forwarded can reach up to 6 Tbps at peak hours [3,4]. The part of a packet switched network that is responsible for forwarding the data packets is called the data—or forwarding—plane. The data plane must be able to quickly process these packets to allow for the increasingly high throughput of the Internet. To handle this workload—as in many areas where heavy computations are performed—engineers have looked to parallelism. This parallelism comes in the form of distributed computations across servers in large data centers but also across multiple cores in the same processor.

Telecommunication network providers, such as Ericsson[1], have a vested interest in protecting their users’ data from potential attackers. Since these providers usually operate at the Internet layer of the network stack, they cannot utilize higher-level protocols such as TLS[2]. Instead, a common and standardized security protocol at the Internet layer is IPsec, which can provide confidentiality, integrity, and authenticity. IPsec establishes simplex connections, called Security Associations (SA), between entities that wish to communicate. When such connections are established, negotiation is required on how IPsec should protect the communication such as which keys and which cryptographic algorithms to use. Parallelizing multiple SAs is trivial as there is no shared data between them and as such the connections can be distributed over multiple cores without taking any additional measures as to synchronize data. As many of the connections use the same path through the network though, such as via a telecommunications base station, it can be useful to aggregate them into a single SA to reduce overhead. When aggregating multiple connections into one SA, it can result in a throughput larger than what a single core can handle, requiring multiple cores for a single
1.1. Research Questions

1. How can a solution be designed so that it can efficiently parallelize the Encapsulating Security Payload protocol of a single IPsec Security Association?

2. How can such a solution be implemented in C for x86_64 using the Data Plane Development Kit based on the ring buffer library?

3. How does a solution based on the Eventdev library of the Data Plane Development Kit compare to the, less complex, ring buffer-based solution on the same platform?

4. How does the throughput and latency of these implementations scale with the number of available cores?

1.2 Delimitations

The ESP implementation presented in this thesis does not fully conform to the ESP standard to reduce the development time and to focus on the aspects that are important for this thesis. Therefore, there are some requirements specified as MUST in the ESP standard—meaning they are obligatory for an RFC 4303 conforming implementation—that the implementation presented in this thesis does not fulfill. These exceptions are listed below:

1. Only IPv4 packets are supported.

2. Only the tunnel mode is supported.

3. Only unicast SAs are supported.

4. Only 32-bit sequence numbers are supported, and sequence numbers are not checked for overflow.
5. Only the combined mode algorithm AES-GCM is supported in confidentiality and integrity mode. Integrity only mode is not supported.

6. SA negotiation (using IKE) is not supported as the SA is pre-shared. Although the ESP standard suggests that the anti-replay service should not be used with manually keyed SAs, the anti-replay service is enabled. This is because a final implementation of this application would most likely not use manual keying and the anti-replay service is a valuable feature of ESP.

7. No signalling mechanism for negotiating IPsec versions is supported.

8. No mechanisms for routing and mapping inbound traffic to SAs are supported.

9. No traffic flow confidentiality is supported.

10. No capability exists for generating or discarding dummy packets.

11. IP fragmentation and reassembly are not supported.

The parts of IPsec that are not part of the ESP protocol have only been implemented to the degree where the ESP implementation was functional. This means that the implementation does not support gateway functionality or any form of routing. This delimitation was made because of the focus of the thesis being on the parallelization of the ESP packet processing and not IPsec itself.

1.3 Thesis Overview

The layout of this thesis is as follows: It begins with a review of the background for this thesis in Chapter 2, explaining the terms and technologies used as well as related work. Following, in Chapter 3 are descriptions of how the Encapsulating Security Payload processing was implemented as well as the application developed for this thesis. Then the evaluations performed on the application are presented in Chapter 4 together with the environment they were performed in. In Chapter 5 the results of the evaluations are presented as well as analyzed and discussed. Finally, the thesis is concluded in Chapter 6 where the conclusions and future work are presented.
This chapter begins by explaining concepts and terms that are necessary to understand the report as well as the background that will be used throughout the report. Some related work that has been done in neighbouring fields and is relevant to this thesis is also presented.

2.1 Data Stream Processing

Data stream processing is a programming paradigm in which set of operators, often called kernels, are applied to a stream of data. This following section describes the theoretical base from the data stream processing field used in this thesis. To do this, it introduces a paper by De Matteis and Mencagli in which the authors present four different patterns for window-based stateful operators, a common approach to data stream processing. The authors show how different parallelizaton issues in data stream processing can be mitigated with these patterns. The patterns build upon the idea that large streams of data can be split into smaller streams on which the processing can be parallelized. These streams are fed through workers which perform a specific operation, such as mapping or filtering. These streams can in turn have multiple substreams uniquely identified by their keys. The paper’s focus is on stateful operators, i.e. operators that keep state between the elements processed. All patterns presented have one feature in common, the sliding window, which is how state is kept in the operators. The window size (i.e. how many elements are processed at one time) is expressed as either number of elements (count-based windows) or time units (time-based windows). This window “slides” on top of the data stream, moving forwards by a sliding factor, expressed in number of elements or time units. The paper describes two paradigms for parallelization: *window parallelism* and *data parallelism*. Window parallelism—which three of the four patterns described belongs to—is a paradigm in which multiple windows can be operated on in parallel. In data parallelism, only a single window is operated on at any given time but the execution of the operators is parallelized. A last distinction made is one between *agnostic* and *active* workers. Agnostic workers only apply the operators to the data stream and the building and updating of the windows is left to separate subroutines. Active workers on the other hand, partially or fully handle the window management themselves in addition to applying the operators to the data stream. Subroutines building and updating windows and giving
them to the workers are called *emitters*, subroutines collecting output from the workers are called the *collectors*.

The first pattern presented is called *window farming*. In this pattern, each worker takes multiple windows (one per key) and applies its operator to each of them. Each worker operates on a unique window so that each window is only processed once. The workers in this pattern can be both agnostic and active. In the agnostic case, each window is given to the worker in one batch. In the active case, the emitter can send individual elements to the workers which they then insert into their existing window while also removing the oldest element at the same time. This reduces the overhead of copying the windows, since subsequent windows can contain the same information to some degree. The window farming pattern optimizes throughput, but the latency is the same as with serial processing. The second pattern described is the *key partitioning* pattern. This pattern partitions the work across the workers by the keys instead. Each worker is dedicated to a single key and processes that key’s windows serially. The pattern requires multiple substreams (i.e. multiple keys) but also has the downside that if the load on each key is not uniform, some workers will have more work to do than others leading to a skewed workload distribution. Key partitioning improves throughput over serial processing. Another advantage is that elements with the same key arrive at the collector in order. The workers can be agnostic or active in the same way as with the window farming pattern. The third pattern, *pane farming*, has a pipelined approach. If the operator can be split into two functions, this pattern works by applying the first function, collecting the results, reordering them if necessary, and then applying the second function. This can be seen as two implementations of window farming being pipelined. This pattern improves throughput and latency as a result of the pipelining. The last pattern presented is the *window partition* pattern. This is the only pattern that is of the data parallelism paradigm, meaning that only one window is processed at any given time. In this pattern the workers are fully active in the window management. The workers perform their operation on individual elements of the window. The result of the operations is then sent to a reducer which computes the final result of the window processing. This pattern improves throughput and optimizes latency.

### 2.2 Parallelism

This section introduces the concept of parallelism, some different classes of parallelism and discusses some common problems and solutions in parallelism. Parallelism can be separated into two areas: software parallelism and hardware parallelism. Software parallelism is defined by the program to be executed and the operating system on which it runs. The operating system schedules different software threads to be run on the hardware, which is often hidden from the programmer. Hardware parallelism is instead defined by the hardware architecture, such as how many physical cores exists and in what fashion instructions can be executed on these cores. Therefore, it is important to make the distinction between software threads and hardware threads. Software threads are threads created by the operating system and can be created, destroyed, paused and resumed at runtime. Hardware threads are fixed and defined by the hardware itself.

There are multiple ways to achieve parallel execution. Some hardware, such as GPUs, are specialized for data parallel execution and may have thousands of cores [7]. These processing units execute hundreds or thousands of threads concurrently which puts restrictions on the programs that can be run; they can usually only execute a single program on all threads at a time, each thread operating on different data. CPUs tend to have much lower core counts, commonly ranging from 2 to 32 cores—although much higher core counts exist—on a single physical CPU [8]. Some CPUs also support simultaneous multi-threading (SMT), allowing
2.2 Parallelism

Multiple threads to be run concurrently on a single core, such as Intel’s proprietary Hyper-
Threading Technology. Unlike GPUs, CPUs allow arbitrary programs to be run on each
hardware thread. For more information about parallelism we refer to Chapter 17 and 18 in
Stallings [9].

2.2.1 Classes of Parallelism

In a paper by Flynn [10], different parallelization techniques are classified into four different
classes:

- Single Instruction Single Data (SISD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Multiple Instruction Multiple Data (MIMD)

SISD is the traditional architecture where a single thread executes instructions on a single
piece of data at a time, i.e. there is no parallelism. SIMD is an architecture where a single
instruction is executed on multiple pieces of data at a time. An implementation of SIMD
can be seen in the Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX)
instruction set extensions for x86, and the NEON instruction set extension for ARM. MISD is
where different instructions are executed on a single piece of data at a time. This technique
is more of an uncommon architecture and not of particular interest to this thesis, but is used
when fault tolerance is required, such as in space missions [11]. MIMD is perhaps the most
common architecture today where different instructions are executed on different data at a
time; this is hardware multi-threading. For more information regarding different classes of
parallelism, refer to Chapter 17 in Stallings [9].

Below follows a few simple code examples (in C-like pseudocode) of how different the par-
allelization techniques can look when implementing a multiply-accumulate operation.

The first example is a SISD implementation, see Listing 2.1.

```
Listing 2.1: SISD example

#define SIZE 8192
float a[SIZE] = { ... };
float b[SIZE] = { ... };
float c[SIZE] = { ... };
for (int i = 0; i < SIZE; i++) {
    c[i] = c[i] + a[i] * b[i];
}
```

The second example is a SIMD implementation using Intel SSE intrinsics[2] see Listing 2.2

---

threading-technology.html

2.2. Parallelism

Listing 2.2: SIMD parallelism example

```c
#define SIZE 8192
float a[SIZE] = { ... };
float b[SIZE] = { ... }; 
float c[SIZE] = { ... }; 

for (int i = 0; i < SIZE / 4; i++) {
    __m128 ma = _mm_load_ps(a + 4 * i);
    __m128 mb = _mm_load_ps(b + 4 * i);
    __m128 md = _mm_mul_ps(ma, mb);
    mc = _mm_add_ps(mc, md);
    _mm_store_ps(c + 4 * i, mc);
}
```

Note that the SIMD example has two additional steps, _mm_load_ps and _mm_store_ps, which are used to convert the data to and from a SIMD friendly layout. These operations are expensive, so performing each only once at the beginning and the end of the program respectively is preferable so that the data stays in the SIMD friendly layout for as long as possible. It is also important to note that in order to get the best performance out of SIMD, the amount of data elements should be evenly divisible by the amount of elements processed in each step. For the same reason, it is also often preferable if the data is aligned to certain boundaries when using SIMD (commonly 128- or 256-bits).

The third example is a MIMD example where processing of the data is split into four different threads, each operating in a SISD fashion, see Listing 2.3. The following example is in pseudocode, as a full example using the threading API available of Linux would be too long. start_thread is a function taking a function pointer as well as a thread index as arguments and starts a thread executing the specified function. In reality, this would create a software thread and not a hardware thread, so true MIMD parallelism is not guaranteed.

Listing 2.3: MIMD parallelism example

```c
#define SIZE 8192
#define THREADS 8
#define SIZE_PER_THREAD (SIZE / THREAD)
float a[SIZE] = { ... }; 
float b[SIZE] = { ... }; 
float c[SIZE] = { ... }; 

void thread(int thread_index) {
    int offset = thread_index * SIZE_PER_THREAD;
    for (int i = 0; i < SIZE_PER_THREAD; i++) {
        c[offset + i] = a[offset + i] + b[offset + i];
    }
}

for (int i = 0; i < THREADS; i++) {
    start_thread(thread, i);
}
```
These different implementations could be combined into a multi-threaded implementation using SIMD to process multiple elements at once on each thread. Overhead of the setup and teardown processes need to be taken into careful consideration since for small data this can be a significant chunk of the time consumed.

### 2.2.2 Amdahl’s Law

Amdahl’s law can be used to calculate the theoretical speed up that can be gained by parallelizing a program [12]. It is formulated as follows where $p$ is the portion of a program that can be parallelized and $n$ is the number of processors available:

$$ S(n) = \frac{1}{(1 - p) + \frac{p}{n}} $$

(2.1)

The maximal theoretical speed up is given when $n \rightarrow \infty$:

$$ \lim_{n \rightarrow \infty} S(n) = \frac{1}{1 - p} $$

(2.2)

Figure 2.1 shows the maximal theoretical speedup that can be gained given the parallelizable portion of a program and number of processors.

![Figure 2.1: The maximal theoretical speedup given the parallelizable portion of a program and number of processors](image)

### 2.2.3 Shared Memory and Synchronization

For parallel programs, the need for memory to be shared across threads is a common problem that has to be solved in order to write non-trivial multi-threaded programs. Solutions to
problems that can be solved across many threads that do not require such memory sharing, or problems that can be transformed into such problems, are highly sought after. Minimizing communication using shared memory between threads is of utmost importance for high performance applications since such sharing of memory requires threads to wait for other threads, essentially interlocking the execution of the individual threads.

Another, but related, problem with sharing memory between threads is that of synchronization. In order to keep data sets consistent between different threads access to data, that data need to be synchronized between the threads. In Listing 2.4 a simple, yet illustrative, example of a race condition is demonstrated. A race condition is an example of a problem that can occur in multi-threaded programs. Race conditions occurs when the result of the program execution depends on the order in which the memory accesses by the threads are executed and can be controlled by the means of synchronization.

Listing 2.4: Race condition example

```c
int a = 0;

void thread_1(void) {
    a = a * 2;
}

void thread_2(void) {
    a = 1;
}

int main(void) {
    start_thread(thread_1);
    start_thread(thread_2);
    wait_for_threads();

    return 0;
}
```

The execution of the program above can result in different results depending on how the instructions generated by the functions thread_1 and thread_2 are interleaved during execution. Say thread_1 and thread_2 are compiled into the following CPU instructions, see Listing 2.5 and 2.6 using a pseudo instruction set for simplicity.

Listing 2.5: Race condition example, thread_1 instructions

```assembly
LOAD a, r0 ; Load variable a into register r0
MUL r0, 2 ; Multiply value in register r0 by decimal constant 2
STORE r0, a ; Store value in register r0 into variable a
```

Listing 2.6: Race condition example, thread_2 instructions

```assembly
STORE 1, a ; Store the decimal value 1 into variable a
```
Each thread has to execute its own instructions in order, but the way the instructions of the different threads are interleaved are subject to the processor architecture and can potentially execute in any order. Therefore, the STORE 1, a instruction of thread_2 can be executed before, after, or in between any of the instructions of thread_1. If the entirety of thread_1 is executed before thread_2, the resulting value of a will be 1. If the entirety of thread_2 is executed before thread_1, the resulting value of a will be 2. Any other interleaving will result in the value of a becoming 0 (since thread_1 will load 0 as the value of a, multiplying it by 2 and writing it to memory. Any changes thread_2 did to a in between those instructions will be overwritten.

The example presented is trivial but demonstrates the problems that can occur in even the simplest of multi-threaded programs. Because of this, synchronization is required to make sure that the result of the program execution is always the same by controlling the order the instructions are interleaved in. There are a few ways to solve the problem of synchronization but they are not without cost. There are three common classes of synchronization techniques; lock-based synchronization, lock-free synchronization, and wait-free synchronization.

Lock-based synchronization—often implemented using semaphores or mutexes—enforces that only a single thread can execute a certain piece of code at a time. Since locks ensure that only a single thread can execute a certain piece of code at a time they can result in threads waiting for each other, thus slowing down the execution of the program. Another problem with locks is that of deadlocks. A deadlock occurs when the program cannot continue execution because one (or more) threads wait for a lock that the other thread has acquired and vice versa. The program is now in a deadlock because the two threads do not release their locks until they have acquired the other lock. Because of these problems, programs should limit the amount of code within mutual exclusion segments in order to minimize waiting. Using locks must also be done with the utmost care in order to avoid problems such as deadlocks.

The lock-free synchronization technique solves some of the problems that locks have. A lock-free implementation guarantees progression of at least one thread within the program as whole, and it can therefore not end up in a deadlock situation. Lock-free data structures and algorithms are often implemented using atomic operations. The support for atomic operations are implemented in the standard library of many programming languages, among them C and C++ but are also supported by some instruction sets intrinsically, such as x86_64.

Lock-free data structures and algorithms, as well as atomic operations, do not ensure that each thread makes progress but rather that one thread will make guaranteed progress, thus the program will finish in a finite amount of steps. Lock-free implementations of common data-structures and algorithms have been widely researched [13, 14, 15, 16] and are to prefer over lock-based implementations.

Further, wait-free data structures and algorithms are implementations where per thread progress is also guaranteed. The idea of lock-freedom has been known for decades [17] but until recently, fully wait-free data structures has been an open problem [18]. A wait-free implementation means that threads must not wait for each other to progress, resulting in even lower overhead from synchronization.
2.3 Cache-Aware Programming

This section introduces the different memory types of a modern computer architecture and discusses how the cache can be exploited to achieve higher performance. The memory of modern computer systems is organized as different memory types. Each memory type has its own advantages and disadvantages and together they allow for efficient memory access. To begin with, there are the four main memory types; cache internal, main, on-line mass storage (such as hard-drive storage), and off-line bulk storage. Moving down this list, the access times become longer but the storage sizes become larger. The internal memory consists of the processor registers as well as its cache.

The cache organization of modern computer systems has an internal hierarchy; the L1, L2, and—sometimes—L3 cache. These are all different in how they can be interacted with and also how they are shared between the cores of the processor. On multi-core systems, each core has access to both a shared cache as well as private cache. The L1 cache is closest to the core itself and is not shared. It often consists of separate instruction and data caches. This cache is most often on the order of magnitude of 10 - 100 kilobytes in size. The limitation in size is due to the cache is accessed, larger sizes means longer access times. The L2 cache can be shared between cores or individual to each core and trades slower access times for more storage. In modern computer systems there is also often an L3 cache which resembles a mix between main memory and a cache. It is usually quite large compared to the other caches and offers better access times than main memory as it is situated closer to the processor. The memory of this cache is shared between processor cores which reduces access times when multiple cores are working on the same data. To give an idea of the difference in access times, it takes just 1 ns to access the L1 cache while accessing the L2 cache takes around 3 ns. For additional information on cache-aware programming, refer to Kowarschik and Weiß' paper on cache optimization techniques and cache-aware numerical algorithms [19].

If memory that is not in one of the caches closest to the processor, or not in the cache at all, is referenced it will cause major access time penalties as they must be fetched from caches further away from the processor or even main memory. These are called cache misses and because of this, it is important, when possible, to pre-fetch data into the cache before initiating work on it. It is also important to align the data to be worked on with the cache. This is done by exploiting the way the data is inserted into the cache. The cache is split into sections which are called cache-lines. Each time data is fetched to the cache, the processor fetches a fixed amount of data at a time in the form of a cache-line, usually including some additional data adjacent in memory as well. This means that data that is going to be worked on later can be fetched with the previous cache-line if that data is part of the previously fetched cache-line. If the system exhibits strong spatial locality—which refers to the use of data elements that are stored relatively close in memory—this kind of optimization in data placement can give good results. There is also another form of locality called temporal locality which refers to how often the same data is accessed in a short time-frame. The cache was created to increase the performance of systems that exhibit strong temporal locality. If an application behaves in such a way that the data in the cache is constantly being swapped for data from memory, it is said to be thrashing the cache. This behavior should be avoided as it implies the number of cache misses will be high and be detrimental to performance. Further information on cache locality can be found in Chapters 5 and 6 in Culler et al. [20].

Cache-aware programming also extends into multi-threaded or multi-process applications. A fundamental factor for how well an application can run on a multi-core system is how well the threads and processes run on each core can co-exist on the same processor, this includes the processor’s cache. In an article by Zhuralev, Blagodurov and Fedorova [21], a scheduler is presented which is designed to schedule processes onto cores sharing the same cache based
on how much of an impact they have on each other’s performance. The authors begin by talking about how much of the work in this field has been focused on the issue of cache-contention as that has been seen as the main cause of performance degradation in multicore systems. A classification system created by Xie and Loh [22] is presented in the article by Zhuralev, Blagodurov and Fedorova as a way of classifying applications based on the impact they have on each other when scheduled onto cores sharing the same cache. This classification scheme uses the following animals to describe different classes of applications:

- Turtle - Low use of the shared cache.
- Sheep - Low miss rate of the shared cache, insensitive to the number of cache ways allocated to it.
- Rabbit - Low miss rate of the shared cache, sensitive to the number of allocated cache ways to it.
- Devil - High miss rate of the shared cache, low reuse of cached data.

The creators of this scheme claim that devils should be insensitive to contention for shared resources but Zhuralev et al. found this to be incorrect. The authors found the opposite to be true as devils showed to be the most sensitive to contention in their experiments. The high miss rates of the devils cause these applications to issue a large number of memory and prefetch requests which causes contention of other resources than the cache itself: The memory controller, memory bus, and prefetching hardware. The authors performed tests by which they were able to show that the cache-contention was not the major factor in performance degradation in five out of six applications. The prefetching hardware was the major factor in all five of those applications with the DRAM controller being the second largest in four out of five applications. Based on these tests the authors conclude that the miss rate is a good heuristic for predicting contention as it highly correlates with the amount of DRAM controller, FSB, and prefetch requests.

2.4 Cryptography

This chapter introduces the terms and techniques related to cryptography later referenced in the report.

2.4.1 Advanced Encryption Standard

The Advanced Encryption Standard (AES) [23] is the most widely used symmetric cipher today and is standardized by the National Institute of Standards and Technology (NIST). AES is a block cipher, meaning it encrypts a block of bits, 128 bits in the case of AES, at a time. Although the block size is limited to 128 bits, keys of varying length—128, 192 or 256 bits—can be used. AES uses a substitution-permutation network and a mathematical construct called Galois, or finite, fields. These substitution-permutation operations are performed multiple times, 10 times for 128 bit keys, 12 times for 192 bit keys and 14 times for 256 bit keys. The level of security provided by AES increases as the number of rounds, and key size, increases and becomes a trade-off between computation time and required security. Knowledge of the inner workings of AES is not necessary for this thesis and will not be explored in detail. More information on AES can be found in Chapter 4 in Paar and Pelzl [24] as well as in the AES standard [23].
2.4.2 Block Cipher Modes of Operation

A block cipher by itself takes the key as well as the cleartext as input and produces a ciphertext. If the same cleartext was encrypted multiple times with the same key, the block cipher would produce the same ciphertext every time. This can compromise the security of the ciphertext. Therefore, block ciphers can be used in other modes of operation. There are five common different modes of operation for block ciphers: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) and Counter (CTR) mode [25]. Electronic Codebook mode is the aforementioned mode where a certain cleartext is always encrypted to the same ciphertext given the same key. Cipher Block Chaining combines the cleartext with the previous block’s ciphertext. This mode requires an initialization vector used when encrypting the first block as a substitute for the previous, non-existent, block. Both Output Feedback, Cipher Feedback and Counter mode turns the block ciphers into stream ciphers by generating a keystream that is used to XOR with the cleartext instead of directly feeding the cleartext to be encrypted into the block ciphers themselves. The Output Feedback and Cipher Feedback modes are variations on using the output of the previous block to encrypt the next block, using an initialization vector as the input for the first block encryption. Counter mode uses a counter as the input to the block cipher instead to generate the keystream. It is important that the counter is never reused with the same key as this would produce the same keystream twice, weakening the cipher. Due to its simplicity, Counter mode can be efficiently implemented in both software and hardware [26].

Neither of these modes of operation provides authenticity and is often left to other algorithms such as hash-based message authentication codes (HMAC). There exist algorithms that combine the encryption with the generation of such authentication codes. These are often called authenticated encryption (AE) or authenticated encryption with associated data (AEAD) algorithms. One of these is the Galois/Counter mode [27]. It combines the Counter mode of operation with a message authentication code based on Galois, or finite, fields—the same mathematical construct AES is based upon. This is the mode of operation of interest to this thesis, and in conjunction with AES is bennamed AES-GCM.

2.4.3 AES Implementations

There are many different libraries with implementations for common cryptographic primitives, and many of them naturally include AES. AES is such a common cipher that Intel has introduced specific instructions in the x86 instruction set for this purpose. The Intel Advanced Encryption Standard New Instructions (AES-NI) is an extension developed by Intel to the x86 instruction set containing instructions specifically for the computation of the AES cipher. These instructions have shown to increase performance by up to 7% compared to hand-tuned assembly implementations by Intel [28].

The AES implementation of interest to this thesis is OpenSSL[6]. While OpenSSL is a software implementation of many cryptographic primitives, it does utilize hardware support such as SSE, AVX and AES-NI where available. OpenSSL supports a multitude of cryptographic primitives, including AES-GCM, the algorithm of interest to this thesis. OpenSSL is very widely available, can be used without hardware support such as AES-NI, and is compatible with many different platforms.

2.5 The Data Plane Development Kit

The following section discusses the Data Plane Development Kit (DPDK) which is a framework for packet processing on Linux and, to a lesser extent, FreeBSD. The section introduces
2.5. The Data Plane Development Kit

various DPDK concepts and libraries within DPDK that can help with development of packet processing applications. DPDK was originally an Intel project but has since then become an open-source Linux Foundation Project. It contains various libraries and utilities to perform fast packet processing, but does not implement higher level functionality such as IP routing, or protocols such as TCP or UDP, which the Linux networking stack does. Instead, it facilitates lower level utilities in order to create applications which perform a wide variety of tasks required on the data plane. Since DPDK applications run separate from the kernel networking stack and solely in user space, it implements its own drivers for various network cards and other offloading devices. It also provides an environment abstraction layer to query and access hardware and CPU capabilities as well as multi-threading, memory allocation facilities such as rings and pools, various classification methods such as longest prefix match—useful in IP routing—hash algorithms and access control lists. DPDK also provides support for cryptographic hardware accelerators as well as software implementations of common cryptographic algorithms.

2.5.1 Memory Management

For simple applications, DPDK handles memory mostly by itself but it can also be managed manually by the application. The minimum memory management that must be done is to allocate a memory pool for storing packets in. When a packet is received at the receive queue of a bound network card, it is placed in this memory pool. Each packet is assigned an \texttt{rte_mbuf} struct which contains fields such as sequence number, timestamp, and a pointer to where the actual packet data is stored. The data for each packet is stored sequentially in memory; beginning with the \texttt{rte_mbuf} struct, followed by a section of application specific data—which is referred to as the "private" data of the packet—and finally the actual packet data. A memory area of 2048 bytes is reserved by default for each packet (in addition to the \texttt{rte_mbuf} struct and the private data), enough to store packets for the commonly used 1500 byte maximum transmission unit, the maximum amount of data that can be sent in a single network layer transaction. If the packet does not fit in the 2048 byte area, multiple areas can be allocated and chained together using multiple \texttt{rte_mbuf} structs and the next pointer of the \texttt{rte_mbuf} struct. The \texttt{rte_mbuf} struct is cache aligned, meaning that it is allocated at the beginning of a cache line to void fetching other packets’ data. This layout means applications using DPDK have strong spatial locality when handling the packet’s \texttt{rte_mbuf} struct as well as its data at the same time, but less so when handling multiple \texttt{rte_mbuf} structs in a row without touching their respective packets’ data. Applications can process each packet by using a pointer to the \texttt{rte_mbuf} struct and then either free it, effectively dropping it, or transmit it on a network port. For more advanced applications, DPDK provides a wide variety of memory management utilities such as prefetching memory into CPU caches, handling memory allocation on different sockets in non-uniform memory access (NUMA) systems, as well as general-purpose memory allocators and data structures. Further information on memory management in DPDK, refer to the DPDK documentation on the Memopool library [29]

2.5.2 Poll Mode Drivers

In DPDK, packets are retrieved from and inserted into different devices using Poll Mode Drivers (PMDs) [30]. The reason for DPDK utilizing poll mode drivers, where hardware devices are polled for data by the driver, is to avoid hardware interrupts that would otherwise decrease performance. These drivers are also run in user space instead of kernel space to reduce the number of context switches between user and kernel space. Because of this, network devices have to be explicitly unbound from the kernel and bound to DPDK-compatible drivers. These PMDs are also useful as they abstract away the device itself and thereby allow for various physical and virtual devices to be interacted with in the same way. For exam-
2.5. The Data Plane Development Kit

ple, there are two PMDs—exposing the same API as any other PMD for a physical network card—that allow for the possibility to utilize other packet sources and destinations than actual Ethernet ports; a libpcap-based PMD and a ring buffer-based PMD. The libpcap-based PMD can be used to send packets to a DPDK application directly from a pcap file—a file format to store packet streams—on disk using libpcap or from a physical or virtual Ethernet device that is bound to the kernel. Files or Ethernet devices, physical or virtual, can be supplied to each pcap PMD for both receive and transmit queues. A file supplied to the receive queue will be used as input to the receive buffer of the software device and the packets sent on the device will be printed to a file supplied to the transmit queue. The ring-based PMD is useful when there are no Ethernet devices on the machine as packets can be inserted into the receive queue of the PMD by another running DPDK application. This allows for the use of packet generators as a packet source without using any physical or virtual Ethernet devices or reading from a pcap file. The PMD interfaces are also used for other purposes in DPDK such as enqueuing and dequeuing cryptographic operations in the Cryptodev library as well as enqueuing and dequeuing events in the Eventdev library [31][32].

2.5.3 Parallelization Support

As mentioned, DPDK has support for various parallelism techniques. Below, a few key libraries and features relevant to this thesis are presented.

**Ring Buffer Library**

The ring buffer library in DPDK is a thread-safe, lock-less, ring buffer implementation [33]. A ring buffer is a fixed size circular buffer that is written to at a head pointer and read from at a tail pointer, see [2.2]. Writing to the ring buffer advances the head pointer and removing from the ring buffer advances the tail pointer. The buffer only stores pointers to `rte_mbuf` structs to avoid unnecessary copy operations. If the ring buffer is full, no effort is made to buffer this data. This lends itself well to streaming applications, such as packet processing, where it is not possible to simply process all the incoming data if the application is overwhelmed. In such a case, recent data is more valuable and old data is discarded. The ring buffer library supports both single producer and single consumer as well as multiple producers and multiple consumers. This works very well for distribution of packets either from a single core to multiple cores or from multiple cores to a single core.

![Ring buffer with seven out of eight slots occupied](image)

**Event Device Library**

DPDK has a library called Event Device [32] (Eventdev) which introduces an event driven programming model for packet processing. This enables more scalable parallelization solutions by providing a scheduler that is responsible for distributing the events provided to it over event queues set up by the application. It is possible to use a hardware device as the
2.5. The Data Plane Development Kit

event device for scheduling, but DPDK also provides a default software event device which can perform scheduling when run on its own core.

The events in Eventdev are represented as \texttt{rte\_event} structs which hold information about the event as well as a pointer to a specific \texttt{rte\_mbuf} struct. Events can be created in multiple ways: They can be created manually by specifying the different fields but they can also be created by the scheduler if an Ethernet device, where incoming packets are received, is supplied to Eventdev. If the events are created manually, then they must be fed to the scheduler by using an Eventdev port. These ports are the gateways into the Eventdev scheduling environment. In Figure 2.3, a flow chart of the Eventdev scheduling environment can be seen. Each Eventdev port can be connected to multiple Eventdev queues. When a set of events is enqueued on a port, the scheduler looks at each of these events and distributes them based on the queue id field of each event. This queue id specifies a certain queue and when an event has been enqueued on such a queue it can be dequeued by all ports connected to that queue. A port does not have to be connected to a queue to enqueue onto it however. In contrast to the ring buffer library, the ring buffers used for events store copies of \texttt{rte\_event} structs instead of just the pointers to them. This means that each event is copied when enqueued and again when dequeued.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{eventdev_flow_chart.png}
\caption{Eventdev flow chart}
\end{figure}
There are multiple scheduling types for the scheduler that dictate how the scheduler treats events coming to and going from an event queue. Atomic is the most constraining scheduling type which makes sure that events within a single atomic flow can only be processed one at a time, preserving the order in which the events were enqueued into the system as well as guaranteeing the processing order. Ordered preserves the order that the events were enqueued in by reordering the packets after they have been processed but the events may be processed in any order. Parallel is the least constraining scheduling type which will allow events of a flow context to be processed and output in any order.

There are multiple configurations that can be made in Eventdev. The following are the configurations that can be made that dictate how the scheduler functions in general:

- **dequeue_timeout_ns** - The dequeue timeout duration can be used to allow for the unit dequeuing from an event port to wait for a set amount of time, if no events are available, before returning. If this is set to 0, the default value will be used. It should be noted that this value appears to be ignored in the default software scheduler implementation in DPDK.

- **nb_events_limit** - The maximum number of events that can be present in all of the event queues at a time. The default software scheduler in DPDK has an upper limit on the maximum number of events of 4096.

- **nb_event_queues** - The number of event queues to reserve. When launching the scheduler, the number of configured event queues must be the same as this value.

- **nb_event_ports** - The number of event ports to reserve. When launching the scheduler, the number of configured event ports must be the same as this value.

- **nb_event_port_dequeue_depth** - The maximum number of events that can be dequeued from a port at a time. This value is ignored in the default software scheduler.

- **nb_event_port_enqueue_depth** - The maximum number of events that can be enqueued on a port at a time. This value is ignored in the default software scheduler.

- **event_dev_cfg** - A bitmask for possible config flags. At the time of writing the only possible flag is RTE_EVENT_DEV_CFG_PER_DEQUEUE_TIMEOUT which tells the scheduler to override the global timeout and to use the one that can be specified in each dequeue call instead.

In addition to the general settings there are also some more specific configurations to be made for the Eventdev components. The following are the possible configurations that can be made for the event ports:

- **new_event_threshold** - If this is set then the number of events with the operation type RTE.EVENT.OP.NEW in the system cannot exceed this value.

- **dequeue_depth** - This can be set to individual values per port if nb_event_port_dequeue_depth is not used. This value is ignored in the default software scheduler.

- **enqueue_depth** - This can be set to individual value per port if nb_event_port_enqueue_depth is not used. This value is ignored in the default software scheduler.
• disable_implicit_release - If this is set to true, the events must be explicitly released by setting their operation type to RTE_EVENT_OP_RELEASE or RTE_EVENT_OP_FORWARD and enqueuing them back into the system.

Lastly, these are the configurations that can be made for event queues:

• nb_atomic_flows - The number of atomic flows in the event queue. Events within an atomic flow can only be dequeued by a single event port at a time if atomic scheduling is enabled.

• nb_atomic_order_sequences - The size of the reorder buffer of the event queue used if ordered scheduling is enabled.

• event_queue_cfg - A bitmask for possible config flags. The two possible flags at the time of writing are RTE_EVENT_QUEUE_CFG_ALL_TYPES and RTE_EVENT_QUEUE_CFG_SINGLE_LINK. All types means that the queue allows for enqueues of events of all different scheduling types; ordered, atomic, and parallel. Single link allows for optimizations in scenarios where the event queue is only connected to a single event port.

• schedule_type - How events should be scheduled in the queue. The possible values are RTE_SCHED_TYPE_ORDERED, RTE_SCHED_TYPE_ATOMIC, and RTE_SCHED_TYPE_PARALLEL.

• priority - The priority of events in this queue. This priority is used to determine the scheduling of events to a port which is dequeuing from multiple event queues at once.

The default software scheduler for Eventdev included in DPDK works by constantly polling the ring buffers of the event ports, where the events are stored after being enqueued. Once it manages to dequeue a set of events it will distribute these to the mapped event queues based on how it has been configured.

2.5.4 Cryptography Support

The Crypto Device (Cryptodev) library in DPDK utilizes hardware or software devices to provide cryptographic support to DPDK applications [31]. These cryptographic devices are managed by PMDs and work can be enqueued for cryptographic operations and dequeued when complete. DPDK supplies several different software crypto devices for different implementations such as OpenSSL and Intel’s Multi-Buffer Crypto. It also has support for a host of hardware offloading devices. The software devices also work well with multiple cores as they can provide a queue per core which means each worker thread can enqueue crypto operations onto its own crypto queue. If hardware crypto devices are used, the transfer times to and from the hardware device should be taken into consideration when queueing cryptographic operations. However, some software crypto devices—such as the OpenSSL device—do not require any additional consideration as the cryptographic operations are performed on the same thread as they are enqueued.

[31] https://github.com/intel/intel-ipsec-mb
2.5.5 Reorder Support

The order in which the packets arrive at a network forwarding node is important to preserve when forwarding the packets as different protocols and applications may function worse with out-of-order packets. To help with this, DPDK provides a packet ordering library called the Reorder library [34]. This library makes use of a sliding window which is used to decide whether a packet is deemed worth ordering or not. It does this by looking at the sequence number of the packet—provided by the application—and comparing it to the window which consists of a minimum sequence number and a window size. If the packet’s sequence number is lower than the minimum value, the packet is denied ordering and the documentation suggests immediately forwarding this packet as it is late. If the packet’s sequence number falls between the upper boundary of the window and a configurable offset above it, it will increase the minimum value and cause the window to slide. Packets with sequence numbers higher than the upper boundary plus a configurable offset are also deemed not worthy of ordering and denied insertion. These packets are very early and it is up to the application to decide what to do with them.

In addition to the minimum sequence number and the size of the window, two circular buffers are used to implement this sliding window. One buffer is called the order buffer and the second is called the ready buffer. The order buffer is where the packets are inserted into that fit within the window and these packets are inserted with an offset relative to the minimum value so that the packets within the window are sorted in relation to each other. The ready buffer is where packets from the order buffer end up when the window slides and the minimum value becomes larger than their sequence numbers. When the sorted packets are to be extracted, they are taken from the ready buffer. If more packets than what was available in the ready buffer were requested, packets are also taken from the order buffer. This will also cause the window to slide for each packet that is taken from the order buffer.

Receive Side Scaling Support

Receive Side Scaling is a technique to distribute packets to different cores directly from network cards. For this technique to be useful, the network card in question has to have multiple receive queues. Each of these queues will then be filled with individual IP flows, usually based on the source and destination addresses, source and destination ports and the transport layer protocol used [35]. The Ethernet device becomes responsible for the load balancing, removing that responsibility from the software application.

2.6 IPsec

IPsec is a specification of a suite of protocols that handle various security-oriented services for traffic at the Internet layer. It is designed to provide a standard for cryptographically-based security to the IPv4 and IPv6 protocols. It was proposed by the Network Working Group in 1995 but has since then gone through various revisions and is currently specified in RFC 4301 [36]. IPsec can provide—through its underlying protocols—confidentiality, integrity, and authenticity. This section introduces IPsec and, more importantly, the Encapsulating Security Payload protocol of the IPsec suite.

The Internet Key Exchange (IKE) protocol [37] provides an automatic way to negotiate which security features to provide for the IP packets and how these services should be provided. IKE is responsible for performing mutual authentication of the parties that wishes to communicate, and for creating the Security Association that will facilitate the communication. The parties will negotiate which protocols, algorithms, and options will be used and then generate cryptographic keys for the Security Association. Security Associations can also be
IPsec provides two different protocols for communication: Authentication Header (AH) and Encapsulating Security Payload (ESP). ESP provides both confidentiality, integrity, and authenticity for IP packets while AH only provides integrity and authenticity. The IPsec standard does not require an IPsec implementation to support AH while ESP support is a must.

### 2.6.1 Encapsulating Security Payload

This thesis focuses on Encapsulating Security Payload (ESP). As mentioned, it provides confidentiality, integrity, and authenticity for the IP packets it encapsulates. In addition to this, ESP can also provide anti-replay protection as well as limited traffic flow confidentiality.

A replay, or playback, attack is an attack that is performed by replaying older packets sent over the network. For example, an attacker can record a valid authentication request and then later replay that request to gain access to an otherwise restricted resource. ESP employs sequence numbers in order to facilitate anti-replay protection. Usage of this functionality is optional and at the discretion of the receiver. If employed, the receiver keeps a sliding window of the sequence numbers it has seen recently. If a packet contains a sequence number already seen, or it is older than the oldest sequence number in the sliding window, it is discarded.

ESP can, as mentioned, also provide limited traffic flow confidentiality. Traffic flow confidentiality means to protect information that could otherwise be extracted by analyzing the traffic flow characteristics, e.g., the size of packets and the delay between packets. Providing full traffic flow confidentiality can have adverse effects on performance and on other users of the network. It also requires more infrastructure than what a single protocol can provide, such as specialized routing of packets. This kind of confidentiality is more common in military than commercial or private communication. The traffic flow confidentiality that ESP provides is padding of the packets, beyond any regular padding that is required by the standard, and any padding that might be required by the cryptographic algorithms used.

ESP is defined agnostic of which encryption and authentication algorithms are used. The encryption and authentication algorithms available are described in RFC 7321. Separate RFC documents are available for specific algorithms for use with ESP.

ESP can work in two distinct modes: transport or tunnel mode. These modes will be described more in detail in the following two sub-sections.

**Transport Mode**

In transport mode, an ESP header is inserted between the IP header and the transport layer header, see Figure (note that ESP works with any transport layer protocol, TCP is only used as an example in the following figures). Consequently, the original IP header is not encrypted, nor checked for integrity or authenticity. Therefore, information about which two entities are communicating is not kept secret.
Transport mode has a variable size overhead, with a minimum of 10 bytes. 8 bytes for the ESP header and 2 bytes for the padding length and next level header fields in the ESP trailer, assuming no padding was required. The payload data must be padded to align it to 4 bytes, and can be further padded to align it with the block size of the block cipher used—which is variable depending on the specific algorithm used—as well as to provide traffic flow confidentiality. Additionally, 0 - 16 bytes of overhead is added for the integrity check value (ICV). This value is used to verify the integrity and authenticity of the packet payload. The presence, and size, of an ICV depends on the authentication algorithm used. Due to this low amount of overhead, transport mode is useful when there is a limited amount of resources and bandwidth available [44, 45], as well as when source and destination confidentiality is not required.

**Tunnel Mode**

Since this thesis focuses on tunnel mode, a more in-depth explanation of this mode follows. In tunnel mode, the whole original IP packet is considered the payload and the ESP header and trailer is applied around it. A new IP header is then prepended, potentially with different source and destination addresses, see Figure 2.5. This mode provides better security, since the original IP header is now also subject to encryption and authentication, at the cost of more overhead—both in processing and size.
2.7. AES in Galois/Counter mode for ESP

which Security Association this packet belongs to. A sequence number follows which is used to mitigate anti-replay attacks as described earlier. The sequence number is a required field of the ESP header even if the receiver does not employ anti-replay protection.

Some encryption algorithms require synchronization data, such as an initialization vector. This is prepended to the payload and considered part of the payload data. The payload data is then padded to align it with the 4 byte requirement as well as any requirements of the encryption algorithm used. Following the padding are two fields stating the length of the padding as well the protocol of the next header in the packet. In tunnel mode, the next header field is always IPv4 (4) or IPv6 (41). At the end of the packet is an optional integrity check value (ICV), similar to transport mode.

![ESP tunnel mode header](image)

Figure 2.6: ESP tunnel mode header

Tunnel mode adds more overhead than transport mode, a minimum of 30 bytes for IPv4 and 50 bytes for IPv6; 20 (with IPv4) or 40 (with IPv6) bytes for the new IP header and 10 bytes for the ESP header and trailer. This means that it is less useful in resource constrained environment but it also provides better security as it also encrypts and authenticates the original IP header.

2.7 Advanced Encryption Standard in Galois/Counter mode for Encapsulating Security Payload

The IPsec implementation in this thesis is, as previously mentioned, using ESP in combination with AES in Galois/Counter mode (GCM), abbreviated AES-GCM-ESP. This particular combination is specified in RFC 4106 [46].

The AES GCM authenticated encryption with additional data algorithm (AEAD), hereby referred to as AES-GCM, has four inputs: a key, a nonce, a cleartext, and any additional authenticated data (AAD). The algorithm produces two outputs: a ciphertext and an authentication code, or tag as it is referred to in RFC 4106. It is important to note that the GCM nonce is also called initialization vector but is not to be confused with the one defined in ESP. The key is shared when establishing the Security Association and is 128, 192 or 256 bits long.

When using AES-GCM-ESP, cryptographic synchronization data—in this case an Initialization Vector—is included in the ESP payload, see Figure 2.7.
This initialization vector must be eight bytes long and must not repeat for any given key. Reusing the same key and initialization vector compromises the security of AES-GCM-ESP.

The ciphertext consists of the cleartext, any required and desired padding, as well as the padding length and next header, encrypted using AES-GCM.

The nonce passed to AES-GCM consists of a four byte salt and the eight byte initialization vector, see Figure 2.8. The salt is chosen when establishing the Security Association together with the key and must be kept constant during the lifetime of the Security Association.

The additional authenticated data (AAD) is eight bytes long and consists of the Security Parameters Index and the sequence number, see Figure 2.9. When using extended sequence numbers, the AAD is instead 12 bytes long since the sequence number is 8 bytes.

In addition to the ciphertext, AES-GCM produces an authentication code—called integrity check value (ICV) in ESP. This code is either 8, 12 or 16 bytes long. This ICV is required to be included—contrary to what ESP requires—since AES-GCM provides no integrity protection if no authentication code is included.

The minimum overhead incurred using AES-GCM-ESP in tunnel mode is 46 bytes for IPv4, and 66 bytes for IPv6, per packet: 20 or 40 bytes for the outer IPv4 or IPv6 header respectively, 10 bytes for the ESP header and trailer, 8 bytes for the initialization vector, and a minimum of 8 bytes for the integrity check value. Note here that the salt is not sent in the packet, this data is shared when the Security Association is established.
2.8 Related Work

There has been much research done in the areas of parallelism within and surrounding packet processing and encryption. There are many articles presenting different approaches to parallelizing encryption algorithms such as AES [47, 48, 49, 50] but there are also papers investigating the parallelization and evaluating the performance of IPsec as a protocol [51, 52, 53]. Most of these papers describe several of the aspects of the problem addressed in this thesis even if they are not always directly applicable to our thesis as the authors most often develop or design hardware solutions or investigate parallelization with a higher granularity, such as within the ciphers themselves.

This section discusses several papers from these areas in depth that are interesting with regard to this thesis. A short presentation of the different papers discussed follows in this paragraph, as well as a motivation to why they are interesting to this thesis. The paper “Processing Data Streams with Hard Real-time Constraints on Heterogeneous Systems” by Verner, Schuster and Silberstein [54] is presented in Section 2.8.1. It deals with offloading cryptographic work to a hardware accelerator and how this can affect performance. This is interesting as parallel processing using hardware accelerators, both for scheduling and cryptographic operations, can be applied to this thesis. In Section 2.8.2, the paper “Elastic Scaling of Data Parallel Operators in Stream Processing” by Schenider et. al [55] is presented. It presents an algorithm for elastic scaling in data stream processing. Although an elastic scaling solution is out of the scope of this thesis, it is interesting to keep in mind how such a solution can help to find the best performing configuration of a stream processing application. Heinze et al. discuss latency spikes that arises as a result of elastic scaling in their paper “Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems” [56] presented in Section 2.8.3. As with the previous paper, elastic scaling is out of the scope of this thesis but it is important to keep in mind how an elastic solution could affect an application from the start which is why this paper is interesting to this thesis. Koronla et al. presents an implementation of the IPsec protocol suite on an FPGA in their paper “FPGA implementation of IPsec protocol suite for multigigabit networks” [51], presented in Section 2.8.4. This paper is interesting to use as a comparison on how an x86_64 solution can compare to a specialized FPGA based solution. The following two papers, “MIDeA: A Multi-parallel Intrusion Detection Architecture” by Vasiliadis et al. [57] and “High Speed Network Traffic Analysis with Commodity Multi-core Systems” by Fusco and Deri [58], presented in Sections 2.8.5 and 2.8.6 deal with another problem domain than this thesis but both of them address high speed data processing of network traffic. These papers deal with real world concerns that appear when dealing with data stream processing. The two last papers presented, “Evaluating the Impact of Simultaneous Multithreading on Network Servers Using Real Hardware” by Yaoping et al. [59] and “The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications” by Saini et al. [60] in Sections 2.8.7 and 2.8.8 respectively, deal with simultaneous multithreading (SMT) and how this affects performance. These papers are interesting to compare with on how our application scales with SMT and how it can take full advantage of it.

2.8.1 Processing Data Streams with Hard Real-time Constraints on Heterogeneous Systems

In an article by Verner, Schuster and Silberstein [54], an algorithm for providing low latency while maintaining high throughput when offloading data stream processing onto a GPU is presented. The authors begin by describing the issues of utilizing the GPU as an accelerator. These issues encompass communication overhead between CPU and GPU as well as the non-linear relation between input size and throughputs. The communication overhead is a result of the bus over which the CPU and GPU communicate. The communication is limited in terms of throughput and it has a high latency. The GPU also requires the data it is to operate
on to be sent as a batch since it performs its computations on a set of data all at once. Since the computations performed on the streams in this article is AES in Cipher Block Chaining (CBC) mode, they cannot be internally parallelized and the encryption of block $N$ depends on block $N - 1$. This implies that each stream must be processed by its own core, which means that the throughput can vary for a set input size when varying the number of streams.

The scheduling algorithm proposed in the paper works by deciding if a batch of jobs should be handled by the GPU or the CPU. The incoming streams are separated into a series of jobs. Each job is defined as a tuple consisting of the number of elements to be processed and a deadline. The deadline is the maximum amount of time the system has to process the job. These deadlines are used as an order of priority when assembling the batches. After a batch has been assembled it is tested for schedulability. A set of jobs is considered schedulable on a computation unit if there exists a schedule such that no stream misses its deadline. If a job is not schedulable on the GPU it is passed on to the CPU which provides lower latency, as long as it is not overloaded with work. The issue with this approach is that the search space when testing for schedulability grows exponentially with the number of streams, making exhaustive search impractical. To deal with this the authors propose a heuristic that they call the rectangle method. The search space is reduced by setting a lower and upper bound on deadlines and stream rates of jobs that are schedulable on the GPU. The lower bound on the deadlines is based on an estimated processing time of the GPU, including data transfers to and from the GPU. The reason for the name of the algorithm is given when plotting the bounds of the deadlines and stream rates which forms a rectangle. All of the jobs that fall within the rectangle are considered schedulable. The rectangle method is described as the process of testing all rectangles for schedulability. The number of rectangles in an $n \times n$ space is $O(n^4)$. This can be reduced to $O(n^2 \log(n))$ by setting the lower stream rate bound to zero (slower streams improve accelerator utilization) and then using binary search to find the upper bound. The algorithm produces deadline-compliant schedules, otherwise marking the input as unschedulable. Experiments performed by the authors showed that usage of the rectangle method in their system resulted in a four-fold speedup over a CPU-only system for streams with 50 ms deadlines.

This article is not directly applicable to this thesis in the sense that stateful execution is not relevant for the proposed implementation. The usage of a hardware accelerator, however, is interesting as it can also be applied to future work on this application, especially for encryption and decryption.

### 2.8.2 Elastic Scaling of Data Parallel Operators in Stream Processing

An algorithm for elastic scaling of parallel computations is presented in a paper by Schneider et al. [55]. In the paper the authors describe the issue of higher degrees of parallelization not always resulting in an increase in performance. They propose a solution to this problem in the form of an algorithm which scales the number of active threads based on performance measurements.

The authors present different classes of threads which they call dispatch threads, worker threads and alarm threads. In their algorithm, a single processing node has one dispatch thread, several worker threads, one alarm thread, and one work queue. The work queue is a global queue which has work put into it by the dispatch thread and work retrieved from it by the worker threads. The reason given for using a global queue instead of a private work queue for each thread is that the granularity of parallelism is coarse. In addition to dispatching the incoming work to the work queue, the dispatch thread is also responsible for controlling the number of active threads in order to optimize performance. It does so by creating or waking up sleeping worker threads, or putting active worker threads to sleep.
Lastly, the alarm thread is responsible for periodically waking up and notifying the dispatch thread when it is time to re-evaluate the number of active worker threads, denoted as the thread level. The thread level is evaluated by keeping records of the peak throughput for all visited thread levels as well as the current throughput for the present thread level. Their definition of throughput is generalized to be the number of elements processed by the system per second. By comparing the current throughput to the peak throughput of the current thread level, performance decline can be detected. Comparing peak throughput between thread levels can also be used to see whether an increase in thread level actually gave an increase in performance. Initially, the thread level should be increased as long as the peak performance increases. When a level is reached where there is no significant increase in performance, there is stability from below. This approach assumes that if a performance increase was gained when going from thread level \( N - 1 \) to \( N \), then there is probably a performance gain when going from thread level \( N \) to \( N + 1 \). Likewise, if there is no significant gain in performance when going from thread level \( N - 1 \) to \( N \), then the performance will likely remain the same or even decrease if the thread level is increased to \( N + 1 \). This is called stability from above. If the current throughput falls significantly below the peak throughput, the algorithm will lower the thread level. After lowering the thread level, the algorithm enters a peak state which means it will only stay on this lower level if, during the next evaluation, the current throughput is higher than the throughput of the previous thread level. To converge faster to a stable thread level, the algorithm decays the peak throughput by a small percentage each evaluation where the current throughput is lower than the peak throughput. When the current throughput is higher than the peak throughput, the peak throughput is set to the current throughput. If the current throughput is significantly higher than the peak throughput, the peak throughput of the thread levels above the current thread level are invalidated. This causes the algorithm to being probing into those levels. The authors claim these properties of the algorithm makes it quickly adapt to new situations where the workload changes.

Based on an experiment performed by the authors in a synthetic benchmark, they conclude that the algorithm works well in terms of elasticity. They note that smaller workloads cause the algorithm to restrict the number of threads used as the parallelization overhead becomes larger in proportion to the workload. In another experiment performed in a real scenario using a radio imaging application, the authors concluded that the algorithm settled at a throughput within 10% of the one of the best static configuration.

This article has the same objective as this thesis which is to optimize the performance of the application. It is, however, out of the scope of this project to implement an elastic solution but it is interesting to keep in mind the potential benefits of such an addition to the application proposed in this report for future work.

### 2.8.3 Latency-aware Elastic Scaling for Distributed Data Stream Processing Systems

In a paper by Heinze et al. [56], a solution is presented to a problem created by elastic scaling; latency spikes. When a data stream processing system is scaled down, the operator(s) of the host being taken down has to be moved to another host which can cause spikes in the latency of events routed to that operator. This is a problem as systems may have latency guarantees that could be broken if large latency spikes occur. The authors use a table which has been created based on measurements to look up the estimated time of a move based on the known factors such as state size, total number of moved operators and host utilization. Finally, they present a cost model which is added to the decision-making when scaling the system in an attempt to reduce the impact of the latency spikes. This paper provides a different perspective on elastic scaling by focusing on the issues presented by it and how to mitigate these. Elastic...
scaling is, as mentioned, outside the scope of this thesis but it is important to contemplate factors that are affect latency as well as throughput.

### 2.8.4 FPGA implementation of IPsec protocol suite for multigigabit networks

In a paper by Koronla et al. [51], an FPGA implementation of IPsec is presented. This paper does not go into much detail about the implementation which makes it an interesting paper to compare with, especially when it comes to the performance analysis, but not to take inspiration from. The paper claims that CPU based implementations require dozens of cores to reach a modest 1 Gbps—referring to the StrongSwan project [8]—something this thesis aims to prove incorrect. Their implementation achieves 10 Gbps in the best case using Data Encryption Standard (DES) as the encryption algorithm and 5.5 Gbps in the best case using AES as the encryption algorithm. They designed a prototype to detect bottlenecks and guide further development. One major bottleneck they encountered with the prototype was that authentication of the packets—using the HMAC-SHA1 authentication algorithm—occupies 86% of the processing time. Another problem was that their prototype design was not pipelined. This resulted in other sections, responsible for encryption and other IPsec operations, not being utilized during authentication. Their prototype solution was improved by pipelining the design, allowing for multiple packets to be processed at any given time and for full utilization of the FPGA. Their HMAC-SHA1 implementation was also improved, halving the time required to compute the hash. Their implementation performs much better with larger packet sizes than smaller ones.

### 2.8.5 MIDeA: A Multi-parallel Intrusion Detection Architecture

The issue of high-speed data stream processing is present in many different applications. In a paper by Vasiliadis et al. [57], the problem of detecting network intrusions on high-speed networks is addressed. The authors present a parallelized implementation of a Network Intrusion Detection System (NIDS) they claim can reach the throughput required to process network traffic at a rate up to 7 Gbps.

Their implementation is based on commodity hardware such as off the shelf CPUs, GPUs and network cards to make it a cheaper and more generic solution compared to ready-made commercial solutions. Their design is based on separating the cores as much as possible to avoid sharing resources. An important area where they have performed such separations is to the driver’s output ring buffer. This buffer is where the driver places the packets it dequeues from the network card’s receive queue. In order to avoid costly locks or atomic operations on the ring buffer, the authors allocate a separate ring buffer for each core. This means that the driver can enqueue packets for a single core at a time but it also creates the issue of load balancing these buffers. To deal with this the implementation makes use of Receive Side Scaling (RSS), which means each core gets its own receive queue on the network card and the load balancing is also taken care of by the network card. A hash-function is used to make sure that all packets of the same one-way stream are placed on the same core for processing. As the use-case in the article considers all streams to be two-way, the authors calculate another hash in order to match cases where packets are going the opposite way to the same core. The authors also describe the need to pin a worker process to its own core to improve the spatial locality. In the proposed implementation, the CPU cores are responsible for preprocessing the packets and ready them for inspection while the GPU is responsible for the inspection itself. After the CPU cores have preprocessed the packets they then place them in a buffer to be sent to the GPU. This buffer is used to batch packets together in order to reduce the overhead of transferring data back and forth to the GPU memory per packet.

[https://www.strongswan.org/](https://www.strongswan.org/)
By batching packets from multiple cores together, the implementation manages to parallelize the inspection of the packets in addition to the preprocessing.

Further, the authors also describe some modifications made to the pattern matching engine used to inspect the packets as well as how their implementation supports multiple GPUs. The article also mentions some performance optimizations in the form of how data is loaded into the GPU memory and how the GPU and CPU work is pipelined to increase performance. The pipelining is achieved by having an additional buffer where a batch is moved to when the batch is full to allow for the CPU to start filling the next buffer while the previous one awaits processing by the GPU. The authors tested their program in experiments using two Intel E5520 Quad-core CPUs at 2.27GHz and 8192KB of L3-cache, one Intel 82599EB 10GbE network card and two NVIDIA GeForce GTX480 graphics cards. With this setup they achieved a maximum throughput of 7.22 Gbps without packet loss when using a synthetic packet stream and a packet size of 1500 bytes.

The problem presented in the article is quite close to the one addressed in this thesis with the difference being the areas of application. The material concerning GPU offloading is not relevant to this report as it is outside of the scope but the decisions made by the authors with regards to the general architecture of their design are very useful. For example, the methods used for resource separation are great tools for designing similar systems. The performance of the application presented in this paper cannot be compared to this thesis, since the processing done is completely different.

### 2.8.6 High Speed Network Traffic Analysis with Commodity Multi-core Systems

In an article by Fusco and Luca [58], the issue of network interface drivers not being optimized for parallelized packet processing applications is brought up. The authors claim that even though there are different hardware solutions such as Receive Side Scaling (RSS) to efficiently split the packet stream over multiple receive queues, the network drivers are not built to utilize it, since the interface of the driver towards the user space is, in most operating systems, set so that it is not possible to distinguish a legacy 100 Mbps network card from a modern 10 Gbps one. This forces the drivers to merge all of the hardware queues into one to maintain the interface towards user space. In relation to this issue they talk about the problem with interrupts thrashing the cache. This occurs if a core that is not polling from a certain receive queue is assigned an interrupt request for that queue. They also mention how copying each packet multiple times must be avoided as the memory bandwidth is of major concern in high throughput packet processing applications. Their solution to these problems is a driver which utilizes multiple virtual capture devices which each collect packets from their own network card receive queue. This allows for each processing thread to have an individual receive queue in form of a ring buffer which is only shared between it and a polling thread in a single-producer-single-consumer relationship. The authors also suggest putting these two threads on the same core if the system has a simultaneous multi-threading technique, such as Intel’s Hyper-Threading, to improve cache-efficiency. To further improve performance they suggest setting the interrupt request affinity so that the polling threads are assigned interrupt requests from the device connected to their own receive queue. A part of their solution is to bring the information of which device is connected to which receive queue to user space to allow applications to utilize it.

This report is very relevant to this thesis in terms of the what the main real world concerns are regarding high speed packet processing. It is also very useful to put the work of this project in context of what limitations would exist around it in a full-scale implementation.
2.8.7 Evaluating the Impact of Simultaneous Multithreading on Network Servers Using Real Hardware

One interesting area to look at concerning parallelism is simultaneous multi-threading (SMT), used in techniques such as Intel Hyper-Threading. In a paper by Yaoping et al. [59], the impact of SMT on network servers is evaluated. Their results suggest that the effect of SMT is very dependent on workload, and that for network servers the additional parallelism can actually hurt performance instead of improving it. In almost all of their tests, running a single processor with SMT decreases performance compared to running only a single processor. Running a dual-core processor with SMT also decreases performance compared to SMT switched off. Using two processors with SMT switched off provides a performance increase compared to using only a single processor in the tests they performed, showing that the application can indeed take advantage of multiple cores. Their conclusions is that SMT does have a non-negligible OS overhead associated with it and that the benefits of using SMT are hard to benefit from.

This paper is interesting as a comparison to how our solution works when using simultaneous multi-threading.

2.8.8 The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications

In Saini et al. [60], the authors investigate the impact of Intel’s Hyper-Threading technology on processor resource utilization. Their findings show a different picture compared to the last paper discussed, with three of the four applications tested having a performance increase using Hyper-Threading. They also show that highly vectorized (SIMD) programs can not take advantage of Hyper-Threading as well as non-vectorized programs. They also found that memory contention is a large problem when trying to take advantage of Hyper-Threading. If the two hardware threads go deep into the memory hierarchy, the performance gain from Hyper-Threading is negligible and can even have worse parallel efficiency than when not using Hyper-Threading.

This paper is also interesting to compare our solution with in terms of how well it performs when using simultaneous multi-threading.

2.9 Analysis of Non-Academic Work on IPsec

There is little academic literature on IPsec, and next to none that discusses parallelism within ESP. Most papers present a solution together with benchmark results without going into any detail on how their solution was designed or implemented. Because of this, research was done on what has been achieved in the open-source space with parallelizing IPsec. Most of the work was found in the Linux kernel [9]. Unfortunately, most of the information is very vague and it is unclear what exactly is implemented and achieved in terms of throughput and latency, especially with a single Security Association.

The work of Steffen Klassert, part of the IT security company secunet Security Networks AG, has appeared a few times in work related to IPsec parallelism. In a presentation from 2010 [61] he notes that IPsec implementations up to that point did not parallelize within a single Security Association in order to preserve the packet order. He later specifies requirements of a parallel IPsec implementation. Among these, the requirement that parallelization should be possible even within a single Security Association suggests that there has been some work done in this area, but it is unclear what and how well.

Further, in a presentation from the Netdev 1.2 conference 2016 \[62\], Klassert presents the status of IPsec in the Linux kernel. He presents performance numbers reaching 155.6 Gbps with 16 bidirectional flows, a total of 32 Security Associations. He mentions that the implementation utilized about 32 cores, suggesting that each Security Association was distributed its own core. During a discussion in the presentation he also mentions that the processing of packets in a single Security Association will always be placed on the same core, further suggesting that the implementation does not parallelize within a single Security Association.
This chapter introduces the different design and implementation decisions made when creating the application presented in this thesis. It introduces the general structure of the application, covers how the ESP processing is done and then goes into detail about the different implementations.

3.1 Application Structure

The different implementations can be switched between by using command line arguments to the application. Multiple configurations that dictate how each implementation behaves can also be made through command line arguments. A single-threaded implementation was created to supply a baseline system to which the multi-threaded implementations can be compared. In addition to the single-threaded implementation, three multi-threaded implementations were created. One based on the DPDK ring buffer library and two based on the DPDK Eventdev library, see Section 2.5.

The application receives a constant stream of packet batches that must be processed according to the ESP standard. The processing, as discussed in Section 2.6.1 consists of either inserting an ESP and IP header, encrypting the packet and generating an authentication code, or—in the case of receiving an ESP packet—removing the outer IP and ESP headers and decrypting and authenticating the packet. This is true for all implementations and is described in more detail in Section 3.2. Each stage in the list below is performed on all packets in the batch before moving on to the next stage. The reason for this structure is the batch oriented structure of the DPDK APIs. The packet flow through the application is as follows:

1. A batch of packets is received in the form of an array of rte_mbuf structs.
2. A loop iterates over the packets and assigns each packet a sequence number.
3. The packets are enqueued onto the workers’ inbound queue(s).
   (Not performed in the single-threaded implementation).
4. The packets are dequeued from the workers’ inbound queues(s).
   (Not performed in the single-threaded implementation).

5. The packets are processed according to the Encapsulating Security Payload standard, see Section 3.2.

6. The packets are enqueued onto the workers’ outbound queue(s).
   (Not performed in the single-threaded implementation).

7. The packets are dequeued from the workers’ outbound queues(s).
   (Not performed in the single-threaded implementation).

8. The packets are reordered.
   (Not performed in the single-threaded implementation).

9. The packets are then transmitted.

The structure can also be seen in Figure 3.1. Items 1, 2, and 3 are considered “receive tasks” and can be seen in blue coloring in the figure. Items 4, 5, 6 items are considered “worker tasks” and can be seen in green coloring in the figure. Items 7, 8, 9 is considered the “transmit tasks” and can be seen in yellow color in the figure. Items 3, 4, 6, 7, and 8 are not performed in the single-threaded implementation since they are not required. Only a single core is used in the single-threaded implementation which removes the need to send packets between cores and reordering is not necessary as all packets are processed in order.

---

3.2 Encapsulating Security Payload Implementation

The Encapsulating Security Payload (ESP) processing is carried out once for each packet. This processing is independent from any other packets and is a serial process. Therefore, the ESP processing is the same for all implementations and is described below.

3.2.1 To ESP

Unencrypted IPv4 packets arriving at the application and that matches the Security Association the application is set up for, should be encrypted and forwarded. This is done in four stages: pre-crypto, ESP, encryption, and post-crypto, see Figure 3.2.
3.2. Encapsulating Security Payload Implementation

When the packet arrives to the pre-crypto stage it is a full Ethernet packet and this stage removes the Ethernet header from the packet. It is then forwarded to the ESP stage. The ESP stage performs the following operations, in the order they are listed:

- It allocates memory for any necessary padding, the ESP trailer and the Integrity Check Value.
- It fills the padding with the mandated pattern and fills out the ESP trailer.
- It allocates memory for the outer IP header, ESP header and initialization vector.
- It fills out the outer IP header, the ESP header and the initialization vector.
- It constructs the nonce required for AES-GCM-ESP.
- It constructs the additionally authenticated data required for AES-GCM-ESP.
- It constructs the DPDK cryptographic operations required to perform AES-GCM-ESP.

The DPDK framework has utilities for freeing and allocating memory before and after the existing packet data using `rte_pktmbuf_adj` and `rte_pktmbuf_prepend` or `rte_pktmbuf_trim` and `rte_pktmbuf_append` respectively.

It is important to note that packets use network byte order, i.e. big-endian, and care must be taken to convert from, and to, the host byte order—if necessary, as on x86_64 which uses little endian—when writing and reading packets.

AES-GCM-ESP specifies that if no traffic flow confidentiality is desired, the implementation should use the minimum amount of padding necessary. It also specifies that the ESP payload needs to be aligned to four bytes. The final payload length is calculated by adding two bytes (for the ESP trailer) and aligning it to four bytes as shown in Listing 3.1.

```
Listing 3.1: Padded payload length calculation

length = RTE_ALIGN_CEIL(rte_pktmbuf_pkt_len(packet) + 2, 4);
```

The padding is filled using the default sequential scheme mandated by ESP, the monotonically increasing byte sequence: 1, 2, 3, ....

The IP header is filled as shown in Figure 3.3.
3.2. Encapsulating Security Payload Implementation

The Internet Header Length (IHL) field is set to 5, denoting that no optional headers are present in this IP header [63]. The identification field is set to an easily recognizable number, although in a real world scenario this value would change depending on the packet to aid the receiver when assembling a fragmented IP packet. The Flags field is set to 0x2, denoting that the IP packet should not be fragmented [63]. The Time To Live field is set to 255 for simplicity but should be set to something more sensible for a real world scenario. The Protocol field is set to 50, the number assigned by the Internet Assigned Numbers Authority for the ESP protocol [64]. The header checksum cannot be calculated yet as it requires information that has not yet been filled in, such as the ESP header and the encrypted packet data. The checksum is calculated and filled in in the post-crypto stage. The outer source IP address and destination IP address fields are filled in with the addresses specified in the Security Association.

The ESP header is filled as shown in Figure 3.4.

The Security Parameters Index is decided by the Security Association and is chosen to 1 for simplicity. The Sequence Number is a monotonically increasing number starting at 1 and is increased for each packet. Using 32-bit sequence numbers, the maximum number of packets that can be transmitted before the sequence number wraps is $2^{32} - 1 = 4294967295$ packets. One is subtracted since the sequence number 0 is never used. Wrapping the sequence number compromises the security of AES-GCM (since the sequence number is used as the initialization vector). RFC 4303 specifies that if anti-replay protection is enabled, which is the default, the sender must not send a packet which causes the sequence number to overflow. Further, in order to avoid the problem of compromising the security of AES-GCM, a new key has to be established. Handling such cases is outside the scope of this thesis as it requires the Internet Key Exchange protocol to do it. This does not have to be handled by the application presented in this thesis since it is out of scope and the tests run during evaluation never reach 4294967295 packets.
The initialization vector (prepended to the ESP payload data) can be chosen at the transmitter's discretion, without coordination with the receiver, although it has to be unique as it must not repeat for a given key. Therefore, the sequence number of the packet is used for the initialization vector. The nonce and additionally authenticated data is calculated as described in Section 2.6.1 of Chapter 2.

The cryptographic operations are then constructed and enqueued onto the DPDK cryptographic device. Since this implementation only uses software cryptographic devices, the operations are performed on the same thread they were enqueued on and are immediately executed when they are enqueued. Therefore, the operations can also be dequeued immediately after they are enqueued. For any operations that failed, the corresponding packet is dropped. Packets with successful cryptographic operations are forwarded to the post-crypto stage. This stage prepends a new Ethernet header, fills it out with the type (IPv4 in this case), source and destination addresses, and calculates the IPv4 checksum for the IP header and fills it in.

### 3.2.2 From ESP

In the case when an ESP packet that matches the Security Association arrives at the application, it should be decrypted, authenticated, and forwarded. This is done in four stages as well, see Figure 3.5.

![Figure 3.5: Processing stages for an incoming ESP packet to be decrypted](image)

Some processing is the same as in the case of creating an ESP packet. The packets arrive as full Ethernet packets to the pre-crypto stage and this stage removes the Ethernet header. This stage also constructs the nonce and additionally authenticated data required for AES-GCM-ESP. The nonce is constructed in the same as as when creating ESP packets, described in Section 2.6.1 One important distinction to make though, is that the initialization vector that is used in the nonce, comes from the received ESP packet. Since this data is authenticated, it is safe to use it in the nonce as well. The additionally authenticated data is supplied with the Security Parameters Index from the Security Association, same as in the construction of ESP packets. To authenticate the sequence number in the ESP header, the sequence number from the packet is inserted into the additionally authenticated data to prove that it has not been tampered with.

The cryptographic operations are then enqueued onto the DPDK cryptographic device. The actual process of enqueuing and dequeueing is the same no matter what operation is actually carried out.

After the cryptographic operations have been completed and dequeued, the packets are sent to the ESP processing stage. This stage performs the following operations, in this order:

1. Verifies that the authentication was successful.
2. Verifies that the padding scheme follows the expected pattern.
3.3 Single-Threaded Implementation

3. Verifies the sequence number with the anti-replay service.
4. Removes the outer IP header, ESP header and ESP trailer.

The cryptographic operations dequeued from the DPDK cryptographic device contains information about the result of the operation. In case that the authentication failed (item 1 in the list above), the packet is dropped. The padding scheme is then verified (item 2) so that it follows the padding scheme mandated by the ESP standard.

The next operation in the ESP processing stage is the anti-replay service (item 3). When decrypting packet streams with the anti-replay service enabled, these sequence numbers must be verified. As described in Section 2.6.1 a sliding window is specified for this purpose in the ESP standard. The ESP standard suggests a sliding window using a bitmask representing the already seen packets, and a max sequence number which keeps track of the highest sequence number seen so far. Another algorithm, presented in RFC 6479 [65], uses a ring-based approach instead of shifting and reduces the amount of times the window has to be adjusted. This algorithm was chosen as it fits better with the multi-threaded implementations, see Section 3.4.3.

The ESP processing stage ends with the outer IP header, ESP header and ESP trailer being removed (item 4). After the ESP processing is complete, the packets are sent to the post-crypto stage, which is the same as when creating ESP packets. It prepends a new Ethernet header and calculates and fills in the IPv4 checksum.

3.3 Single-Threaded Implementation

This section presents the single-threaded implementation, from hereon referred to as “single” or “the single-threaded implementation”. This implementation performs all of the steps previously described sequentially; receiving packets, processing them according to the ESP standard, and then transmitting them. It receives a batch of packets from the network card, processes the packets according to Section 2.6.1, and then transmits them.

An attempt was made to linearize the packet processing—i.e. process each packet through all processing stages before starting work on the next packet—in order to keep the packet in the cache until it was fully processed. This involved calling out to the encryption operations for each packet, something which proved to perform much worse than batching all of the packets and processing each stage independently. The overhead of calling out to the encryption operation for each packet, in conjunction with the instruction cache being kept hot by batching packets, is the hypothesis as to why this, more linear approach, was much slower. Because of the low performance, this solution was not used.

3.4 Multi-Threaded Architecture

Before presenting the multi-threaded implementations themselves, some considerations that were taken into account are presented.

3.4.1 Buffer Configuration

In the paper by De Matteis and Mencagli [6], presented in Section 2.8, four patterns for data stream processing are presented. The problem of parallelizing a single Security Association (SA) is a special case of the data stream processing patterns described in the paper. The number of keys is inherently one since only one data stream, i.e. SA, is considered. The window
size can vary, but since each packet is to be processed independently, it can be as low as one. The sliding factor is inherently the same as the window size since each packet is to be processed independently and should only be processed once. Due to the nature of this problem at hand, independent packets of a single stream, the window farming pattern mentioned is well suited for this use case. If multiple SAs had been of consideration, the key partitioning pattern could have proved useful since each SA can be seen as a key. Since a window size and sliding factor of one might incur an overwhelmingly large amount of overhead in comparison to the actual work performed, the window size can be increased as a way to batch multiple packets together. The processing is still performed on a per packet basis though. This will potentially increase the throughput (by reducing the overhead of reading and writing to memory each time a new batch is dispatched) but also increase latency as multiple packets need to be batched before sending them off to processing and each packet has to wait on the other packets in its batch.

The paper [6] also discusses whether the workers should be agnostic or active in the window management. In an active worker scenario, a single global inbound queue could be employed from which the workers dequeue windows to process when they are finished processing their previous window. In the agnostic case, each worker has a private queue from which they dequeue windows, and a separate thread handles distribution of the windows to each worker. The same can be employed for outbound results; either each worker writes to a global outbound queue or each worker has a private buffer queue from which a separate thread dequeues windows. In the paper by Schneider et al. [55] the issue of private buffer queues is discussed. They claim that private buffer queues reduce the contention of global data structures which reduces overhead from parallelization as discussed previously. Nevertheless, they decided to not use private queues since this introduces load balancing problems between the workers. Also, the granularity at which they parallelized was coarse enough to not incur significant overhead of a global data structure. Vasiliadis, Polychronakis and Ioannidis also discuss this issue in their paper [57]. Their design aims to reduce the resource sharing between cores as much as possible and therefore, they use separate ring buffers for each core to reduce the reading from and writing to global data structures. Because of this uncertainty of what is the better approach, both of these approaches are evaluated.

The four chosen queue configurations are as follows:

- A single, global, emitter queue and a single, global, collect queue (1-1).
- A single, global, emitter queue and multiple collect queues (1-M).
- Multiple emitter queues and a single, global, collect queue (N-1).
- Multiple emitter queues and multiple collect queues (N-M).

In the case of single (global) queues, either for emitting or collecting packets, the queues have to be able to handle multiple producers and multiple consumers. This increases the complexity of the queue implementation. When using one queue per worker, this is no longer the case since there is only one producer and one consumer. A single inbound queue inherently balances the load between the workers. Both the ring buffer library and the Eventdev library in DPDK support single and multiple consumers and producers.

### 3.4.2 Core Configuration

Preliminary tests showed that receiving and transmitting packets are relatively time consuming operations. Placing both of these tasks on a single core proved to be a significant
3.4. Multi-Threaded Architecture

bottleneck. Therefore, a different configuration regarding the receive and transmit cores was considered. This configuration places the workload onto two cores; one for receiving and emitting and one for collecting and transmitting. This resulted in one less worker core but since this was shown to be a bottleneck, it resulted in higher performance.

3.4.3 Shared Data

There are two cases of data that potentially has to be shared between the workers when processing the packets of a single Security Association (SA) on multiple threads. When constructing ESP packets, a unique sequence number has to be assigned to each packet. These sequence numbers should preferably be applied to the packets in the order they were received, so that they can be transmitted in the same order. In the multi-threaded implementations presented in this thesis, this number is assigned on the receive core, which removes the need for shared data and fulfills the order requirements.

The other case is the anti-replay service. This window has to be shared between all workers so that a specific packet is only processed once and any fraudulent replayed packets are discarded. This could be implemented on the transmission core, but it would cause the additional processing of enqueueing the packet on a worker’s outbound queue and dequeueing it on the transmission core for invalid packets. Further, accessing each packet on the transmission core would most likely cause cache misses on that core; something that would not happen on the worker core since it most likely already has the packet in its cache. The anti-replay service must be run after the packet has been decrypted and authentication in order to verify that the sequence number has not been spoofed. Therefore, the anti-replay service is implemented on the worker cores.

The anti-replay service was implemented using the algorithm described in RFC 6479 [65] was used. A multi-threaded, lock-less, implementation could be created by using a 128-bit integer where the high 96 bits are used to represent the packets seen—one bit per packet—and the low 32 bits are used to represent the highest sequence number seen. This implementation would have to utilize the x86_64 CMPXCHG16B atomic instruction, allowing for atomic instructions on 128-bit integers. However, a window this small, only supporting a window size of 96 packets, does not suffice for the intended throughput of this application. As there is no widely available support for atomic variables larger than 128 bits at the time of writing, locks were utilized instead. The window size is much larger—1024 packets by suggestion from Ericsson—and can easily be expanded if required.

3.4.4 Ordering

It is important to preserve the order in which the packets arrive to the application. There are several other protocols in the network stack, such as TCP, that would suffer reduced performance if no effort was made to preserve the order. The parallelized implementations naturally create an issue of packets departing the application in a different order than they arrived in due to the implementations’ parallel nature. As the workers may complete their work in an order not consistent with the order it was given to them, there is a need to sort these packets before they leave the application. To deal with this, the implementations must provide a best-effort reordering of the packets before they are sent out of the system. The solution for this comes in the form of the reorder library of DPDK which was described in Section 2.5.5. This library provides a reorder-buffer as well as several methods that can be used to insert packets into and drain packets from it. The only possible configuration for this buffer is the size which was set to the size of the sequence number window used in this application, 1024. If a packet is more than 1024 packets out of order, the anti-replay service would most likely drop the packet, meaning that it is unlikely that the application
3.5 Ring Buffer-Based Implementation

This section describes the parallel implementation using ring buffers, from hereon referred to as “ring” or “the ring implementation”.

This implementation is based on the ring buffer library in the DPDK framework. The ring buffer library is a lock-free, first-in-first-out queue implementation. It supports both single and multiple consumers and producers. This allows for implementation of all the designs discussed in Section 3.4. Depending on the queue configuration selected, the queues are set up accordingly to not incur additional overhead when not necessary.

The application can utilize all cores available on the system by starting a worker thread on all cores, save for the two cores which are used for receiving and transmitting packets. Each of the cores’ tasks is shown in Table 3.1.

<table>
<thead>
<tr>
<th>Core</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0</td>
<td>Performs the receive tasks. Receives packets, assigns them sequence numbers and enqueues packets to worker inbound queues.</td>
</tr>
<tr>
<td>Core 1</td>
<td>Performs the transmission tasks. Dequeues packets from worker outbound queues, reorders them if necessary, and transmits packets.</td>
</tr>
<tr>
<td>Core 2 – n – 1</td>
<td>Performs the worker tasks. Processes packets according to Section 3.2.</td>
</tr>
</tbody>
</table>

Table 3.1: Core tasks for the ring buffer-based implementation

3.5.1 Receive Core

The receive core receives a batch of packets from a network card in the form of an array of rte_mbuf structs. This batch can contain up to \( M_{\text{receive}} \) packets, but can of course be smaller if the network card has not yet received \( M_{\text{receive}} \) packets. The number of actually received packets is denoted \( N_{\text{receive}} \). The received packets are then split into smaller chunks, one for each core. Each core can accept up to \( M_{\text{core}} \) packets. The actual amount of packets a core receives is denoted \( N_{\text{core}} \). These chunks are fairly distributed across the cores, see Table 3.2 for an example with four worker cores, to balance the load. This is achieved by the algorithm outlined in Listing 3.2.

<table>
<thead>
<tr>
<th>( M_{\text{receive}} )</th>
<th>( M_{\text{core}} )</th>
<th>( N_{\text{receive}} )</th>
<th>( N_{\text{core}} ) for cores 2–4</th>
<th>( N_{\text{core}} ) for core 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>64</td>
<td>183</td>
<td>46</td>
<td>45</td>
</tr>
</tbody>
</table>

Table 3.2: Example packet distribution across 6 cores of a single receive batch

Listing 3.2: Packet distribution algorithm

```c
uint16_t quotient = total_number_of_packets_to_enqueue / number_of_workers;
uint16_t remainder = total_number_of_packets_to_enqueue % number_of_workers;
for (uint8_t worker = 0; worker < number_of_workers; worker++) {
    uint16_t number_of_packets_to_enqueue = quotient + (worker < remainder ? 1 : 0);
    // Enqueue packets
}
```
3.6. Eventdev-Based Implementations

The same effect could be achieved by a more naive approach. If the loop instead is over the packets, one packet can be enqueued each iteration, performing round robin distribution over the workers. This increases the number of enqueue operations and introduces unnecessary overhead, wherefor the more complex solution using the algorithm above is used.

These smaller chunks, consisting of \( N_{\text{core}} \) packets, are then enqueued onto each workers’ inbound queue. If the remaining free space in the inbound ring is not enough to hold all packets, this means that the inbound traffic on the network card—the line rate—is greater than what the application can process and the packets that cannot fit in the inbound ring are dropped. No effort is made to buffer up more packets than the workers’ inbound queues can hold. If the application can not process at the line rate, it is better that fewer, more recent, packets are processed than all packets are processed and delivered late.

3.5.2 Worker Cores

Each of the worker cores runs a loop where it dequeues \( M_{\text{core}} \) packets from the inbound queue. The number of actually dequeued packets, \( N_{\text{core}} \), can be less than the number of packets that is requested if the number of packets received by the receive core was less than \( M_{\text{receive}} \). The packets dequeued are then processed in the same way the packets are processed in the single-threaded application. At the end of the processing, the packets are enqueued onto the worker’s outbound queue. As with any calls to the ring buffer library, all packets are not necessarily enqueued if the queue becomes full. If that is the case, the packets are dropped with the same reasoning as before.

3.5.3 Transmit Core

Packets are dequeued from the outbound queue(s), and put in a temporary buffer. This temporary buffer can hold up to \( M_{\text{receive}} \) packets. The batch sizes are therefore the same on the receive core as on the transmit core. The packets in this buffer are fed into the reorder buffer, using the DPDK reorder library. The transmit core then tries to dequeue a batch of packets from the reorder buffer. Note that these packets are not necessarily the same packets that were enqueued just before. When the batch of packets has been dequeued from the reorder buffer, the packets are transmitted to a network card. If the network card’s transmit buffer is full, resulting from the application processing packets faster than the network card can transmit them at, the packets are immediately dropped using the same reasoning as before.

3.6 Eventdev-Based Implementations

In this section the two implementations based on the Eventdev library are presented: (a) the Eventdev-based implementation and, (b) the Eventdev-based implementation utilizing chained packet buffers; from hereon referred to as “event” or “the event implementation” and “event-chained” or “the event-chained implementation” respectively. These implementations can be configured to use the following queue configurations:

1. 1-1: A single inbound event queue is connected to each of the workers’ event ports. A single outbound event queue is connected to the event port of the transmitter.

2. 1-M: A single inbound event queue is connected to each of the workers’ event ports. One outbound event queue per worker is connected to the event port of the transmitter.

3. N-1: The event port of each worker is connected to its own inbound event queue. A single outbound event queue is connected to the event port of the transmitter.
4. N-M: The event port of each worker is connected to its own inbound event queue. One outbound event queue per worker is connected to the event port of the transmitter.

The core tasks differ somewhat from the ring implementation as one core is a dedicated service core, handling the scheduling of events enqueued into Eventdev, see Table 3.3. The structure of the implementation with the queue configuration N-M can be seen in Figure 3.6.

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Performs the receive tasks. Receives packets, assigns them sequence numbers and enqueues packets to worker inbound queues.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 1</td>
<td>Performs the transmission tasks. Dequeues packets from worker outbound queues, reorders them if necessary, and transmits packets.</td>
</tr>
<tr>
<td>Core 2</td>
<td>Service core. Schedules packets from event queues to event ports and vice versa.</td>
</tr>
<tr>
<td>Core 3 – n – 1</td>
<td>Performs the worker tasks. Processes packets according to Section 3.2.</td>
</tr>
</tbody>
</table>

Table 3.3: Core tasks for the Eventdev-based implementations

![Figure 3.6: Eventdev-based implementation structure](image)

3.6.1 Eventdev Specific Configurations

As detailed in Section 2.5.3, there are multiple configurations that can be made when using Eventdev. The general configurations used in these implementations can be seen in tables 3.4 and 3.5.

The default software scheduler was used for this thesis which meant that some configurations were not supported as the default software scheduler does not support the entire interface. Some parameters are ignored within the scheduler and some were difficult to find out what difference they actually made as the documentation of the scheduler left much to be desired.
To get a better understanding of the scheduler, extensive investigation of the source code had to be done.

The following are the motivations for the values used in this thesis for the general configurations, see Table 3.4. The timeout was set to the default value as there was no actual usage of this variable in the default software scheduler. The number of event queues varies with the queue configuration as, for example, the N-M setup requires \(2 \times \text{number of workers}\) queues while the 1-1 setup only requires two. The same goes for the number of event ports which varied with the number of workers. The additional two ports were used by the receive and transmit cores. The enqueue and dequeue depth values are not used by the default software scheduler and were, therefore, not evaluated. The event device config flag was not used as the only available flag at the time of writing is used to enable a per queue timeout which is not supported by the DPDK default software scheduler.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dequeue_timeout_ns</td>
<td>0 (default)</td>
</tr>
<tr>
<td>nb_event_queues</td>
<td>(\text{nb}<em>{\text{inbound}} + \text{nb}</em>{\text{outbound}})</td>
</tr>
<tr>
<td>nb_event_ports</td>
<td>(\text{number of workers} + 2)</td>
</tr>
<tr>
<td>nb_event_port_dequeue_depth</td>
<td>128</td>
</tr>
<tr>
<td>nb_event_port_enqueue_depth</td>
<td>4096</td>
</tr>
<tr>
<td>event_dev_cfg</td>
<td>0x0</td>
</tr>
</tbody>
</table>

Table 3.4: General configuration of the Eventdev-based implementations

The following are the motivations for the values used in this thesis for the number of events limit in each Eventdev-based implementation, see Table 3.5. The number of events limit was discovered to have a major impact on the latency of the application and was therefore calculated based on the number of workers unless this number exceeded the maximum number of events allowed by the scheduler, 4096. These values were based on initial testing where the throughput was maximized while keeping the latency low. The reason for the lower limit of 128 for the event-chained implementation is the way that the default Eventdev software scheduler works. Each event port has a number of credits to be used when enqueuing new events into the system. If there are not enough credits on a port to enqueue the new events, a \(\text{quanta}\) of 32 events is reserved from the total number of events allowed to be in-flight at a time. Before this is done, however, a check is performed to make sure that the number of in-flight events has not surpassed the limit for number of new events allowed in the system at a time; \(\text{total}_{\text{inflight}} < \text{new}_{\text{event}}_{\text{threshold}}\). Each time a dequeue is performed on a port, the \(\text{inflight}_{\text{credits}}\) of that port is increased by the number of released events iff implicit release is enabled. These credits, as well as the \(\text{total}_{\text{inflight}}\), are then reduced by a \(\text{quanta}\) iff \(\text{inflight}_{\text{credits}} \geq 2 \times \text{quanta}\). This means that when \(2 \times \text{quanta} = 64\) events have been dequeued on a port, the in-flight event counter will be decreased. The problem here occurs when the system is not allowed more than 63 new events in the system at a time, that is when the \(\text{new}_{\text{event}}_{\text{threshold}}\) is set to lower than 64. This means that enqueues will be performed until two quantas have been reserved from the in-flight event counter. This puts the counter at 64. The check \(\text{total}_{\text{inflight}} < \text{new}_{\text{event}}_{\text{threshold}}\) will then fail when attempting to enqueue even though the actual number of events in the system is actually 63. As the maximum possible number of dequeued events at this time is 63, the condition of \(\text{inflight}_{\text{credits}} \geq 2 \times \text{quanta}\) will never be fulfilled and consequently the in-flight event counter will never be decreased. This causes a deadlock between enqueue and dequeue. As the \(\text{new}_{\text{event}}_{\text{threshold}}\) is calculated as \(\frac{\text{nb}_{\text{events}}_{\text{limit}}}{2}\) in these implementations, see Table 3.6, this occurs at \(\text{nb}_{\text{events}}_{\text{limit}} = 127\). Therefore, the minimum value is set to 128.
### 3.6. Eventdev-Based Implementations

#### Implementation

<table>
<thead>
<tr>
<th>Implementation</th>
<th>nb_events_limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>((256 \times \text{number of workers}) &lt; 4096)</td>
</tr>
<tr>
<td>event-chained</td>
<td>(128 &lt; (\text{number of workers} \times 65) &lt; 4096)</td>
</tr>
</tbody>
</table>

Table 3.5: Number of events allowed at a time in the Eventdev-based implementations

The motivations for the parameter values presented in Table 3.5 is as follows: The `new_event_threshold` is set to half of the total number of events allowed in the system. The idea is to prevent the receive core from flooding the scheduler with events, potentially preventing the workers from enqueuing. The values of the enqueue and dequeue depths are set to their maximum values here as they are, as previously mentioned, not actually being taken into consideration by the default software scheduler. Disable implicit release was set to false as it is desired to allow for the scheduler to release an event context when the event is dequeued, allowing for a new event to be enqueued.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_event_threshold</td>
<td>nb_events_limit</td>
</tr>
<tr>
<td>dequeue_depth</td>
<td>128</td>
</tr>
<tr>
<td>enqueue_depth</td>
<td>4096</td>
</tr>
<tr>
<td>disable_implicit_release</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.6: Event port configuration in the Eventdev-based implementations

Lastly, these are the motivations for the configuration values for the event queues presented in Table 3.7. As the application makes use of a general reordering solution and because only a single flow is worked on, the scheduling is set to parallel. This also means that the `nb_atomic_flows` and `nb_atomic_order_sequences` settings are irrelevant. The value of the `event_queue_cfg` setting depends on what queue configuration is used. In the queue configurations where a queue is only connected to a single port, the `RTE_EVENT_QUEUE_CFG_SINGLE_LINK` is used. The `RTE_EVENT_QUEUE_CFG_ALL_TYPES` flag is not available as the software scheduler in DPDK does not support it at the time of writing. It is not needed either, as only the parallel scheduling type is used in these implementations.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb_atomic_flows</td>
<td>1</td>
</tr>
<tr>
<td>nb_atomic_order_sequences</td>
<td>1</td>
</tr>
<tr>
<td>event_queue_cfg</td>
<td>varies with queue config</td>
</tr>
<tr>
<td>schedule_type</td>
<td>RTE_SCHED_TYPE_PARALLEL</td>
</tr>
<tr>
<td>priority</td>
<td>RTE_EVENT_DEV_PRIORITY_NORMAL</td>
</tr>
</tbody>
</table>

Table 3.7: Queue configuration in the Eventdev-based implementations

#### 3.6.2 Receive Core

A batch of packets is received by the receive core from the network card in the form of an array of `rte_mbuf` structs. An `rte_event` struct is then created for each packet in the batch and each event is then given a pointer to the packet as well as an id for the inbound event queue where it is destined. These event queue ids are set in a round-robin fashion to evenly distribute these across the worker cores. Each of these new events are also given the operation type `RTE_EVENT_OP_NEW` to inform the scheduler that these events are new to the system.
This is important as there is a limit on how many new events can exist in the system at a time to avoid back-pressuring. When a batch of events have been gathered, they are enqueued into the Eventdev environment by using the `rte_event_enqueue_burst` function call. This function takes a event device id, an event port id, an array of events to be enqueued and the number of events to enqueue as parameters. These events are now handed over to the scheduling core which places them in the event queues based on the queue id they were given earlier.

### 3.6.3 Worker Cores

The worker cores poll the inbound queue connected to its event port for events using the `rte_event_dequeue_burst` function call. This function takes a device id, a port id, an array to write the events to, the number of events to dequeue and the timeout duration as parameters. The worker then loops through these events and processes the referenced packets. When the processing is complete the worker modifies the events by setting the queue id to the outbound event queue where they are destined and setting the operation type to `RTE_EVENT_OP_FORWARD`. This operation type informs the scheduler that these events are not new to the system but are simply being forwarded between queues. It can then enqueue the events into the Eventdev environment using the aforementioned `rte_event_enqueue_burst` function call.

### 3.6.4 Transmit Core

The transmitting core polls the outbound queue(s) for events. When it manages to dequeue a set of events, it collects the packets referenced by the events and enqueues them into the reorder buffer. It then attempts to drain packets from the reorder buffer and, if any packets were drained, transmits them.

### 3.6.5 Eventdev Implementation using Chained Packet Buffers

An alternative way of handling the packets in Eventdev was created as the documented usage requires an `rte_event` struct to be copied multiple times, once for each enqueue and dequeue, for each packet. This implementation makes use of the `userdata` field of the `rte_mbuf` struct to chain together multiple packets in a singly linked list by writing the address of the next packet into that field. This allows for an arbitrary amount of packets to be enqueued with a single event as an event can reference the entire list by pointing to the first packet in that list. The flow of a batch of packets in this implementation is fairly similar to the original implementation with the difference that only one event is created for each batch of packets. The distribution of packets across the worker cores is done in the same fashion as in the ring implementation, see Listing 3.2, as this way of batching packets with events allows for it. The linked list is null-terminated by explicitly writing `NULL` to the `userdata` field of the last `rte_mbuf` struct in the chain. The worker threads process the packets by beginning with the `rte_mbuf` struct directly linked to the received event and then traversing the linked list. A downside to this solution is the fact that prefetching can only be done on the next `rte_mbuf` struct in the list. This is because further traversing of the list would require dereferencing several `rte_mbuf` structs, defeating the point of prefetching. As mentioned in Section 2.5.1 the layout of the memory related to each packet causes the spatial locality to diminish if the packet data is not processed directly after dereferencing its `rte_mbuf` struct. This is also a factor when creating the packet chains as each `rte_mbuf` struct in the chain must be dereferenced to be linked. Another impact of this technique is the one on load balancing. Since all of the packets in one batch are now inseparable, they can only be distributed as a unit to a single worker. This means the load balancing becomes much more coarse.
In this chapter, the methodology used to evaluate the application as well as the environment and metrics used in the testing are presented. The evaluation was split into several different types of tests. Firstly, multiple different configuration parameters were evaluated to find the best configurations of each implementation. One configuration of each implementation was then chosen for in-depth testing.

This chapter begins with a description of the setup, packet generation, packet streams, as well as metrics used and how they were measured in all of the tests. Following that are descriptions of how the application was verified to be correct, as well as how the application was evaluated.

4.1 Hardware, Software and Compiler

The majority of the evaluations were run on a dual socket system with two Intel Xeon E5-2620 v3 CPUs running at 2.40 GHz with 256 GB RAM, the six core system. This gives a total of 12 physical cores and 24 hardware threads with 6 cores on each NUMA socket, see Figure 4.1. The application was only evaluated running on a single processor. The reason for this is that all packets are located in the memory of a single socket. Copying this data over to the second socket, or letting the second socket access the memory of the first socket, has to happen over the Intel QuickPath Interconnect on this machine. Preliminary tests showed that there was very little to gain from using cores on the second socket and in most cases performance decreased. Therefore, evaluating the application with more than one socket was not so interesting to investigate for performance improvements.

The evaluations were performed on one processor isolated from the Linux scheduler (using isolcpus) so that it does not schedule any processes on the processor the evaluation is running on. In addition, Intel Hyper-Threading was disabled to avoid any performance implications that simultaneous multi-threading might introduce. Although Intel Hyper-Threading is disabled for the majority of the tests, the impact of Intel Hyper-Threading is evaluated and discussed in an in-depth test. Intel Turbo Boost was also disabled to keep the processors at a constant clock speed in order to make the test results more stable and reliable. Each
of the threads was pinned to its own specific core to avoid migration of threads between cores. This increases performance stability and improves the temporal locality of the system [66]. The system ran Ubuntu 16.04.4 LTS (Xenial Xerus) and compilation was done using gcc version 5.4.0 20160609 with -O3 optimizations.

Socket 0:
+-------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| | 0 12 | | 2 14 | | 4 16 | | 6 18 | | 8 20 | | 10 22 |   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| | 32 kB | | 32 kB | | 32 kB | | 32 kB | | 32 kB | | 32 kB |   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| | 256 kB | | 256 kB | | 256 kB | | 256 kB | | 256 kB | | 256 kB |   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| +---------------------------------------------------------------+   |
| | 15 MB |   |
| +---------------------------------------------------------------+   |
+-------------------------------------------------------------------+

Socket 1:
+-------------------------------------------------------------------+
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| | 1 13 | | 3 15 | | 5 17 | | 7 19 | | 9 21 | | 11 23 |   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| | 32 kB | | 32 kB | | 32 kB | | 32 kB | | 32 kB | | 32 kB |   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| | 256 kB | | 256 kB | | 256 kB | | 256 kB | | 256 kB | | 256 kB |   |
| +--------+ +--------+ +--------+ +--------+ +--------+ +--------+   |
| +---------------------------------------------------------------+   |
| | 15 MB |   |
| +---------------------------------------------------------------+   |
+-------------------------------------------------------------------+

Figure 4.1: Processor and cache topology generated using likwid-topology -g. The top boxes in each socket shows the thread identifiers for each core. The size of the L1 cache follows below, then the size of the L2 cache. The L3 cache is shared across the whole socket and can be seen in the bottom box in each socket.

Figure 4.1 shows the cache topology of the processors. Each core, which has two hardware threads (with Intel Hyper-Threading enabled), has a 32 kB L1 cache and a 256 kB L2 cache. All cores on each processor then share a 15 MB L3 cache.

DPDK requires packets to be allocated from hugepages, and recommends the usage of 1 GB hugepages if supported. A total of 40 hugepages, a total of 20 hugepages, 1 GB each, were allocated for the processor where the application was run.

In order to evaluate the application with more than 6 cores, an additional system with 14 cores was used, the 14 core system. This system was acquired later on during the project and was not available early enough to perform all tests on. This system has two Intel Xeon E5-2680 v4 CPUs, each with 14 cores and 28 threads, running at 2.40 GHz with 128 GB RAM. This increases the maximum number of workers from 3 to 11 for the Eventdev-based implementations and from 4 to 12 for the ring implementation. This system was not fully under the
4.2 Packet Generation

The six core system used to evaluate the application was equipped with two network cards, each with two 10 Gbps ports. Due to time constraints and the relatively low speed of the network cards, the packets were instead read from a pcap trace file using the pcap driver supplied by DPDK. This driver allows for the application to interact with it in the way it would with a physical network card, enqueuing and dequeuing batches of packets. During some initial testing, however, it was discovered that actually enqueuing and dequeuing from this driver was quite costly, as it required writes and reads to disk. To avoid this cost and to focus the testing on the application and packet processing, some counter measures were taken. To remove the cost of dequeuing from the driver, the application receives a batch of 1000 packets at initialization and then produces new batches by allocating new memory and copying the packet data from the original packets into this new memory area. This avoids a costly disk read every time the application needs new packets. To remove the cost of enqueuing packets to the driver, the memory related to each packet is freed instead of actually transmitting the packet data. By doing this the limitation of the application was no longer the speed of the pcap driver.

One important note to make here is that packets failing the anti-replay check are not actually dropped by the application, as would happen in a real world scenario. This is due to the fact that it is not possible to generate packets with unique sequence numbers when generating packets in the fashion described above. Since the sequence number used in the encryption and authentication of the packets, changing the sequence number on the packet when they are generated would make the authentication of the packets fail, causing them to be thrown away. This would cause all packets but the first 1000 packets to be dropped since the following packets are just copies of the first batch, and therefore the sequence numbers repeat. In order to make sure that the sequence number verification checks were not removed by the compiler for optimization reasons, these packets are instead marked with a special identifier in the IPv4 header and passed on to transmission.

4.3 Packet Streams

In order to evaluate how the application handles different types of traffic, tests were performed with different streams of packets. The primary packet stream that is used to evaluate the application is an Internet Mix—or IMIX—of packets 40, 660 and 1500 bytes in length. The packet sizes as well as the distribution of this mix was based on the observed distributions in a technical report from 2007 by Sinha, Papadopoulos, and Heidemann [67] and can be seen in Table 4.1 in the “% of IMIX Stream column”. The distribution in the report is continuous but due to the constraints of using an artificial packet stream, a small set of discrete values was chosen. The IMIX stream was created by generating a packet at a time using the specified packet distribution. This stream, consisting of 1000 packets, was then loaded into the application and repeated until all measurements had been made.

[2] See footnote 1
In addition to the IMIX stream, streams containing just 40, 660 or 1500 byte sizes packets were also used to see how the application performs with small, medium and large packets. The packet sizes, along with their final packet size after encryption, and the ratio between payload and overhead are shown in Table 4.1. The percentage of the ESP packet that is overhead is more than 50% with small packets, but quickly dwindles down into single digit percentages when the packet sizes increase. These sizes are the sizes specified in the Total Length field of the IPv4 header, some additional overhead is then added from the Ethernet header. Further, when evaluating the decryption performance, more overhead is added from the outer IPv4 and ESP headers on the inbound packets.

<table>
<thead>
<tr>
<th>Original Packet Size</th>
<th>ESP Packet Size</th>
<th>% Overhead</th>
<th>% of IMIX Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>40 bytes</td>
<td>86 bytes</td>
<td>53%</td>
<td>40%</td>
</tr>
<tr>
<td>660 bytes</td>
<td>706 bytes</td>
<td>6.5%</td>
<td>40%</td>
</tr>
<tr>
<td>1500 bytes</td>
<td>1546 bytes</td>
<td>3.0%</td>
<td>20%</td>
</tr>
</tbody>
</table>

Table 4.1: Packet sizes, their overhead, and distribution in the IMIX stream used during evaluation

4.4 Metrics

The metrics used in the evaluations are the following:

- Throughput, in terms of packets per second (in million packets per second, Mpps)
- Throughput, in terms of data rate per second (in megabits per second, Mbps)
- Latency (in milliseconds)
- Speedup (compared to the single-threaded implementation)
- Cache miss rate (in last level cache miss rate per test run)

Throughput is a commonly used metric in this field and is commonly defined as data processed over a certain time interval \[48, 49, 53\]. Throughput is, in this thesis, calculated by dividing the total amount of data, or the number of packets, processed over a period of time by the time passed. The speedup is also calculated for throughput, in Mpps, by dividing the throughput of the parallelized implementations with the one of the single-threaded implementation. The overhead introduced when parallelizing the processing of packets is bound to increase the time each individual packet spends in the application since they have to be distributed between cores. To monitor this difference, latency was also measured. Latency is, in this thesis, defined as the time it takes from when a packet arrives at the application until it leaves. Finally, the cache miss rate has been shown in a paper by Zhuralev, Blagodurov and Fedorova \[21\] to be a major indicator of how well an application is suited for running on multiple threads on a single multi-core CPU, and is as such interesting to look at. For brevity this was only evaluated for the highest number of workers for each implementation.

4.5 Measurements

The metrics that are measured inside the application itself are throughput and latency. The cache measurements were done externally using the program perf stat\(^3\) which uses hardware counters to gather data about the cache performance of a process. The cache misses were

\(^3\)\text{http://man7.org/linux/man-pages/man1/perf-stat.1.html}
only measured for the maximum number of workers for each implementation on the 6 core system. This was due to time limitations and a focus on finding out about what the limitations of the application were at higher levels of parallelism. The tests were run multiple times to ensure that the results were consistent and that no anomalies went unnoticed. However, the results presented are from a single test run.

When running the application in encryption mode, i.e. it receives IP packets and transmits IP / ESP packets, the throughput was measured when the packets arrived at the application. This is important in order to give a fair comparison since the packet size-overhead of ESP, compared to the actual payload, varies greatly depending on packet size. This is a representative way of measuring the throughput of the application as the measured rate is the one at which the workers can accept packets. For decryption, the throughput is measured when the decrypted packets are transmitted from the application. This makes the throughput comparable to the throughput measured when encrypting packets, since both measurements are made of non-ESP packets.

The throughput measurements were performed by using counters for the number of packets processed as well as the total amount of data processed. The latter was calculated by summing up the packet length of every packet. Every 1 million packets—roughly, since some configurations process packets in chunks making hitting exactly 1 million packets impossible—a snapshot of this information, together with the absolute time and relative time since the last measurement, was taken. Each latency measurement was then calculated based on those entry and exit times. Each test ran until 40 throughput measurements, for a total of roughly 40 000 000 packets, and 2 000 latency measurements had been made. The first 50 latency measurements were then trimmed from the results as a warm-up anomaly consistently appeared in those measurements. Timestamps were taken using clock_gettime and the CLOCK_MONOTONIC clock.

4.6 Verification of the Encapsulating Security Payload Protocol Implementation

As the correctness of the application is key for the performance measurements to have any significance, tests were performed to ensure that the application conforms to the specified subset of the ESP standard, see Section 1.2. The tests were performed on the multi-threaded implementations in addition to the single-threaded one to make sure that the parallelism in the application did not damage the integrity of the application in any way. Two different tests were performed for this purpose. The first to prove the correctness of the packets that are encrypted and decrypted and the second to prove that the anti-replay service is adhering to the ESP standard.

4.6.1 Packet Construction and Deconstruction Testing

For the application to fulfill its purpose, the packets that are encrypted need to be able to be decrypted by an arbitrary implementation of the ESP standard (assuming it adheres to the same subset that is presented in this thesis). To confirm this, the implementation cannot be tested using the application itself. Packets produced by it may not adhere to the standard, but could still be decryptable by the application itself. Therefore, Wireshark, a program to capture and analyze network packets, was chosen to verify the correctness of the packets.

The Security Association established in the application was entered into Wireshark and verification of ESP packets was enabled, see Figure 4.2. Since AES-GCM is an algorithm for [https://www.wireshark.org/](https://www.wireshark.org/)
4.6. Verification of the Encapsulating Security Payload Protocol Implementation

authenticated encryption with associated data (AEAD), no authentication key has to be supplied for the Security Association as this is implicit in the encryption key. Pcap trace files of packets encrypted using the application were opened in Wireshark to validate that they could be decrypted and authenticated according to the Security Association.

Figure 4.2: Wireshark ESP SAs preferences

An application on the receiving end of ESP packets, and that wishes to decrypt them, should implement verification of the padding contents of the decrypted packets according to the ESP standard. AES-GCM-ESP does not specify a specific padding scheme so the default padding scheme described in the ESP standard was used. This scheme states that the padding should be filled with the following sequence: 1, 2, 3, .... This padding cannot be verified in Wireshark since it strips out everything but the original packet when displaying the decrypted data. This was instead verified by printing the decrypted packets’ contents in the application and verifying the padding scheme manually. The packet contents were printed after the Ethernet header had been removed from the packet and after it had been decrypted, but before the outer IP and ESP headers had been removed.

4.6.2 Anti-Replay Service Testing

ESP specifies an anti-replay service to provide protection from replay attacks, as described in Section 2.6.1. Because the application presented in this thesis parallelizes a single Security Association, this service has to be shared across multiple threads. To prevent a packet from being replayed, i.e. a packet with the same sequence number as another packet, the ESP anti-replay service uses a sequence number window of a fixed size that keeps track of which sequence numbers have been seen. Already seen packets and any packet with a sequence number that is smaller (older) than the smallest sequence number in the window are discarded.

To verify the implementation of this anti-replay service, four different encrypted packet sequences were sent through the application and the output was then verified according to the ESP standard. Since the sequence number information is not carried across to the decrypted packets, this was verified by printing the accepted sequence numbers after the Ethernet header had been removed from the packet and after it had been decrypted, but before the outer IP and ESP headers had been removed. The pcap trace files containing these streams were generated using the application with the single-threaded implementation modified to generate the packet sequences presented below.

1. Correct sequence numbers
   
   Input: 1 2 3 4 5 6 7 8
   
   Expected output: 1 2 3 4 5 6 7 8
2. Correct sequence numbers in incorrect order
   Input: 4 6 8 1 2 3 5 7
   Expected output: 4 6 8 1 2 3 5 7

3. Repeated sequence numbers
   Input: 8 9 10 11 12 8 13
   Expected output: 8 9 10 11 12 13 (packets with repeated sequence numbers were discarded)

4. Sequence numbers outside the window
   Input: 1200 1201 5 1202
   Expected outputs: 5 1200 1201 1202 or 1200 1201 1202 (packet with sequence number 5 was discarded)

Due to the multi-threaded nature of the application, the order the packets are processed in determines which packets are discarded. The fourth packet sequence—1200 1201 5 1202—actually has two different valid outputs. If the first packet verified against the sequence number window is 5, it will be accepted, and when any of the other packets are processed, they are accepted too. If the first packet instead is any of 1200, 1201 or 1202, the packet with sequence number 5 will be discarded since the sequence number window has already been moved. Therefore, both of these outputs are accepted and the test is run multiple times to verify that both outcomes are possible.

4.7 Configuration Parameters

The application has a few parameters that can be configured. They are the how many packets each worker processes at a time, the number of workers as well as how the queues are configured. The parameters, and the different values used, are described in more detail below. The application is tested with different combinations of these parameter values to find which values performs best.

4.7.1 Burst Size

In the application, the burst size is represented as the number of packets processed per worker at a time. For the single-threaded implementation, this is simply how many packets are processed between each receive and transmit batch. In the multi-threaded implementations, the burst size represents the window size which is the number of packets enqueued onto a worker’s work queue as well as how many packets a worker processes at a time. The different burst sizes that were tested are: 1, 2, 4, 8, 16, 32, and 64. Preliminary tests showed that smaller burst sizes performed better in general, which is why burst sizes as powers of two was chosen as the value density is higher at lower numbers. Powers of two also conveniently reduced the number of values that had to be tested for this parameter. A burst size of 128 showed a dramatic decline in performance during preliminary test which is why 64 was chosen as the upper bound.

4.7.2 Number of Workers

The number of workers was chosen as a configuration parameter as it is interesting to see how the performance of each configuration scales as the number of workers increases. It is also interesting to see what other configuration parameters matter the most at different levels of parallelism. Due to constraints of the six core system and DPDK, the number of workers tested for the EventDev implementations were 1, 2, and 3 and the number of workers for the ring buffer-based implementations were 1, 2, 3, and 4. The 14 core system that was
acquired late in the project allowed for testing of some configurations with up to 11 workers for Eventdev-based implementations and 12 for the ring buffer-based one.

4.7.3 Queue Configuration

Four different queue configurations were tested; 1-1, 1-M, N-1, and N-M. The 1-1 configuration consists of two queues, one which all workers dequeue packets from and one which all workers enqueue packets onto. The N-M configuration consists of two queues per worker, allowing them to dequeue and enqueue from and onto individual queues. The 1-M and N-1 configurations are combinations of the 1-1 and N-M configurations. These different configurations are interesting to test as their impact on important factors such as load balancing, reordering and cache synchronization differs.

4.8 Configuration Evaluation

As the number of possible combinations of the configuration parameters are so great, the number of configurations to evaluate in depth must be confined to a smaller number. In order to find the best configuration for each implementation, all configuration combinations were tested and the best configurations for each implementation is then tested in the further evaluation. These tests were done using the IMIX packet stream as this is the most realistic packet stream.

4.9 Further Evaluation

To be able to further analyze the implementations, additional tests were performed on the subset of configurations chosen based on the configuration evaluation. These configurations were also tested using streams with different packet sizes: 40 bytes, 660 bytes, and 1500 bytes. Following is a presentation of different in-depth areas that were chosen to further evaluate the implementations and how the evaluations of these were performed.

4.9.1 Bottleneck Evaluation

In these tests, different parts of the application were modified to remove some bottlenecks and to discover other ones. The modifications were chosen as they reduce the work done on certain cores to be able to see how the work of those cores limits the performance. The two modifications that were made are listed below:

- Removing the copying of packet data when generating packets was done to reduce the work done by the receiving core. It also reduces the number of memory accesses performed by the receive core. The application processes packets filled with whatever data that is currently in the newly allocated memory area instead of copying real packet data into them.

- Disabling encryption and decryption was done so that the workload of the workers was essentially removed. This is useful to see just how fast the scheduler is able to distribute the incoming packets. This may also reduce the number of memory accesses done by the worker cores if any packets have been evicted from the cache.

4.9.2 Intel Hyper-Threading Evaluation

The Hyper-Threading functionality, a variant of simultaneous multi-threading, of Intel processors allows for several, usually two, threads to be processed concurrently on a single processor core. This can, at best, provide the performance of two physical processor cores. The
possible benefit depends a lot on the type of computation, however, as the possible level of performance achievable is largely based on to what degree the Hyper-Threads accesses the same resources of the processor at the exact same time. If there is a lot of contention on the same CPU resource by the Hyper-Threads, the performance can even decrease. This has been explored in detail by Ruan et al. [59].

To evaluate what performance could be achieved with the implementations presented in this thesis when using Hyper-Threading, the tests were performed where the Hyper-Threads of the cores were utilized as well. To perform a fair comparison the application ran on the same number of threads during the tests where Hyper-Threading was enabled as during tests where Hyper-Threading was disabled. The difference being that two Hyper-Threads were running on one physical processor core instead of separate physical processor cores. The difference in performance was then compared to see how close the performance when utilizing Hyper-Threading could get to the performance of using a dedicated physical core for each thread.

For the ring implementation, see Figure 4.3, the physical cores that previously handled receiving and transmitting packets now share their cores with a worker. The last core then ran two workers, resulting in a total of 4 workers.

For the event implementation, the physical cores that previously handled receiving and transmitting packets as well as the service core now share their respective physical cores with a worker, see Figure 4.4.

4.9.3 Evaluation of the Impact of Reordering Packets

One cost of parallelization in an application, such as the one presented in this thesis, is the cost of reordering the packets that may have become rearranged as they were distributed to and collected from the multiple worker threads. This cost can depend on the configuration used as some parameters, especially burst size and queue configuration, may increase the likelihood of packet batches ending up out of order. To evaluate this cost, tests were performed for the different configurations with the reordering functionality turned on as well as off.
5 Results and Analysis

In this chapter, the results of the evaluations described in Chapter 4 are presented as well as analyzed. The results of the verification of the ESP protocol implementation is presented first. A short section then follows on latency measurement issues. The chapter then presents and discusses the performance of each of the four implementations: the single-threaded implementation, the ring implementation, the event implementation and finally the event-chained implementation. Following these results are a section on which configurations were chosen for the extended testing as well on some comparison between the implementations. The results from the bottleneck, Hyper-Threading as well as the reordering tests are then presented and discussed. At the end of the chapter, a small conclusion from the results is presented and the methodology used to perform these tests and evaluations is discussed.

5.1 Verification of the Encapsulating Security Payload Protocol Implementation

Below, the results from the verification of the ESP protocol implementation are presented. All implementations—sharing much of the code for the actual construction and destruction of ESP packets as well as encryption—passed the tests of verifying and decrypting the packets according to the ESP standard, see Section 4.6 for more detail on the tests.

5.1.1 Packet Construction and Deconstruction Testing

Packets produced by the application were shown to be decryptable using Wireshark. All different implementations, i.e. the single, ring, event and event-chained implementations, in combination with all packet sizes were verified. Wireshark is able to identify the packets as ESP packets and part of the correct Security Association. With this information, it is able to decrypt the packets and show their original contents. In Figure 5.1, an IPv4 packet from 100.100.100.100 to 200.200.200.200 containing a UDP packet of length 12 filled with 0x58, or 'X', is shown. The padding scheme was verified to be correct for all of the different implementations and packet sizes as well. Following the payload data, see the 12 bytes of 0x58 in the bottom window of Figure 5.1 is the correct bytes of padding (0x01 and 0x02) which is then followed by the correct padding length (0x02).
5.1. Verification of the Encapsulating Security Payload Protocol Implementation

5.1.2 Anti-Replay Service Testing

The results from the anti-replay service testing is presented in Table 5.1. These tests were done with a burst size of one in order to increase the randomness of the order the packets were processed in. Except for the single-threaded implementation, the results are not exactly the expected output. The sequence numbers of the packets that were output is correct, no duplicate or out-of-window packets are output, but the order that they are output in is scrambled, which is unexpected. This is because of two reasons:

- The print happens before the packets are reordered on the transmission core.
- The prints are issued from the worker cores and their order can not be guaranteed.

Test number four was run multiple times for the multi-threaded implementation to ensure that both of the possible outcomes discussed in Section 4.6 occurred, and the first occurrences of both outcomes are recorded here.
5.2 Latency Measurement Issues

An important thing to note regarding all latency measurements of the parallelized implementations is that, in several cases, the input throughput is constantly higher than what the application can process. This is due to the fact that the receiving core is generating packets as fast as it can, with no regards to how much data the worker cores and transmit core can actually process. This means, in the cases where the receive core generates higher throughput than the application can process, that the queues of the application will be constantly full and thereby cause the latency measurements to consistently exhibit the worst case scenario latency. In a real world scenario, the input throughput would vary over time, which would allow the application to keep the queues empty more often and thereby reduce the average latency. The worst case latency is directly proportional to the queue sizes or, in the case of the Eventdev-based implementations, the number of events allowed in the application at a time. This, together with the fact that no requirements have been set for how large the queues should be, makes all of the latency measurements taken where the queues fill up difficult to analyze.

5.3 Single-Threaded Implementation Evaluation

The results from the evaluation of the single-threaded implementation on the six core system are presented and discussed in this section. The single-threaded implementation works as a point of comparison for the multi-threaded implementations. The presented results were generated using the IMIX packet stream.

5.3.1 Throughput

The throughput of the single-threaded implementation is shown in Figure 5.2. This figure—and the figures in the rest of this chapter—is split into two plots: one for encryption and one for decryption. The throughput, in million packets per second (Mpps) is on the y-axis, and the burst size is on the x-axis. For each value of the burst size, there is one bar for each
queue configuration. Since there is no queue configuration to be made in the single-threaded implementation, only one bar is visible for each burst size in this section.

For the single-threaded implementation, the throughput is relatively even across all burst sizes, peaking at a burst size of 8 packets, both for encryption and decryption, see Figure 5.2. It reaches a maximum of 1.59 Mpps when encrypting and 1.53 Mpps when decrypting. With a larger burst size, the intuition would be that the instruction cache could be kept from swapping out data since more time is spent processing packets before the core has to transmit and receive new packets. The actual amount of code run for each packet is relatively modest though and it is unclear if this actually has any impact on the result. A potential reason for the peak at a burst size of 8 is that all the packets can be kept in the cache at the same time with a burst size of eight. In DPDK, each packet is allocated 2000 bytes of memory, no matter the actual size of the packet. With a burst size of eight, the total size of these packets is $8 \times 2000 = 16000$ bytes. With a burst size of 16, the total size is $16 \times 2000 = 32000$. The L1 cache size for each core on the six core system is 32 kB which could explain why a burst size of eight performs the best. Below eight, the cache is underutilized and above eight, the cache is overfilled and has to swap, but this obvious looking at the cache miss rate data, see section 5.3.3.

![Throughput of the single-threaded implementation using the IMIX packet stream](image)

5.3.2 Latency

Latency increases linearly with the packet burst size, doubling for each doubling of the burst size, see Figure 5.3. The latency is very low across all burst sizes, reaching a maximum of 35.3 $\mu$s for encryption and 36.7 $\mu$s for decryption with a burst size of 64. With a burst size of 8 the average latency was 4.6 $\mu$s for encryption and 4.8 $\mu$s for decryption. The linear relationship makes sense as each packet of each burst must wait for the $\text{burst}_\text{size} - 1$ other packets in the burst to finish being processed before it can be released from the application.
5.4. Ring Buffer-Based Implementation Evaluation

5.3.3 Cache Misses

The percentage of loads that were last level cache (LLC) misses of the single-threaded implementation can be seen in Figure 5.3. It appears to not be significantly affected by the burst size and is fairly stable around, or just below, 0.1%. The LLC misses does, however, decrease slightly as the burst size increases. This is reasonable as with more packets being processed at a time, more packets can be kept in the cache at the same time. The number of LLC misses seems to have a weak relation to the throughput or the latency though. The number of loads that are LLC misses are very small and cannot be expected to impact performance very much.

5.4 Ring Buffer-Based Implementation Evaluation

This section contains the results from the evaluation of the ring implementation on the six core system as well as some observations and discussion on how the performance correlates to the configuration parameters. The ring implementation can utilize all four cores not used for receiving and transmitting as workers. To see how the implementation performs with different numbers of workers—one, two, three, and four—all have their own graphs. Throughput, in Mpps, and burst size is plotted in the same fashion as with the single-threaded implementation. The ring implementation can use all of the queue configurations presented...
5.4. Ring Buffer-Based Implementation Evaluation

previously; each burst size therefore has four bars, one for each queue configuration. The presented results were generated using the IMIX packet stream.

5.4.1 Throughput

In Figures 5.5, 5.6, 5.7, and 5.8 the throughput is plotted against the burst size. There is a steady increase in throughput as the burst size increases with one and two workers. With three or more workers and large burst sizes the throughput declines. For a single worker, a burst size of 64 performs best in terms of throughput. This is expected since the amount of time spent enqueuing, dequeuing and performing per-burst work is reduced in comparison to per-packet work when the burst size increases.

![Figure 5.5: Throughput of the ring implementation using 1 worker core using the IMIX packet stream](image)

![Figure 5.6: Throughput of the ring implementation using 2 worker cores using the IMIX packet stream](image)

As noted, large burst sizes perform worse as the number of workers increases. For three and four workers the best performing burst sizes are eight and four respectively. A plausible explanation is that the number of cache misses increases as the burst size increases (see Figure 5.13), which in turn could lead to a decrease in throughput. Due to the way the application is structured; first the “pre-crypto” operations are performed on all packets, then all packets are encrypted (or decrypted), and then the “post-crypto” operations are performed on all packets, each of these stages might incur cache misses for many packets which could...
explain the decrease in throughput with larger burst sizes. Another possible explanation is that multiple workers try to enqueue and dequeue large amounts of data on the same queues, which causes memory contention and thereby reduces throughput. This does not seem to be the primary cause though since all queue configurations exhibit this behavior. With the N-M configuration, each worker has its own inbound and outbound queues to work on, removing any contention between workers. The contention between the workers and the receive and transmit cores still exists though.

What can also be observed is that the queue configuration does not impact the performance in a major way. For very small burst sizes, the 1-M and N-M queue configuration perform better, but that difference becomes negligible when the burst size grows past two packets. The top performing configuration for the ring implementation, in terms of throughput, is with four workers and a burst size of four, reaching 4.26 Mpps during encryption and 4.10 Mpps during decryption.

Figure 5.7: Throughput of the ring implementation using 3 worker cores using the IMIX packet stream

Figure 5.8: Throughput of the ring implementation using 4 worker cores using the IMIX packet stream

5.4.2 Latency

Figures 5.5, 5.6, 5.7, and 5.8 show the latency of the ring implementation. The latency results show a more sporadic behavior compared to the throughput graphs. For a single worker, see
Figure 5.9: Latency of the ring implementation using 1 worker core using the IMIX packet stream

With two workers, see Figure 5.10, the N-1 and N-M queue configurations show almost identical latency as with one worker. With N-1 and N-M there are two inbound queues meaning there is twice the space for packets to linger. Having two workers means the queues are emptied faster as well, but it evens out with the extra queue space. The 1-1 and 1-M queue configurations has their latency almost halved compared to with one worker. This is probably due to the fact that there is still only one inbound queue but two workers processing the packets in the queue, meaning packets cannot linger in them for as long.
For three workers, see Figure 5.11, the results are not as consistent. For the 1-1 and 1-M queue configurations, the latency was again halved compared to with two workers. For decryption, the N-1 and 1-M queue configurations show the same behaviour as with two workers, at least for burst sizes below 32. For burst sizes of 32 and 64, all queue configurations perform fairly similarly. For encryption, all queue configuration perform similarly, except for N-1 and N-M combined with burst sizes of one and two packets. The reason for these inconsistencies is most likely that in some configurations the queues become full, exhibiting the worst case latency, and with the other configurations the queues are empty, showed by the lower latency.

With four workers, see Figure 5.12, the graph looks quite different from one, two, and three workers. The receive core has become the bottleneck of the application for almost all configurations, leaving the inbound queues empty at all times. The application has now reached a point where the latency is limited by the time it takes for a packet to actually flow through the application instead of the time a packet spends in the inbound and outbound queues. The latency exhibited by the ring implementation is now very similar to the latency of the single-threaded implementation, both in terms of magnitude and in terms of how it relates to the burst size. The only overhead now comes from the fact that the packets have to be distributed to the different worker cores and then collected on the transmitter core. Some
configurations are still limited by the worker cores though, causing the queues to fill up, and are therefore displaying a high latency. For four workers the minimum latency is achieved with one packet per burst at about 2.3 µs for both encryption and decryption. There are some spikes with some queue configurations and burst sizes, similarly to with three workers.

Figure 5.12: Latency of the ring implementation using 4 worker cores using the IMIX packet stream

The inconsistent behaviour and spikes with three and four workers were shown to be consistent over multiple runs of the application, implying that there is a consistent reason for this behaviour besides external factors. Most likely, this is due to queues either becoming full and staying full, or becoming empty and staying empty. As noted, a full queue will incur massive latency penalties, but as soon as queues are empty, the latencies are almost as small as with the single-threaded implementation. Overall, the latency steadily decreases as the number of workers increases, reaching its lowest point when using four workers. As presented previously, there was close to no throughput increase when going from three to four workers. However, the latency continues to decrease, showing that there is a benefit to using more workers even if the throughput does not necessarily scale well.

5.4.3 Cache Misses

The share of loads that were last level cache (LLC) misses is presented in Figure 5.13. The rate of LLC misses increases quite steadily as the burst size increases. At a burst size of 64, the 1-1 and 1-M queue configurations exhibit a lower percentage of cache misses than the N-1 and N-M counterparts. Overall, the percentage of loads that are LLC misses are in the range of 4 - 10%.
5.5 Eventdev-Based Implementation Evaluation

The following are the results of the configuration evaluation of the event implementation on the six core system. The throughput, latency, and cache measurements are presented here as well as some observations and discussion on how the configuration parameters affect them. The Eventdev-based implementation can utilize three workers on this system, each number of workers is presented with their own graph. The presented results were generated using the IMIX packet stream.

5.5.1 Throughput

The throughput when varying the queue configuration and burst size with one worker can be seen in Figure 5.14, two workers in Figure 5.15 and three workers in Figure 5.16. As the number of workers increases, the less performance is gained for using larger burst sizes. The main reason for this is due to how the default Eventdev software scheduler is implemented. Eventdev has a limit as to how many events can be enqueued at a time. This limit is set to 64 and means that if three workers and a burst size of 32 is used, only 64 out of those $32 \times 3 = 96$ events are actually enqueued. As the events are given their queue ids in a round-robin fashion, this does not cause issues in terms of load balancing but it does limit the viable burst size to $\frac{64}{number\_of\_workers}$. The difference between two and three workers is quite dramatic, see Figures 5.15 and 5.16 as the throughput when using a burst size of 64 is lower than the throughput of a single worker core using the same burst size. With regards to the queue configurations, there are a couple of small advantages for configurations using one inbound queue when using two and three worker cores. In general, however, it would seem it is not a major factor when it comes to throughput.
Figure 5.14: Throughput of the event implementation using 1 worker core using the IMIX packet stream

Figure 5.15: Throughput of the event implementation using 2 worker cores using the IMIX packet stream

Figure 5.16: Throughput of the event implementation using 3 worker cores using the IMIX packet stream
5.5.2 Latency

Figures 5.17, 5.18, and 5.19 show the latency of the event implementation. It is clear that the burst size is the major factor when it comes to latency as well. There are some spikes in the latency with different queue configurations but there is no clear trend there; the queue configuration does not seem to make much of a difference in most cases. There are a couple of exceptions in the graphs with two and three worker cores where some configurations using queue configurations N-1 and N-M display fairly high latency spikes, this is most likely due to queues becoming full. An interesting thing to note here is that the latency increases overall when going from one to two workers. This is due to how the number of allowed events is calculated, see Table 3.5. Since the processing rate of two workers is not enough to process at the rate at which the receive core is generating packets, the system fills up with events. This together with the increased queue space enables higher latencies. The drop in latency with a burst size of 32 and 64 using two workers and a burst size of 16, 32, and 64 using three workers is most likely due to the queues becoming empty. This is probably caused by attempting to enqueue large burst sizes to the default software scheduler of DPDK. It was observed during testing that the default software scheduler would deny an enqueue if it would put the number of events in the port buffer over the maximum limit of 64. This caused enqueues with a burst size of 32 to often fail as the scheduler would not have time to empty the port buffer before the next enqueue. Such behavior would reduce the rate at which the receive core could enqueue which would, in turn, explain why the queues could be emptied while the throughput is low.

![Graphs showing latency with different burst sizes and queue configurations](image-url)

Figure 5.17: Latency of the event implementation using 1 worker core using the IMIX packet stream
5.5. Eventdev-Based Implementation Evaluation

Figure 5.18: Latency of the event implementation using 2 worker cores using the IMIX packet stream

Figure 5.19: Latency of the event implementation using 3 worker cores using the IMIX packet stream

5.5.3 Cache Misses

The last level cache (LLC) misses of the event implementation using three workers can be seen in Figure 5.20. Strangely enough, there appears to be a different trend here compared to with the ring implementation as the percentage of LLC misses first increases slightly and then decreases as the burst size increases. This is likely due to the increasing risk of packet data dropping out of the cache as each core handles an increasingly large burst at a time. The drop at the three highest burst sizes could be due to the drop in throughput. If the packets are processed at a slower pace, there is a larger chance that a packet can be processed in its entirety before being flushed out by new packets arriving.
5.6 Eventdev-Based Implementation Utilizing Chained Packet Buffers Evaluation

The following sections present the results of the configuration evaluation of the EventDev-based implementation using chained packet buffers on the six core system. The throughput, latency, and cache measurements are presented here as well as some observations and discussion on how the configuration parameters affect them. The presented results were generated using the IMIX packet stream.

5.6.1 Throughput

The throughput when varying the queue configuration and burst size using one, two, and three workers can be seen in Figures 5.21, 5.22, and 5.23 respectively. Initially, the throughput increases as the burst size increases but this trend ends when using three workers. As can be seen in Figures 5.22 and 5.23 the throughput of three workers when encrypting using a burst size of 64 is at the same level as the throughput of two workers using the same burst size. It would seem the choice of queue configuration has no significant impact on the throughput of the application. The trend here resembles the one seen with the ring-buffer based implementation more than the event implementation. This is probably due to the fact that this implementation is not affected by the enqueue size limitation of the default Eventdev software scheduler.
5.6. Eventdev-Based Implementation Utilizing Chained Packet Buffers Evaluation

Figure 5.21: Throughput of the event-chained implementation using 1 worker core using the IMIX packet stream

Figure 5.22: Throughput of the event-chained implementation using 2 worker cores using the IMIX packet stream

Figure 5.23: Throughput of the event-chained implementation using 3 worker cores using the IMIX packet stream
5.6.2 Latency

In terms of latency, the burst size is still the most influential configuration parameter as can be seen in Figures 5.24, 5.25, and 5.26. The latency increases with the burst size, much like with the single-threaded implementation. The curve of the latency measurements resembles the single-threaded implementation, see Figure 5.3, more than the other parallelized implementations as the latency appears to be linearly proportional to the burst size. This is, however, not true when using three workers as can be seen in Figure 5.26. The huge spikes in this graph are difficult to analyze as they sometimes appear in different configurations or disappear completely over multiple test runs. As these elusive spikes have appeared in the test data from the event implementation as well, it is possible it is somehow related to the default software scheduler. It should also be noted that the average latency of the event-chained implementation increases when going from two to three workers even though the throughput increases. This could also be caused by the default software scheduler as it keeps a buffer for each event port as well as each event queue. This means that even if only one queue is used, the number of actual buffers used by the scheduler increases with each added worker. This means there are more places where events may have to wait for periods of time before the scheduler has time to move them on to the next buffer. Another interesting thing to note is that the minimum latency decreases when going from one to two workers but increases when going from two to three workers. The overall unpredictable behavior of the application when using three workers is likely due to the workers sometimes being able to process packets faster than the receive core generates them and sometimes not. Spikes in average latency occur when the queues manage to fill up and the workers are not able to empty them quick enough. It is also important to note that when going from one to two workers, the number of allowed events in the system only increases by two due to the lower bound of 128, see Table 3.5. When going from two to three workers, however, the value increases to 195. As previously mentioned, more queue space allow for higher latencies in those cases where the queues fill up. Finally, the queue configuration does not seem to have much of an effect on the latency of the application except for a couple of random spikes that were not consistent over multiple runs.

![Figure 5.24: Latency of the event-chained implementation using 1 worker core using the IMIX packet stream](image-url)
5.6. Eventdev-Based Implementation Utilizing Chained Packet Buffers Evaluation

Figure 5.25: Latency of the event-chained implementation using 2 worker cores using the IMIX packet stream

Figure 5.26: Latency of the event-chained implementation using 3 worker cores using the IMIX packet stream

5.6.3 Cache Misses

The last level cache (LLC) misses of the event-chained implementation using three workers can be seen in Figure 5.27. As previously observed with the ring implementation, the number of LLC misses increases with the burst size and they are on the same level (4 – 10%) as that implementation as well. Both the ring and event-chained implementations display higher LLC miss rates than the event implementation for higher burst sizes. The reason for this is the low throughput of the event implementation at higher burst sizes since a lower throughput means the rate at which the cache is flushed is reduced.
5.7 Configurations Chosen for Further Evaluation

In this section the values of the configuration parameters that were discovered to be insignificant or detrimental from a performance perspective are discarded and the chosen configuration of each implementation for further evaluation are presented.

5.7.1 Queue Configuration

This configuration parameter was shown to have little to no effect on the throughput and latency in most cases. Worth noting though, is that using the N-1 and N-M queue configurations cause more latency spikes than other queue configurations. However, the parameter does have an impact from the perspective of load balancing and memory usage. Using only two queues, instead of two queues per worker (of the same size), inherently lowers memory usage. Using only a single inbound queue also comes with load balancing. If a single worker is slower than the other workers, that worker does not have a queue that is full but rather does not dequeue from the global queue instead, allowing other workers to process those packets. This way, all workers has the possibility to dequeue the packets that has been waiting the longest. For these reasons, the 1-1 configuration has been chosen as the one to use for all implementations as it provided the best, or close to the best, performance and removes the need for doing per-worker load balancing.

5.7.2 Burst Size

The burst size proved to have the largest impact of all parameters on both the throughput and latency. The best performance was achieved with a burst size of 8 for the single-, event- and event-chained implementations while a burst size of 4 gave the best performance for the ring implementation. The were some spikes in latency observed in the Eventdev-based implementations when using the middle-of-the-road burst sizes but, as the spikes appeared sporadically, these have been ignored when choosing configurations.

5.7.3 Number of Workers

As there was no decline in the maximum throughput of each implementation when increasing the number of workers, it will be set to the highest possible for each implementation. This means that the ring buffer-based implementation will use four worker cores and the Eventdev-based implementations will use three worker cores on the six core system. This was chosen as the priority is to maximize throughput rather than minimizing latency.

Figure 5.27: Last level cache miss rates of the event-chained implementation using 3 worker cores using the IMIX packet stream
5.7.4 Chosen Configurations

In Table 5.2, the chosen configuration for each implementation can be seen.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Queue Conf.</th>
<th>Burst Size</th>
<th>Number of Workers</th>
<th>Cores Utilized</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>N/A</td>
<td>8</td>
<td>N/A</td>
<td>1</td>
</tr>
<tr>
<td>ring</td>
<td>1-1</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>event</td>
<td>1-1</td>
<td>8</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>event-chained</td>
<td>1-1</td>
<td>8</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5.2: Parameters of the chosen configurations

5.8 Comparison of the Chosen Configurations

Using the chosen configurations presented above, this section will present some more in-depth results and compare the different implementations on the six core system. The throughput, latency and speedup of the different implementations are presented and discussed below, followed by more in-depth latency as well as and throughput results with different packet sizes.

5.8.1 Throughput Comparison

A performance comparison with respect to throughput, in Mpps as well as Mbps, and latency can be seen in Figures 5.28, 5.29, and 5.30 respectively. These results are using the six core system.

Figure 5.28: Throughput in Mpps of the chosen configuration of each implementation using the IMIX packet stream
5.8. Comparison of the Chosen Configurations

Figure 5.29: Throughput in Mbps of the chosen configuration of each implementation using the IMIX packet stream

Figure 5.30: Latency of the chosen configuration for each implementation using the IMIX packet stream

The average throughput, latency and speedup for the top performing configurations compared to the single-threaded implementation can be seen in Tables 5.3 and 5.4 for encryption and decryption, respectively. The speedup is calculated from the throughput in million packets per second (Mpps).

The event implementation exhibits the lowest speedup of 2.46x when encrypting while the event-chained and ring implementations exhibit speedups of 2.62x, and 2.68x respectively. All of the implementations perform similarly or slightly worse when decrypting compared to encrypting in terms of throughput. The speedup, however, is similar to when encrypting with speedup of 2.68x, 2.46x, and 2.54x for ring, event, and event-chained implementations respectively. In general, the throughput is lower and the latency higher when decrypting than when encrypting.
5.8. Comparison of the Chosen Configurations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Mpps</th>
<th>Mpps</th>
<th>Average latency</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>1.59 Mpps</td>
<td>7074 Mbps</td>
<td>4.7 $\mu$s</td>
<td>1.00x</td>
</tr>
<tr>
<td>ring</td>
<td>4.26 Mpps</td>
<td>18944 Mbps</td>
<td>5.5 $\mu$s</td>
<td>2.68x</td>
</tr>
<tr>
<td>event</td>
<td>3.92 Mpps</td>
<td>17326 Mbps</td>
<td>38.4 $\mu$s</td>
<td>2.46x</td>
</tr>
<tr>
<td>event-chained</td>
<td>4.18 Mpps</td>
<td>18477 Mbps</td>
<td>37.7 $\mu$s</td>
<td>2.62x</td>
</tr>
</tbody>
</table>

Table 5.3: Throughput, latency, and speedup for the top performing configurations compared to the single-threaded implementation during encryption using the IMIX packet stream.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Mpps</th>
<th>Mpps</th>
<th>Average latency</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>1.53 Mpps</td>
<td>6618 Mbps</td>
<td>4.8 $\mu$s</td>
<td>1.00x</td>
</tr>
<tr>
<td>ring</td>
<td>4.10 Mpps</td>
<td>17772 Mbps</td>
<td>7.8 $\mu$s</td>
<td>2.68x</td>
</tr>
<tr>
<td>event</td>
<td>3.77 Mpps</td>
<td>16247 Mbps</td>
<td>92.3 $\mu$s</td>
<td>2.46x</td>
</tr>
<tr>
<td>event-chained</td>
<td>3.88 Mpps</td>
<td>16723 Mbps</td>
<td>150.1 $\mu$s</td>
<td>2.54x</td>
</tr>
</tbody>
</table>

Table 5.4: Throughput, latency, and speedup for the top performing configurations compared to the single-threaded implementation during decryption using the IMIX packet stream.

5.8.2 Latency Comparison

Below follows a more in-depth presentation and discussion of the latency for each of the implementations.

Unsurprisingly, the single-threaded implementation has a very low and stable latency as can be seen in the histogram in Figure 5.31. It has some, but very few, outliers up to 17 $\mu$s, although these are too few to show up in the graphs.

![Figure 5.31: Latency distribution of the chosen configuration for the single-threaded implementation using the IMIX packet stream.](image)

The ring implementation also has a quite stable latency, see Figure 5.32. For encryption, the mean latency is 5.5 $\mu$s and has a standard deviation of 0.9 $\mu$s. There are very few outliers meaning that the latency is very stable and can be expected to stay low. For decryption, the mean is 7.8 $\mu$s and standard deviation 1.9 $\mu$s. The spread is a bit higher which suggests that for decryption, packet size has a larger impact; the standard deviation has increased by more than 200%. For both encryption and decryption, the maximum latency recorded never
rises above 30 μs (again these cases are too few to show up in the graphs), showing that the implementation has very predictable latency.

![Image](image1.png)

Figure 5.32: Latency distribution of the chosen configuration for the ring implementation using the IMIX packet stream

The event implementation displays a much less concentrated latency distribution than the single-threaded and ring implementations, see Figure 5.33. Even though the latency measurements when encrypting are mainly focused around the mean of 38.5 μs, there are several outliers around 60 μs. The standard deviation of 9.3 μs reflects this. When decrypting, the spread is larger and the standard deviation increases to 11.5 μs. The mean value is also much higher at 92.3 μs, a difference of 240% compared to when encrypting.

![Image](image2.png)

Figure 5.33: Latency distribution of the chosen configuration for the event implementation using the IMIX packet stream

The latency distributions for the event-chained implementation are a bit more stable than for the event implementation, see Figure 5.34. The average latency when encrypting is about the same, 37.7 μs, but the standard deviation of 3.9 μs is much lower compared to the event implementation. The values when decrypting are once again very high as the mean increases by close to 400% to 150.1 μs compared to when encrypting. The standard deviation does not
5.8. Comparison of the Chosen Configurations

increase by quite as much but is still much higher at 9.8 µs, an increase of 251% compared to when encrypting. What is interesting to note is the relatively low spread of the measurements when decrypting. The average latency varied a lot between runs and so it would be expected to see a more random behavior with a larger spread. Instead it would seem like the measurements are consistently very high or low.

Figure 5.34: Latency distribution of the chosen configuration for the event-chained implementation using the IMIX packet stream

5.8.3 Comparison of Different Packet Streams

The throughput, both in Mpps and Mbps, and latency of the chosen configurations when varying the packet sizes—40, 660, and 1500 bytes—of the packet stream can be seen in Figures 5.35, 5.36, and 5.37 respectively. The throughput in Mpps, Figure 5.35, looks as expected with a higher amount of packets processed per second at lower packet sizes. It is logical that the throughput when using a stream with packets of size 660 bytes is slightly lower than with the IMIX stream as the average packet size of the IMIX packet stream is lower than 660 bytes. The throughput in Mbps, Figure 5.36, is high when the throughput in Mpps is low and vice versa. This is also expected as, even though the application can process more 40 byte packets than 1500 byte packets per second, the actual data rate in bytes per second achieved is very small. Increasing the packet size increases the data rate tremendously, reaching a maximum of 33 Gbps using 1500 bytes packets with the ring implementation.

The event-chained implementation performs on par, or better, than the event implementation. The throughput with a packet size of 660 and 1500 bytes is quite similar between the event and event-chained implementations. A major difference is the throughput with a packet size of 40 bytes where the event-chained implementation excels. This together with the high frequency of 40 bytes packages on the Internet means that it also gains an advantage with respect to throughput using the IMIX stream. The ring implementation achieves the highest throughput across the board, peaking at 4.26 Mpps—or 19 Gbps—for the IMIX stream.
5.8. Comparison of the Chosen Configurations

The Eventdev-based implementations exhibit very large latencies with packet sizes larger than 40 bytes when encrypting, see Figure 5.35. An interesting phenomenon is that when encrypting, the 660 and 1500 byte packet streams perform worst but when decrypting, the IMIX stream performs the worst. This is quite unintuitive and difficult to explain. It is not up to differences in the measuring process as all latency measurements are done in the same way when encrypting and decrypting. It might hint that the default Eventdev scheduler could be the cause behind this as this phenomenon does not appear in the ring or single implementations. The ring implementation performs almost as well as the single-threaded implementation no matter the packet size in term of latency.
5.9 Bottleneck Evaluation

This section presents and discusses the test results of the bottleneck evaluation on the six core system in terms of throughput and latency. The same configurations that were chosen for the evaluations in Section 5.8 were used. The modifications that were made for these tests were:

- The copying of packet data was disabled
- The encryption of packet data was disabled
- The copying and encryption of packet data was disabled

5.9.1 Throughput

In Figure 5.38, the throughput measurements when enabling and disabling the copying of packet data when generating packets are displayed. For the ring and single-threaded implementations the results show an increase in throughput with copying disabled. What is interesting, and counterintuitive, is a relatively large decrease in throughput for the Eventdev-based implementations. What is important to understand here is that this change reduces the work done on the receive core, allowing for faster packet generation. This causes a lot of wasted work on the receive core as it produces packets much faster than what can be distributed and processed. This results in many packets failing to be enqueued and being freed which can be a contributing factor to the decrease in performance for the Eventdev-based implementations as enqueueing packets is more expensive compared to the ring buffer-based implementation.
5.9. Bottleneck Evaluation

The throughput measurements of the bottleneck tests where encryption was disabled can be seen in Figure 5.39. The difference in throughput is more or less negligible for the parallelized implementations but for the single-threaded implementation it is huge. This is quite sensible seeing as the only meaningful work performed by the single-threaded implementation is the encryption and the copying of packets. The most important thing to note about these results is that they show that the worker processing rate is not the most limiting bottleneck of the parallelized implementations.

Finally, Figure 5.40 shows the throughput measurements of the final bottleneck test where the copying of packet data and the encryption was disabled. Here the throughput is doubled for all of the parallelized implementations while the single-threaded implementation displays an even higher throughput than in the previous bottleneck test. This is to be expected as the receiving core is the limiting factor when the processing rate of the workers increases. The high throughput measurements also show that the distribution of packets is not anywhere near the limiting factor of the application even though some implementations achieved better results than others. This test does a good job of showing the maximum speed at which the parallelized implementations are able to distribute and collect the packets flowing through the application.
5.9. Bottleneck Evaluation

These results show that performance in terms of throughput is not majorly impacted when either the copying of packet data or the encryption is disabled. However, disabling both copying and encryption almost doubles the throughput. One hypothesis for this behaviour is that the processing rate of the worker cores is close to the rate at which the receive core can generate packets. Another hypothesis is that the advantage gained from reducing memory contention by disabling, for example, packet data copying is nullified. This would be due to the fact that the packet data is not fetched to the cache by the receive core when not generating packets, forcing the worker cores to do it instead. This scenario is not too likely though as the packets cached while generating packets are probably flushed before the workers begin working on them.

5.9.2 Latency

The latency measurements of the bottleneck test where the copying of packet data when generating packets was disabled can be seen in Figure 5.41. As with the throughput measurements, the Eventdev-based implementations display worse performance when the packet data copying is turned off. However, the ring implementation shows vastly increased values as well for this metric. The most likely reason for this behaviour is that the receive core can now fill the inbound queues much faster than the worker cores can empty them. The inbound queues are then full for the entire duration of the test, causing an increased latency.
When encryption is disabled, the throughput does not vary much but the latency shows large differences for all of the parallelized implementations, as can be seen in Figure 5.42. The largest difference by far is with the Eventdev-based implementations whose average latency with encryption disabled is a sixth of its normal value. The single-threaded implementation displays a large decrease in average latency as well which is expected due to the high throughput. These are expected results since if no encryption is performed the only processing that is done is adding and removing IP and ESP headers. This causes the inbound queues to be emptied and packets to be processed much faster, reducing the time they spend in the application.

The latency measurements of the last bottleneck test where the encryption and copying of packet data were disabled can be seen in Figure 5.43. The values are very close to the the ones of the test where the encryption was disabled but the copying was enabled. This implies that the worker cores can again process the packets faster than the receive core can enqueue them, resulting in the queues being empty, lowering latency.
5.9. Bottleneck Evaluation

Figure 5.43: Latency of the chosen configuration of each implementation with copying of packet data and encryption enabled versus disabled using the IMIX packet stream

5.9.3 Cache Misses

Figure 5.44 shows that the cache performance is drastically reduced when turning off the copying of packet data when generating packets. The percentage of misses when accessing the last level cache (LLC) is more than 5x higher for the event implementation and around 7x higher for the ring implementation without packet data copying. The reason for this large increase in LLC misses is that, when copying is enabled, the packet data is inserted into the cache when it is copied all in one go but when the copying is disabled all sporadic accesses to the packet data during processing will be cache misses.

Figure 5.44: Last level cache miss rates of the chosen configuration of each implementation with copying of packet data enabled versus disabled using the IMIX packet stream

Whether encryption is performed or not does not appear to have a huge effect on the cache performance as can be seen in Figure 5.45. It appears to have a larger impact on the cache performance of the ring implementation than the Eventdev-based ones though. The reason for this very modest impact on LLC misses is most likely due to the fact that when copying of packet data is enabled, the packet data is inserted into the cache at the time of copying. When the encryption then happens, the packet data is already in the cache, resulting in few cache misses.
Lastly, when both encryption and copying of packet data is disabled, the effect is somewhat lower than when only disabling packet data copying. The difference is the cache performance of the single-threaded implementation which is drastically reduced, from having close to no last level cache misses to a 25% miss rate. A possible explanation to this is that when no copying of encryption is done, all of the sporadic accesses to packet data causes cache misses. The impact is not as severe as when only copying of packet data is disabled, since the encryption can not cause any cache misses in this case.

As can be seen in Figure 5.44, the cache miss rate is drastically increased when the receive core is able to generate packets faster than what can be distributed and processed. As previously mentioned, the throughput drops when this happens as can be seen in Figure 5.38. A reasonable conclusion to draw from this is that the Eventdev-based implementations are more sensitive to this kind of cache thrashing than the ring implementation.

5.10 Hyper-Threading Evaluation

This section presents the results of how Hyper-Threading impacts the performance of the application on the six core system as well as some analysis of the results. With Hyper-
5.10. Hyper-Threading Evaluation

Threading disabled, each thread ran on one physical core for a total of six physical cores. With Hyper-Threading enabled, two threads shared the same physical core for a total of three physical cores. Figures 5.47 and 5.48 show the throughput and latency, respectively, of the chosen configurations for each implementation with Hyper-Threading disabled and enabled respectively.

Running two threads on the same physical processor dramatically reduces performance, both in terms of throughput, see Figure 5.47, and latency, see Figure 5.48. Both the event and event-chained implementations achieve the same throughput with Hyper-Threading enabled, even though the event-chained implementation performs better than the event implementation with Hyper-Threading disabled. The event implementation achieves 2.62 Mpps for encryption and 2.48 for decryption. Event-chained achieves 2.60 Mpps for decryption and 2.45 for encryption. This implies that both of these implementations hit some other bottleneck, presumably a memory bandwidth limitation or full utilization of the CPU cores. To further strengthen this claim, the performance achieved for the ring implementation is much higher compared to the Eventdev-based implementations. The ring implementation does not hit this bottleneck, and can utilize the Hyper-Threads more efficiently than the Eventdev-based implementations. Ring achieves 3.43 Mpps for encryption and 3.18 Mpps for decryption. For encryption, the event implementation performs at 67% of its original performance, the event-chained one at 62% and ring one at 80%. For decryption, the numbers are similar at 66%, 63% and 77%, respectively. Using only half the number of cores compared to with Hyper-Threading disabled, the implementations perform better than half the throughput, showing that there is some merit to Hyper-Threading but that the performance impact is more likely than not to make this approach viable.

![Graph](image)

Figure 5.47: Throughput of the chosen configuration of each implementation with Hyper-Threading (HT) enabled versus disabled using the IMIX packet stream

Latency greatly increases when using Hyper-Threading, see Figure 5.48. The event implementation has a latency 7.3x greater when encrypting and 5.6x greater when decrypting, the event-chained implementation 12.4 and 15.3x respectively. The latency with the ring implementation increases by 15.3x when encrypting and 14.4x when decrypting. It is interesting to note that the event-chained implementation and the ring implementation suffers much more in terms of latency with Hyper-Threading than the event implementation. The latency achieved with Hyper-Threading enabled is much too high for most practical uses and shows, together with the low throughput, that Hyper-Threading is not a viable approach for this application.
5.11 Evaluation of the Impact of Reordering

This section presents and discusses the impact that disabling reordering has on the performance of the application on the six core system. The measurement data of the reordering evaluation tests is visualized in Figure 5.49 for throughput and Figure 5.50 for latency.

The impact that disabling reordering has on throughput is quite remarkable; the throughput increased in all implementations when having reordering enabled, see Figure 5.49. The largest difference is in the event implementation when encrypting where the throughput increased by 0.28 Mpps. The throughput of the event-chained and ring implementations increased as well, but not by as much. Reordering of the packets is obviously more work but it does not have a negative impact on the performance of the application. This is most probably due to the workload of the transmission core being quite low in comparison to the receive and worker cores. Therefore, it is reasonable that reordering does not have a negative impact on performance, but why does it have a positive one? It is possible that reordering has an effect on how the transmission core accesses the memory. This could, in turn, have an effect on the performance of the other cores as the memory resources are likely under heavy contention in this application. The increased work on the transmission core also slows it down, reducing the frequency at which it polls the outbound queue(s) for packets or events. It is unclear, however, if the contention on the queues affect the cost of enqueuing and dequeuing.

Since all of the threads in the application do somewhat similar work, mostly involving memory accesses, it is intuitive that it cannot fully take advantage of Hyper-Threading. The encryption and decryption in OpenSSL is also heavily vectorized, which makes it a particularly bad target for Hyper-Threading, as discussed in the article by Subhash et al. [60]. Yaoping et al. [59] evaluates similar applications to the one presented in this thesis, performing many memory accesses on large amounts of data, where they showed that these kinds of applications do not benefit from Hyper-Threading very much.

Figure 5.48: Latency of the chosen configuration of each implementation with Hyper-Threading (HT) enabled versus disabled using the IMIX packet stream

![Figure 5.48: Latency of the chosen configuration of each implementation with Hyper-Threading (HT) enabled versus disabled using the IMIX packet stream](image-url)
5.11. Evaluation of the Impact of Reordering

Reordering appears to also have a very large impact on the latency of the Eventdev-based implementations, less so on the ring buffer-based implementation, see Figure 5.50. The event implementation sees the largest variation as it is 3.0x higher when encrypting and 5.0x higher when decrypting with reordering enabled. The event-chained implementation also sees a large impact when decrypting as the latency is 2.2x higher with reordering enabled. The ring implementation shows similar latencies when encrypting and somewhat lower latencies with reordering turned off. The latency measurements are less surprising than the throughput measurements. The more surprising aspect is the variations as the increase in latency when reordering differs a lot between the encryption and decryption tests. The variations between modes may be due to the difference in processing time when encrypting as opposed to decrypting. If a packet takes longer to process, the wait time will increase for packets in the reorder buffer. The variations between implementations could be caused by differences in how often bursts end up out of order when passing through the application. The use of a service core does make this aspect of the application a bit less predictable for the Eventdev-based implementations. This is due to the service core being responsible for moving the events between the event ports and queues.

Figure 5.49: Throughput of the chosen configuration of each implementation with reordering enabled versus disabled using the IMIX packet stream

Figure 5.50: Latency of the chosen configuration of each implementation with reorder enabled versus disabled using the IMIX packet stream
5.12 Extended Testing with 14 Cores

This section presents and discusses the results produced on the 14 core system, which was acquired late in the project and could therefore not be used for all of the different tests.

5.12.1 Ring Buffer-Based Implementation Evaluation

On the 14 core system, a lower overall throughput—Figure 5.51—and higher overall latency—Figure 5.52—can be seen. This is most likely simply due to the different hardware and configuration of the 14 core system. Up to four workers, the pattern is strikingly similar to the results acquired with the six core system. With the 14 core system, larger burst sizes—especially 8 and 16—performs a little bit better but this is most likely due to other differences than the number of cores. Increasing the core count above four, the lower burst sizes starts to perform better. The peak performance is reached with five workers for encryption and six workers for decryption. This shows that the implementation may have some performance to gain from higher core counts, but not a significant amount. Increasing the worker count above six shows a steady decline in performance.

![Figure 5.51: Throughput of the ring implementation with queue configuration 1-1 using the IMIX packet stream](image)

The latency also follows a similar pattern to the previous results. Starting at a high latency around 240 $\mu$s it quickly drops as the worker count increases. On the 14 core system, the lowest values are achieved with 5 workers. For encryption, more workers do not impact the latency significantly. For decryption, the latency drastically increases as the worker count goes above 8. These results are similar to what was seen with the six core system and is expected although it gives some insight into how the application scales with even more cores.
5.12. Extended Testing with 14 Cores

The pattern observed in the throughput results from the corresponding tests of the ring implementation continue for the event implementation, see Figure 5.53. The throughput with higher burst sizes keeps decreasing as the number of workers increases. A peak in throughput appears to be reached at five workers using a burst size of two. At higher degrees of parallelism there is no gain in throughput.

The latency appears to also be following the patterns observed in the corresponding tests of the ring implementation of higher values with higher burst sizes, see Figure 5.54. The differences being that the event implementation achieves a lower latency with a small number of workers and huge spikes when decrypting using 10 and 11 workers. These spikes in latency are difficult to explain but most likely they have something to do with the very large queue space at 10 and 11 workers paired with a low throughput. At 10 workers, for example, the number of allowed events is at 2560 which is quite high. If the throughput, at that point, would drop below the packet generation rate, it would result in very high latencies. Whether this explains the spikes entirely is difficult to say.
5.12. Extended Testing with 14 Cores

Figure 5.54: Latency of the event implementation with queue configuration 1-1 using the IMIX packet stream

5.12.3 Eventdev-Based Implementation Utilizing Chained Packet Buffers

Evaluation

The throughput results, shown in Figure 5.55, are similar to the ones of the event implementation. The main differences being the smaller negative impact of the number of workers on the throughput when using larger burst sizes, similar to the results for the six core system.

Figure 5.55: Throughput of the event-chained implementation with queue configuration 1-1 using the IMIX packet stream

The latency measurements are fairly similar to the ones of the event implementation, see Figure 5.56. There is a trend of the latency being higher at the low and high values of the number of workers. The trend is not as easy to spot, however, due to the massive spike in latency of 7.6 ms when decrypting using 11 workers. The same reasoning as for the spikes in the event implementation goes, that the throughput drops below the packet generation rate while the queue space is very large. These results show that the way the number of allowed events is calculated means that using additional workers after the throughput has stagnated presents a threat to the latency of the Eventdev-based implementations. This could possibly be solved by changing the formula to be less linear but at the same time there is no real reason to use more workers when there is no more performance to be gained.
5.12. Extended Testing with 14 Cores

5.12.4 Extended Bottleneck Evaluation

Since it was not clear what was the limiting factor when disabling the copying of packet data during the bottleneck tests, this test was chosen to also be run on the 14 core system to investigate this further. This section presents and analyses the results of this test.

The measurements from these tests can be seen in Figures 5.57, 5.58, and 5.59. The trends in these measurements are very similar to the ones from the non-bottleneck tests run on the 14 core system. The difference is the peak in throughput which occurs much later at a worker count of seven for the ring implementation and eight for the Eventdev-based implementations. It should also be noted that the ring implementation reached a peak of 7.1 Mpps with 7 workers and event-chained implementation at 7.2 Mpps with 8 workers while the event implementation only reaches just below 6.5 Mpps with 8 workers.

Figure 5.56: Latency of the event-chained implementation with queue configuration 1-1 using the IMIX packet stream

Figure 5.57: Throughput of the ring implementation with queue configuration 1-1 and copying of packet data disabled using the IMIX packet stream
The most interesting result from these tests is the large increase in throughput when reducing the contention on the memory resources by disabling the copying of packet data on the receive core. From these results it can be concluded that the receiving core is indeed the factor limiting the level of parallelism in this application.

5.13 Results Conclusion

Based on the classification scheme from the article by Xie and Loh [22], the implementations would seem to be closest to the devil class, i.e. a process that has a high miss rate of the shared cache and low reuse of cached data. This due to the constant flow of new data causing cold cache misses and the low re-usage of cached data. According to the article by Zhuravlev et al. [21], devil-type processes are not well suited to running on different cores on the same CPU due to the heavy load they cause on the memory bus, memory controller, and prefetching hardware. This is likely one of the reasons behind the fact that the throughput increase tapers off at fairly low levels of parallelism.
When comparing the implementations, the performance of the ring implementation is consistently better than the Eventdev-based implementations, both when it comes to throughput and latency. This is a bit surprising, especially at higher levels of parallelism where a dedicated scheduling core would seem beneficial. It seems like other bottlenecks of the system are reached long before the bottleneck of distributing the packets. This was also shown to be the case in the bottleneck tests where the packet generation proved to be the main bottleneck of the system. It should be noted, however, that the removal of the packet data copying also reduces the contention on the memory resources substantially. These observations together with the devil classification of the packet processing means that all overhead, especially memory-accessing overhead, on the receive core is going to have a negative impact on the performance of the application.

Something that affects the Eventdev-based implementations is the cost of enqueuing larger bursts compared to the ring buffer-based implementation. The event-chained implementation has an advantage over the event implementation though as the event implementation has to create and enqueue a new event struct for each incoming packet, even if only 64 are actually enqueued in the end, while the event-chained implementation does not have this limitation. The discrepancy between the two implementations at higher burst sizes is at its most obvious when comparing Figures 5.16 and 5.23. The conclusion that follows from these tests is that creating and handling an event for each packet is not a preferable approach and it might be what causes the event implementation to fall behind the event-chained and ring implementations.

5.14 Method Discussion

This section discusses the methodology used for the development as well as for the evaluation of the application. As discussed in Chapter 4, the evaluation method used was chosen to eliminate many other factors of a real-life scenario, such as network card interfaces etc. This comes with some limitations of the way the application could be tested.

Firstly, due to the measurements having to take place inside of the application, some processing power has to be devoted to the actual measurements of throughput and latency, especially taking timestamps. This can adversely affect the application, not only in actual clock cycles used but also in cache thrashing and memory usage.

Further, due to the way our application was tested—by loading pre-defined packets into the application’s memory and then replaying the same packets over and over—the network stream modelling was not as accurate or realistic as could have been hoped for. The distribution presented in the technical report by Sinha, Papadopoulos, and Heidemann [67] is a continuous distribution but was applied discretely with three different packet sizes when evaluating the application presented in this thesis. The coarse discretization of the packet size distribution probably had some effect on the throughput and latency of the application but doubtfully enough to affect any conclusions drawn from the result. The relatively small set of 1000 packets used for generating the packet stream can create patterns in the stream. These patterns could for example cause one core to always get more small packets than the others which would imbalance the work load and affect the results. The risk of this is higher when using multiple input queues though as the order in which the workers dequeue is bound to change constantly during a test run when using a single input queue.

Both of these issues could be avoided by using real network card interfaces and moving the measurements to other nodes on the network, the nodes sending and receiving the traffic.
This can also open up for use of real traffic generators, such as Cisco’s TReX\(^1\) to generate more realistic network traffic patterns. This does require more work in terms of setting up the actual network cards for best performance and tuning the hardware and operating system to achieve the best possible performance. In order to achieve the throughput presented in this thesis, the network cards must be at least 40 Gbps network cards as well. Due to resource and time constraints, as well as wanting to cut the network cards out of the equation, the choice was made to go with a more static network packet stream to evaluate the application.

\(^1\)https://trex-tgn.cisco.com/
Conclusions and Future Work

The aim of this thesis was to investigate how a solution for parallel processing of the Encapsulating Security Payload protocol of a single IPsec Security Association can be implemented, based on current data stream processing techniques and research, together with DPDK. To do this, three different implementations were developed; one based on the ring buffer library and two based on the Eventdev library. Due to suspicions of superfluous overhead of the documented usage of the Eventdev library, one implementation based on the Eventdev library utilized chained packet buffers.

In the paper by Koronla et al. [51] they claim—referring to the StrongSwan project [473]—that CPU based implementations of IPsec would require dozens of cores to reach a modest 1 Gbps. This has been proven false by this thesis as the application presented in this thesis can reach speed up to 33 Gbps by utilizing 6 cores, with the right configuration and packet stream. Worth noting is that this throughput is reached with a single Security Association which requires the workers to coordinate with sequence numbers. A more common scenario which does not require sequence number coordination is when multiple SAs are used and any SA is never handled by multiple cores at the same time. Such a scenario has a much larger need for load balancing, as the size of the flows might differ, but it is not limited by data synchronization. Using a more realistic Internet mix packet stream, the application can still reach an average throughput up towards 19 Gbps with an average latency as low as 10 µs. Even in the worst case scenario, with 40 byte packets, the data rate achieved, 2 Gbps, is twice the suggested data rate by Koronla et al. It is also worth noting that data rate is not necessarily the most important aspect for small packets. 40 byte packets has the lowest data rate, it also has the highest throughput in packets with over 4.5 million packets processed per second. The scaling achieved when increasing the number of cores saw the largest gain between 1 and 3 workers, with more workers than that providing little extra performance. The maximum speed up achieved was 267% using 6 cores, giving a speed up of 44.5% for each core added.

https://www.strongswan.org/
The ring implementation was shown to outperform the Eventdev-implementations, not only when it comes to the performance metrics but also when it comes to the scaling in performance with increasing levels of parallelism. This was an interesting result as it showed that the seemingly increased packet distribution capabilities of the Eventdev library did not provide much advantage. It was shown that the packet generation rate of the receiving core was limiting the performance before the scheduling of the packets could have any effect. The documented usage of the Eventdev library was concluded to introduce superfluous overhead as the implementation utilizing chained packet buffers performed better in all metrics. We propose the application structure presented in the article by Fusco and Deri [58] as a way forward as it mitigates this bottleneck by making use of Receive Side Scaling (RSS) and dedicated hardware threads for polling each receive queue. This solution is also focused on reducing the contention on memory resources which is also beneficial for the implementations presented in this thesis.

In conclusion, the Eventdev library does provide many features to assist with scheduling but the issue of the receiving core not being able to generate packets fast enough prevents the scheduler from providing an edge over less complex solutions such as the implementation utilizing the ring buffer library.

6.1 Future Work

It would be interesting to see the application presented in this thesis evaluated using a more realistic network stream model. There is much work done in the field of network stream modelling and analysis of how the Internet traffic has long-range dependence and is very self-similar [68, 69, 70]. It would be interesting to see how the throughput and latency of the application presented in this thesis is impacted with a more realistic model of Internet traffic. It would also be interesting to see how the application performs in real world scenarios with physical network cards.

The default software scheduler used for the Eventdev implementations in this thesis may cause much of the odd behavior as well as some performance limitations. An interesting development of this thesis would be to evaluate different schedulers, software or hardware, that were not evaluated in this thesis. There is the potential to use hardware schedulers which could allow for the performance of Eventdev-based implementations to surpass the ring buffer-based solution.

Another topic that could be interesting to investigate is that of hardware accelerators for the crytographic operations. The DPDK API for crytographic devices has support for a range of different hardware accelerators, and there has been work done in offloading these kinds of operations onto GPUs as well. As Verner et al. showed in their paper [54], and Vasiliadis et al. in theirs [57], hardware acceleration can increase performance and can even be used in such a way that the latency is kept at an acceptable level.

While this thesis did go into some detail regarding memory usage and cache behaviour, it would be interesting to see further investigation into this area within data stream processing, especially when it comes to high performance networking applications. There has been general work done in this area, such as the paper by Zhuralev, Blagodurov and Fedorova [21] as well as the classification system presented by Xie and Loh in their paper [22]. This could be applied more rigorously to an application, such as the one presented in this thesis. It would also be interesting to see how the work can be scheduled more efficiently by splitting up the work done in a pipelined fashion in multiple cores instead of parallelizing the workers as done in this thesis.
As previously discussed, there has been research done in elastic solutions which vary the level of parallelism in an application to optimize the performance [55] as well as how to mitigate issues created with this approach [56]. As seen in the results where up to 12 worker cores were used, the performance does not keep scaling with the number of workers for very long. A potential future prospect would be to investigate whether different packet sizes change the way the performance scales. This would open up for an elastic solution which could change the level of parallelism when the input packet type changes.


