Masaryk University
Faculty of Informatics

Optimization and implementation of AF_KTLS in user-space applications

Master's Thesis

Ananya Chatterjee

Bangalore, December 2017

Masaryk University
Faculty of Informatics

Optimization and implementation of AF_KTLS in user-space applications

Master's Thesis

Ananya Chatterjee

Bangalore, December 2017

MASARYK UNIVERSITY
Faculty of Informatics

MASTER'S THESIS DESCRIPTION

Student:	Ananya Chatterjee
Programme:	Informatics
Field of Study:	Information Technology Security (eng.)
Field guarantor:	prof. RNDr. Václav Matyáš, M.Sc., Ph.D. (BITA)
Thesis supervisor:	prof. RNDr. Václav Matyáš, M.Sc., Ph.D.
Department:	Department of Computer Systems and Communications
Title of the thesis/dissertation:	Optimization and Implementation of AF_KTLS in user-space applications
Title of the thesis in English:	Optimization and Implementation of AF_KTLS in user-space applications

Description:

AF_KTLS is a kernel module that introduces parts of TLS and DTLS protocols to the Linux kernel. After a handshake is done in userspace all the key material is passed to the Linux kernel via appropriate socket system calls supported by AF_KTLS. After that, the user-space application can transparently use AF_KTLS type socket that transparently performs encryption and decryption of traffic. The main advantage of using AF_KTLS lies in applications that serve static content that can be sent using sendfile(2) system call or by applications that do not necessarily need decrypted content available in user-space. An example of such applications could be OpenConnect VPN where AF_KTLS could be used to optimize copies from user-space to kernel-space and vice versa just for encryption or decryption. A similar use-case can be found also in TCP/HTTP balancer HAProxy. This work will extend the AF_KTLS [1, 2] kernel module that introduces parts of TLS [3] and DTLS protocols to the Linux kernel.

The goal of this thesis is to analyse possible use-cases and scenarios where AF_KTLS could be beneficial and to propose changes in user-space applications or directly inside AF_KTLS module implementation that would lead to optimization.

The student will also implement the changes, evaluate benchmarks on implemented changes and suggest other optimization alternatives, based on bottlenecks found in benchmarks.

The student will also implement some changes as agreed with the supervisor and technical consultant, evaluate benchmarks on implemented changes and suggest other optimization alternatives, based on bottlenecks found in benchmarks.

Literature:

[1] Pokorný, Fridolín, "Linux VPN Performance and Optimization". Brno, 2016. URL: http://www.fit.vutbr.cz/study/DP/DP.php?id=18032 [2] Pokorný, Fridolín, "AF_KTLS - TLS/DTLS Linux kernel module", FOSDEM, Brussels, Belgium, 2017. [3] Dave Watson, "Kernel TLS (Transport Layer Security) Socket", NETDEV 1.2, Tokyo, Japan, 2016. URL: http://netdevconf.org/1.2/papers/ktls.pdf

I agree with the description (signature, date):

Ananya Chatterjee	prof. RNDr. Václav Matyáš, M.Sc., Ph.D.	prof. RNDr. Václav Matyáš, M.Sc., Ph.D.
student	**Master's thesis supervisor**	**field guarantor** BITA

Statement of an author of a school work

Student´s Name and UČO: _____

I acknowledge that the Masaryk University (MU) is entitled in accordance with the law (Article 35 § 3 and 4 of the Copyright Act No. 121/2000 Sb.) to use for educational and other internal purposes on a non-commercial basis my thesis or my other school work, which I authored to fulfill my study obligations towards MU (my work).

The use of my work for internal purposes includes the use of the original work as well as of its derivatives and might consists also of assigning of my work for additional processing to another student or a member of the MU academic community, or of making it available as a basis for a creation of a derivative thesis or other school work at MU. Any such use of my work will acknowledge my authorship, the original name and source of my work and will be conducted exclusively in order to further develop educational and other interests of MU related to further development and utilization of my work within its academic community.

I also acknowledge my duty to inform MU at the latest upon the submission of my work about my intent to further develop or use my work at MU or elsewhere or about any other relevant issues related to my work.

Brno, _____ _____

Student´s signature

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ananya Chatterjee

Advisor: prof. RNDr. Václav Matyáš

Acknowledgement

I would like to express my gratitude to my supervidor, prof. RNDr. Václav Matyáš, for his continued support, comments and supervision of my thesis. Furthermore I would like to thank Ing. Fridolín Pokorný for introducing me to the topic, guidance, patience and support on the whole way. Last but not the least, I would like to thank Red Hat Czech for providing me with the opportunity to work on this topic.

Abstract

AF_KTLS is a kernel module that introduces parts of TLS and DTLS protocols to the Linux kernel. User-space applications can transparently use AF_KTLS type socket to perform encryption and decryption of traffic. This thesis provides an analysis of the AF_KTLS module and identifies use cases where it could be used to optimize performance for user-space applications. The changes required for the optimization with AF_KTLS are then implemented and the results are benchmarked to identify bottlenecks and suggest possible solutions.

Keywords

TLS, AF_KTLS, load balancing, HAProxy, optimization, security

Contents

List of Tables

List of Figures

1 Introduction

Nowadays, heavy-use websites, such as Github, Instagram and Twitter, accept a large number of concurrent requests from clients and serve text, image, video, and application data. Speed and reliability are the primary concerns for these websites. Efficient scalability to meet this high volume traffic requires the concept of load balancers.

A load balancer is a device that is used to efficiently distribute network or application traffic across a cluster of servers, designated as the server pool. The load balancer sits between the clients and the server pool, accepts incoming network and application traffic, and distributes the traffic across multiple backend servers. It works by routing client requests across all available servers in the server pool that are capable of fulfilling the requests. It ensures that the requests are served such that the speed is maximized and maximum capacity is utilized. It also takes care of performance degradation by not assigning a single server to do the major share of work. If one server goes down, the load balancer redirects traffic to the remaining servers. Load balancers are important because they reduce individual server load and prevent single point of failure scenarios. This improves the overall application availability and responsiveness. Figure 1 shows the schematic representation of a load balancer environment.

An SSL load balancer [1] additionally performs encryption and decryption of traffic transported via HTTPS. HTTPS uses the Secure Sockets Layer (SSL) protocol or the Transport Layer Security (TLS) protocol [2] to secure HTTP traffic as it crosses the network. SSL and TLS are the standard protocols for encrypting HTTP data before being sent across the network. This prevents unauthorized data disclosure to third parties who intercept it. It is vital to use the protocols for protecting sensitive data such as user credentials when they are transmitted over a public network like the Internet.

The SSL load balancer intercepts the incoming encrypted client requests, decrypts them and distributes them across a group of backend servers. It also gets the plaintext response from the servers and encrypts it before sending it back to the client. We can think of the SSL load balancer as the server-side SSL endpoint for encrypted client connections.

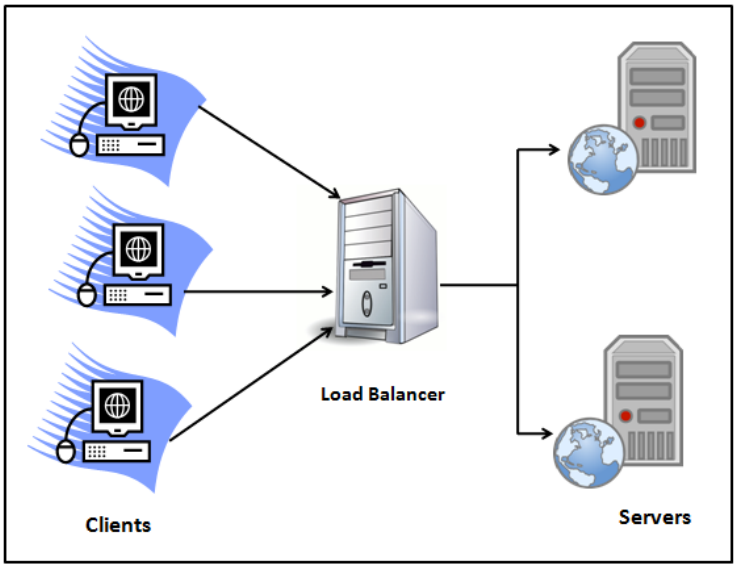

Figure 1.1: Schema of a load balancer.

There are two ways in which the SSL load balancer operates:

- The SSL load balancer can be configured to decrypt a client request, and to forward the request to the server in the unencrypted form. It encrypts the response coming from the server, before returning it to the client. This approach is usually taken when the load balancer and server are on a secure network.

- In case of an insecure network, the SSL load balancer can be configured to decrypt the request, and re-encrypt it before forwarding it to the server. The same process is followed while serving the client request as well.

Encryption and decryption are considered to be computationally intensive processes. Shifting this responsibility to the SSL load balancer speeds up content delivery and improves the user experience. Moreover, in the first case, SSL certificates need to be installed and managed on the load balancer only, instead of every web and application server. This significantly reduces administrative overhead.

The most common way of load balancing is to use the transport layer. The network traffic is hosted on multiple servers using layer 4 load balancing. The user traffic is forwarded based on the IP range and port. An SSL load balancer uses the TLS protocol as follows:

- The client using the TLS protocol sends a **client hello** message that lists the cryptographic information such as the TLS version, the CipherSuites supported by the client, and a `nonce`.
- The SSL load balancer responds with a **server hello** message that contains the CipherSuite chosen by it, the session ID, and another `nonce`. The load balancer also sends its digital certificate. It can also ask for the client certificate to perform client authentication.
- The client verifies the digital certificate of the load balancer, and sends a `nonce` encrypted with the public key of the load balancer. This enables the client and the load balancer to compute the secret key. This key is subsequently used for encrypting messages. The corresponding process in repeated if client authentication is desired.
- The client sends the load balancer a **finished** message encrypted with the secret key. This completes the client part of the handshake.
- The load balancer sends the client a **finished** message encrypted with the secret key. This completes the load balancer part of the handshake.
- The load balancer and the client can now exchange messages encrypted with the shared secret key for the duration of the TLS session.

A user space application, such as the SSL load balancer, which requires SSL/TLS support implements the steps mentioned above by using user-space libraries. The SSL libraries take care of TLS handshake, client request, server response, data encryption and decryption amongst other functions. In this case, all the data to be encrypted must be in the user space and the encryption and decryption also take

place in the user space by using primitives provided by the SSL user-space library. The analysis of performance overhead caused by context switches and copies from/to user-space for a variety of applications is documented in the next chapter. An approach to overcome this is to move the encryption/decryption to the kernel space with the help of a kernel module that is configurable from user-space. This might increase the speed of operations and possibly increase throughput.

The objective of this thesis is to experiment with the integration of such a kernel module with an SSL load balancer application, and to test the optimization, if any, in the speed and throughput of the application.

The rest of the thesis deals with the implementation and testing of a chosen load balancer with the kernel module. Chapter 3 elaborates the proposed approach for the final integration and its implementation. Chapter 4 deals with the experimental setup and configurations used for testing and benchmarking of the implemented solution. Chapter 5 provides a detailed look at the results obtained from the benchmarking and dives into analysis of the derived experimental results. We then conclude this thesis by observing the optimizations, if any, to corroborate or contradict our initial study and envisioning the path to take in the future.

2 Background and Related Work

2.1 TLS & DTLS in Linux Kernel

The Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) [3] protocols provide transport and communication security for data transmitted over a computer network. The protocols provide two communicating applications with confidentiality (symmetric cryptography), integrity (message authentication codes) and authenticity (public-key cryptography). The TLS protocol, amongst others, comprises the TLS Record Protocol and TLS Handshake Protocol. The TLS Record Protocol performs the following tasks:

- Data encryption using symmetric cryptography. The keys for this are generated based on secrets negotiated by the TLS Handshake Protocol.
- Message integrity checks using a keyed Message Authentication Code (MAC).

The TLS Handshake Protocol has the following functions:

- Authentication of peer identity using asymmetric cryptography.
- Secure and reliable negotiation of shared secret.

The TLS Handshake Protocol allows peers to negotiate security parameters for the record layer, to authenticate each other by exchanging certificates and cryptographic information, to instantiate negotiated security parameters, and to report error conditions. TLS works over the TCP protocol and hence comes associated with the reliability of stream-oriented protocols.

DTLS is Datagram Transport Layer Security protocol for applications that have been designed to use datagram transport, which does not require or provide reliable or in-order delivery of data. The DTLS protocol preserves this property for the payload data. RFC 6347 describes DTLS as TLS over datagram. TLS has no internal facilities to handle lost and reordered packets, and therefore TLS implementations break for datagram transport. The purpose of DTLS is to make only

the minimal changes to TLS to make it work over unreliable transport protocol. It provides the same security functions as TLS (integrity, authentication and confidentiality), but under the UDP protocol for datagram transport.

GnuTLS [4] and OpenSSL [5] are widely used user-space libraries that implement TLS and DTLS. TLS implementations in the user-space incur significant overheads as shown in a study carried out by Facebook [6]. Facebook encrypts majority of its external traffic over HTTPS. In a TLS enabled service, OpenSSL's `SSL_read` and `SSL_write` primitives are used. OpenSSL requires all data to be in the user space to be encrypted and every operation in the user-space would involve context switches and data copies to and from the kernel. This contributes to an increase in cost due to a higher number of additional CPU cycles. The `read()` and `write()` system calls between the kernel and user-space lead to added overhead due to context switch and data copy. The Facebook study found that their SSL overheads resulted in approximately 2% of total CPU spent on copy from/copy to user space due to encryption, and approximately 10% of total CPU spent on encryption and decryption routines on machines that made heavy use of TLS.

Due to this overhead, the idea of optimizing TLS/DTLS by moving it into the kernel has been taken up by many developers.

Netflix [7] has improved a solution provided by Nginx that makes TLS use the `sendfile(2)` system call to perform a data flow from disk to socket. The `sendfile(2)` system call copies data between file descriptors to reduce the overhead of context switches and data copies between user-space and kernel. The session management stays in the application space and the encryption is inserted into the `sendfile(2)` data pipeline in the kernel. Netflix uses this implementation for serving static content.

Solaris [8] has also implemented a solution to move TLS/DTLS into the kernel space. The implementation of Record Protocol and Handshake Protocol were both moved to the kernel. This solution was not carried forward.

Facebook [9] investigated the use of `sendfile(2)` or `splice(2)` system calls to send static content directly from disk to the network, with zero-copy through user space. Facebook also experimented with the Kernel Connection Multiplexer (KCM) [10], but it required access

to the unencrypted bytes in the kernel. They subsequently found that encrypting the data in the kernel resulted in ideally zero copies, with only the encryption taking the bulk of the CPU usage. User space only needs to perform control functions. Facebook leveraged the existing Linux kernel crypto interface, AF_ALG [11]. The basic idea is that an AF_ALG socket and a regular TCP socket are both created. The TCP socket is used to do the handshake with the remote endpoint, which establishes key materials and security parameters for the Record protocol. The send and receive keys are passed to the AF_ALG socket using `setsockopt()` system call. An operational socket is created on the crypto socket and it is used in further processing, such as setting IVs and control messages. The file descriptor for the TCP socket is passed to the operational socket in a control message, the application can then read and write data from the operational socket. According to Facebook, this optimization of the in-kernel TLS showed 2-7% improvement in performance against the equivalent done in user space.

Fridolin Pokorny from Red Hat and Dave Watson from Facebook developed a Linux kernel TLS socket. The kernel handles the symmetric encryption and decryption, while the handshake protocol is carried out in the user space. TLS and DTLS framing, sending messages and receiving messages is implemented in the kernel. The implementation can also be used with `sendfile(2)` or `splice(2)` system calls. This led to the introduction of a new kernel module AF_KTLS **[12]**.

2.2 AF_KTLS Kernel Module

This kernel module introduces a socket that can be used to transmit data over TLS 1.2 or DTLS 1.2. AES GCM 128-bit is the supported cipher for authenticated encryption. The socket performs only data transmission. The handshake, re-handshaking and other control messages are served by the user-space by using OpenSSL or GnuTLS user-space SSL libraries. The module aims to eliminate copies of data from the user-space to the kernel space and vice versa, by carrying out all data transfers within the kernel space. This also eliminates the requirement for expensive and resource consuming context switches. The AF_KTLS module is ideal for scenarios where data does not need

to exist in decrypted form in the user-space. In this case, the encryption, decryption and transfer of data can be done in the kernel space itself. Studies have found that using system calls such as `recv()` and `send()` on the AF_KTLS socket, actually slows down data transfer by 18-20%. This happens because for `send()` and `recv()` calls initiated by the user-space libraries, the data needs to be held in a user-space buffer. The `sendfile(2)` and `splice(2)` system calls are used to make data transmission faster by eliminating user-space interaction. The optimization saves context switches by using Linux kernel module system calls such as `sendfile(2)`. These system calls are able to directly transfer data between two sockets, or between a socket and a kernel-space buffer, known as a `pipe`. This saves unnecessary copies and context switches, thus making the data operations faster. The AF_KTLS module is implemented as a standalone kernel module and can be used on modular Linux kernel architecture. After a handshake is done in the user-space with the help of a user-space library, all the key material is passed to the Linux kernel via appropriate socket system calls supported by AF_KTLS. After that, the user-space application can transparently use AF_KTLS type socket that performs encryption and decryption of traffic. The main advantage of using AF_KTLS lies in applications that serve static content that can be sent using the `sendfile(2)` and `splice(2)` system call or by applications that do not necessarily need decrypted content available in the user-space.

3 Problem Description and Solution Space

3.1 Objective of The Thesis

The goal of this thesis is to analyze possible use-cases and scenarios where AF_KTLS could be beneficial and to propose changes in user-space applications or directly inside the AF_KTLS module implementation that would lead to optimization. This will be followed by implementation of the proposed changes, evaluation of benchmarks on implemented changes and reviews of other optimization alternatives, based on bottlenecks found in benchmarks.

3.2 Solution Identification

After discussions with Red Hat and literature study, HAProxy **[13]** was considered as a viable use-case for integration with the AF_KTLS implementation. HAProxy is a very fast and reliable solution offering high availability, load balancing, and proxy for TCP and HTTP-based applications. According to the HAProxy documentation, it is a single-threaded, event-driven, non-blocking engine combining a very fast I/O layer with a priority-based scheduler. As it is designed with a data forwarding goal in mind, its architecture is optimized to move data as fast as possible with the least possible operations. It can accept a TCP connection from a listening socket, connect to a server and attach these sockets to facilitate traffic flows in both directions. It also acts as a server and receives HTTP requests over connections accepted on the listening TCP socket. It then passes these requests to servers using different connections. TLS may be used on the connection coming from the client and on the connection going to the server. It implements a layered model offering bypass of higher levels mechanisms at each stage. Most of the processing is performed in the kernel and the HAProxy helps the kernel to do the work as fast as possible by giving some hints or by avoiding certain operations when it guesses they could be grouped later. HAProxy only requires the haproxy executable and a configuration file to run. Once the HAProxy is started, it processes the incoming connections, periodically checks the server status and exchanges information with other haproxy nodes.

The HAProxy has support for TLS in order to serve encrypted traffic. In this scenario, the traffic has to be decrypted before it is sent to an output interface. The HAProxy has tried multiple approaches to reduce the overhead required for TLS processing in its newer versions. These approaches include:

- TLS NPN (Next Protocol Negotiation) and ALPN (Application-Layer Protocol Negotiation) extensions to reliably offload SPDY or HTTP2 connections and pass them in clear text to backend servers. ALPN avoids an additional round trip by having the client list the application layer protocols it supports in the `client hello` message. The server chooses a protocol and includes it in the `server hello` message. In NPN the server lists the supported protocols in the `server hello` message and allows the client to choose.

- OCSP (Online Certificate Status Protocol) stapling to reduce first page load time by delivering inline OCSP response when the client requests a Certificate Status Request.

- Dynamic record sizing to achieve high performance and low latency by letting the browser start to fetch new objects while packets are still in flight.

These existing optimizations make the HAProxy a good candidate for further optimization through the AF_KTLS kernel module.

The TCP splicing **[14]** method has already been implemented for the HAProxy to reduce the number of process wake-ups and data copies between user space and kernel buffers. TCP Splicing is a method by which a user-space proxy can communicate to the kernel that the kernel itself can perform the transfers and reduce the overhead of the data transfers. This is particularly advantageous for protocols in which data is sent till the end of the session. HAProxy has been designed with a clear distinction between the headers and the data phase, so the hooks for TCP splicing were added to the project successfully. The Linux Layer7 Switching (L7SW) provides this service to help user-space proxies achieve very high performance. In the current version, the HAProxy consists of a loadable kernel module providing TCP Splicing under Linux. The first step to optimize the HAProxy will be to

integrate this implementation into the HAProxy source code to build it with `splice(2)` support for raw sockets. The `splice(2)` system call moves data between two socket descriptors or between a kernel-space buffer and a socket descriptor, without copying between kernel address space and user address space. With `splice(2)` optimization, we can try to optimize data copy from one interface of the TLS enabled HAProxy to the user space for encryption and then sending it to the socket. This can be done using the `splice()` system call as :

```
splice(int fd_in, off_t *off_in, int fd_out,
off_t *off_out, size_t len, unsigned int flags)
```

3.3 Proposed Modifications

The use case that we are planning to optimize in HAProxy follows the sequence of calls:

- `recv(2)`,
- `encrypt()` or `decrypt()`,
- `send(2)`.

This involves 2 context switches and 2 copies for the data. The proposed version of the HAProxy will try to use the `recv(2)`, `send(2)` and `splice(2)` system calls with the AF_KTLS socket as its source and destination. The user space OpenSSL is used to bind the communicating TCP socket with the AF_KTLS socket and supply all communication configuration based on the handshake . This configuration includes initialization vectors, keys, salt and sequence numbers for `recv(2)` and `send(2)` calls. These credentials are supplied to the AF_KTLS socket by OpenSSL by using the `setsockopt(2)` call. The configuration can be received by user space by using `getsockopt(2)` call. The record assembling and encryption will take place based on the handshake done in OpenSSL.

To support this effort, the implementation requires custom implementation of the aforementioned system calls. The AF_KTLS implementation supports the following calls on the AF_KTLS sockets in the `struct proto_ops` kernel structure:

- `tls_bind()` : Binds the TCP socket to the AF_KTLS socket.
- `tls_setsockopt()`: Sets the TLS credentials for the AF_KTLS socket.
- `tls_getsockopt()`: Receives the communication configuration bback to user-space.
- `tls_sendmsg()`: Sends data from a buffer to the AF_KTLS socket.
- `tls_recvmsg()`: Receives data from the AF_KTLS socket to a buffer.
- `tls_splice_read()`: `splice_read` operation using the AF_KTLS socket as the source.
- `tls_sendpage()`: Supports copy-less writing to the AF_KTLS socket.

The AF_KTLS calls have to be integrated with the HAProxy code for handling TLS connections.

For the implementation of the AF_KTLS in the HAProxy, we are using the `net_next_ktls` **[15]** patches on top of the `net_next` kernel repository for Linux kernel version 4.x. This sets up the environment setup required for AF_KTLS. We are using the Linux kernel TLS/DTLS Socket Tool **[16]** to demonstrate the usage, benchmark and verify the implementation of the AF_KTLS socket. The tool implements a client-server architecture which leverages the AF_KTLS socket for communication. It is using GnuTLS now. The tool benchmarks the number of records sent and received, and the time required to run the scenario. To evaluate the speed impact, files are transmitted with `send(2)` and `recv(2)` calls, `splice(2)` call and `sendfile(2)` call. To verify the intended working of the implementation, there is a test suite **[17]** for AF_KTLS using the Google Test framework. We are currently working on the HAProxy stable version 1.7.8 with the TLS support enabled and the `splice(2)` support integrated into it.

The first step in this approach is to make the AF_KTLS kernel module work with latest Linux kernel 4.x. The environment to generate the kernel module from source needs to be set up first. Changes were required in the AF_KTLS code we initially started working with to

make it compatible with the latest kernel source tree. The changes are as follows:

- The `skb_splice_bits()` function used in the code has to be changed according to the parameters supported by the Linux kernel 4.x source:

```
int skb_splice_bits(struct sk_buff *skb,
struct sock *sk, unsigned int offset,
struct pipe_inode_info *pipe, unsigned int len,
unsigned int flags);
```

- The GnuTLS library has to be set up to allow the usage of 128 bit AES-GCM as the default encryption algorithm.

After installation of the AF_KTLS module into the kernel source tree, the module is tested on this set up by making use of the benchmarks provided for AF_KTLS. If the testing is completed successfully, AF_KTLS is ready to be used by user-space applications for processing data in the kernel. Subsequently, the chosen application, HAProxy with TLS and `splice(2)` support, will be compiled. This application will be tested by studying the generated `syslog` files on our environment. Thereafter, socket communications between HAProxy client and server using TLS protocol will be identified and modified to use AF_KTLS sockets as source and destination.

On completion of the integration of AF_KTLS support for HAProxy, we will evaluate the benchmarks on HAProxy implementation with and without AF_KTLS support. We will compare the results to verify if any optimization is achieved in the latter case. The main parameters for benchmarking are as follows:

- Bandwidth: testing the data rate with different file sizes from small to large.
- Latency: testing the time taken for the first response for smaller file sizes.
- Number of client connections: testing with periodically increasing number of clients connecting concurrently to the servers through HAproxy.

`lighttpd` **[18]** is being used as a web server to which HAProxy routes client requests. Lighttpd is an open-source web server that is fast, secure and flexible for different environments. It has been optimized for environments where speed is a critical factor. For the client side, there are a variety of options available. The initial implementation can be tested with a browser. This will give a clear idea of how the protocol is working, the amount of data that is being successfully transferred after the implementation, and whether the data is being routed through the AF_KTLS socket. For benchmarking and load testing, `siege` **[19]** is being used to simulate client requests to the HAProxy. Siege is used for HTTP and HTTPS load testing and web server benchmarking by simulating a large number of concurrent users accessing one or more URLs repeatedly. After every run siege summarizes the performance measures for the web server that was tested. Performance measures include elapsed time of the test, amount of data transferred, response time, transaction rate, throughput and concurrency. In our case, siege will be used to benchmark the HAProxy load balancer.

4 Proposed Approach and Implementation

In this section we present the design and implementation of the solution. The steps to implement AF_KTLS in HAProxy are as follows:

- Identification of functions in the HAProxy code which are used for socket communication.
- Identification of the specific functions in the HAProxy code which are used for socket communication over TLS protocol.
- Creation of the AF_KTLS socket of the protocol family defined by the AF_KTLS module.
- Binding of the AF_KTLS socket with the AF_INET socket being used for communication.
- Set up of the TLS communication configuration for the AF_KTLS socket using the existing SSL library.
- Handling of TLS control messages, since they cannot be handled by the AF_KTLS module.
- Exchange of data through the AF_KTLS socket using `recv(2)`, `send(2)` and `splice(2)` calls, as defined by the AF_KTLS module. This would entail replacement of SSL user-space library calls by the calls defined by the AF_KTLS module.

4.1 Setting up HAProxy

HAProxy has options for providing a variety of features to the user. These options have to explicitly chosen by the user at compile time so that it can be available at run-time. For SSL support, HAProxy has to be compiled with the option to enable it. Apart from this, `splice(2)` support can also be enabled explicitly.

The HAProxy load balancer is handling SSL connections using the OpenSSL library. Two source files, `ssl_sock.c` and `shctx.c` are compiled into the HAProxy whenever SSL support is requested. The

`ssl_sock.c` code is used to set up the SSL connection, decrypt and forward client requests and encrypt response from servers. The `SSL_CTX` SSL contexts are created and allocated for servers in this file, the certificate chain is created and used, and private keys are generated. The handshake takes place here using the `SSL_do_handshake` function. The `shctx.c` code is used for maintaining the shared SSL context. The functions of interest for this thesis are as follows:

- `ssl_sock_init()`: This function is used for initializing the SSL socket over which communication will take place. This is designed to be called before any other data-layer operation. This function is responsible for setting up the SSL context, identifying whether HAProxy is in client mode or server mode, and setting the handshake flag on the connection.

- `ssl_sock_handshake()`: This is the callback which is used when an SSL handshake is pending.

- `ssl_sock_to_buf()`: This function is used for receiving data from the SSL connection socket.

- `ssl_sock_from_buf()`: This function is used for sending data to the SSL connection socket.

In this configuration, HAProxy has been configured to use SSL termination. This means that the HAProxy load balancer accepts encrypted requests from the client over HTTPS, decrypts the request and send the unencrypted request to the server. In the reverse direction, the HAProxy load balancer accepts unencrypted response from the server, encrypts it, and passes it on to the client. One of the reasons for using SSL termination is that it offers a centralized place to prevent or detect SSL attacks. SSL termination at a variety of web servers, which are out of the control of an organization might lead to additional complexity as well as possible security vulnerability.

The HAProxy load balancer maintains two TCP sockets for communication. One socket is used for accepting client requests and stays open in the listening mode. The other socket is used for connecting to web servers to extract responses for the client requests. The requests from the clients and the responses from the web servers are

passed between these two sockets using either a buffer or a pipe. When the HAProxy is used with SSL termination, the communication between the client and the HAProxy occurs over the first socket using the TLS protocol. This is the socket that has to be bound to the AF_KTLS socket that is created. This communication is defined in the `ssl_sock_from_buf()` function.

The HAProxy configuration has to be provided externally to run the binary. The configuration file contains 3 major sources of parameters:

- Command line arguments on priority basis.
- Global section which sets process-wide parameters.
- Proxies section containing the defaults, frontend and backend. This section lists out the SSL certificate, available web servers, HTTP forward requests, redirection requests, amongst others.

This file is used to specify the use of SSL, enabling and disablement of splice and additionally it also lists out the default cipher suite and TLS version. In our case, the cipher suite used is ECDHE-RSA-AES-128-GCM-SHA256 and the TLS version used is TLSv1.2. These parameters have to specified because the AF_KTLS module supports only TLSv1.2 and AES 128-bit ciphers in the GCM mode.

4.2 Setting up AF_KTLS

The AF_KTLS module was compiled on Linux kernel version 4.9-rc7. The RFC5288 patches also needed to be rewritten to be usable with the newer vanilla kernel versions. The changes mentioned in the previous chapter, i.e, redefinition of `skb_splice_bits()` function and set up of GnuTLS default cipher as AES-128-GCM, were necessary to make it work with the newer kernels. Protocol families in the Linux kernel have to provide the `struct net_proto_family` constant that is used for instantiation of the socket during socket creation method. The `socket(2)` system call requires the definition of domain, type and protocol. After this, the `bind(2)` system call is used by user-space to bind a socket. The protocol family constant for AF_KTLS is explicitly defined in the HAProxy code as:

```
#define AF_KTLS 12
```

The custom structure definition for `bind(2)` is implemented as:

```
struct sockaddr_ktls {
    __u16 sa_cipher;
    __u16 sa_socket;
    __u16 sa_version;
};
```

4.3 Integration of AF_KTLS with HAProxy

The file `ssl_sock.c` has been mainly used for the implementation of AF_KTLS in HAProxy. The function `ssl_sock_init()` sets the SSL context and the method. This sets the TLS version and cipher list to be used by the new context. This has to be set in the HAProxy code as:

```
SSL_CTX_set_ssl_version(objt_listener(conn->target)
->bind_conf->default_ctx, TLSv1_2_method());

SSL_CTX_set_options(objt_listener(conn->target)
->bind_conf->default_ctx, SSL_OP_NO_SSLv2);

SSL_CTX_set_cipher_list(objt_listener(conn->target)
->bind_conf->default_ctx,
"ECDHE-RSA-AES128-GCM-SHA256");
```

This function also sets the state of HAProxy as either connecting or accepting. HAProxy is in the accepting state when it receives requests from clients and in the connecting state for extraction of responses from the web servers.

The SSL handshake is done in the `ssl_sock_handshake` function of `ssl_sock.c` file of HAProxy. In the SSL termination configuration, the handshake is carried out only between the client and the HAProxy. This is done over regular AF_INET sockets on both sides. This uses the primitives provided by the OpenSSL library such as `SSL_do_handshake` and `SSL_peek()`.

One of the main challenges of the implementation was that the The HAProxy load balancer operates in the non-blocking mode. By

default, TCP sockets are in the blocking mode. This means that the process waits for data to appear and does not return control back to the program until the operation is complete. HAProxy sets its socket descriptors as non-blocking using the `fcntl` API. When placed in non-blocking mode, the program cannot wait for an operation to complete. This is important if it switches between many different connected sockets which have a potential to block the program. If the system call recv() is used in non-blocking mode, it will return any data that the system has in it's read buffer for that socket. If the read buffer is empty, the system will return with an "Operation Would Block!" error. The send() system call also behaves similarly. It puts the data into a buffer, and removed from the buffer upon read. If the buffer gets full, the system returns the error 'Operation Would Block" on next write.

For an accept() system call, if no clients are ready to connect, it returns with an "Operation Would Block" error. The connect() system call behaves in a different manner. If a `connect()` system call is not successful, then it returns with an "Operation In Progress" error. In the non-blocking mode, `SSL_do_handshake()` will return with error if the requirements to continue the handshake are not satisfied. `SSL_get_error()` can be used on the return value of `SSL_do_handshake()` to determine the type of error which occurred during the handshake. It might be either `SSL_ERROR_WANT_READ` or `SSL_ERROR_WANT_WRITE`. The former means that the socket is not ready for reading yet, and the latter means that the socket is not ready for writing yet. The calling process has to repeat the call after taking appropriate action to satisfy the requirements of `SSL_do_handshake()`. HAProxy uses `epoll()` and `select()` system calls to check for the required conditions after every iteration of an unsuccessful handshake.

Once the AF_KTLS socket is initialized, then the underlying TCP socket cannot be used in a regular way. The attempt to handle any control messages will result in the AF_KTLS socket returning the error `EBADMSG`. For all control operations, e.g, `SSL_handshake_renegotiation` in the event where handshake fails to complete and returns with an error, the control needs to be passed back to the OpenSSL library for continuation of control operations. The state of the AF_KTLS socket parameters has to be saved and the control has to pass back to the socket when the program enters the data plane after completion of control operations.

The AF_KTLS socket is only used for the symmetric encryption/decryption of every record which carries actual data. The key concept of optimizing HAProxy is to move TLS/DTLS record assembling, disassembling and symmetric encryption, decryption of the data records into the kernel space.

The requests from the client is received by HAProxy in `accept()` mode using the `SSL_read()` call in the function `ssl_sock_to_buf()` as:

```
static int ssl_sock_to_buf(struct connection *conn,
struct buffer *buf, int count){
ret = SSL_read(conn->xprt_ctx, bi_end(buf), try);
}
```

This function handles the receipt of requests in non-blocking mode. After receiving the encrypted client request, HAProxy decrypts it and sends it to the sockets connecting to the web servers. Once it receives the response, it encrypts it and writes it back into the client-HAProxy socket using `SSL_write()` in the `ssl_sock_from_buf()` function as:

```
static int ssl_sock_from_buf(struct connection *conn,
struct buffer *buf, int flags){
ret = SSL_write(conn->xprt_ctx, bo_ptr(buf), try);
}
```

In the implementation, this socket is bound with the AF_KTLS socket and the OpenSSL calls are replaced with ordinary system calls which are further defined by the AF_KTLS module. To summarize, the TLS connection is done over AF_KTLS socket using the following scheme:

- Call `accept()` on an `AF_INET` socket, which is a TCP socket.
- Use OpenSSL to initiate a handshake between client and HAProxy by using the `SSL_do_handshake` call.
- Create an `AF_KTLS` socket descriptor and initialize the `sockaddr_ktls` structure which binds the `AF_KTLS` socket with the original TCP socket.
- Extract the TLS Initialization Vectors (IVs), session keys, and sequence IDs from the OpenSSL library. Use `setsockopt` on the `AF_KTLS` socket descriptor to pass them to the Linux kernel.

- Use standard read(), write(), and splice() system calls on the AF_KTLS socket descriptor.
- Transfer control to original TCP socket if TLS control messages arrive, and transfer it back to the AF_KTLS socket by calling getsockopt().

We create the AF_KTLS socket as follows:

```
struct sockaddr_ktls sa = { .sa_cipher =
KTLS_CIPHER_AES_GCM_128, .sa_socket = origfd,
.sa_version = KTLS_VERSION_1_2};

int ktls_sock = socket(AF_KTLS, SOCK_STREAM, 0);
bind(ktls_sock, (struct sockaddr *) &sa,
sizeof(sa)
```

This creates the AF_KTLS socket and returns the socket descriptor ktls_sock for use by the HAProxy code. The origfd is the socket descriptor of the orginal TCP socket which has been bound to the AF_KTLS socket for data operations. We can manipulate options for the AF_KTLS socket using the calls getsockopt() and setsockopt(). The setsockopt() calls are used to set the send key, receive key, send IV and receive IV, send nonce and receive nonce, and sequence numbers for the AF_KTLS socket as:

```
setsockopt(ktls_sock,AF_KTLS,KTLS_SET_KEY_SEND,
writeKey, 16)
setsockopt(ktls_sock,AF_KTLS,KTLS_SET_KEY_RECV,
readKey, 16)
setsockopt(ktls_sock,AF_KTLS,KTLS_SET_SALT_SEND,
writeIV, 4)
setsockopt(ktls_sock,AF_KTLS,KTLS_SET_SALT_RECV,
readIV, 4)
setsockopt(ktls_sock,AF_KTLS,KTLS_SET_IV_SEND,
writeSeqNum, 8)
setsockopt(ktls_sock,AF_KTLS,KTLS_SET_IV_RECV,
readSeqNum, 8)
```

After setting the TLS communication configuration of the AF_KTLS socket, the socket is now ready for use for data operations. Since

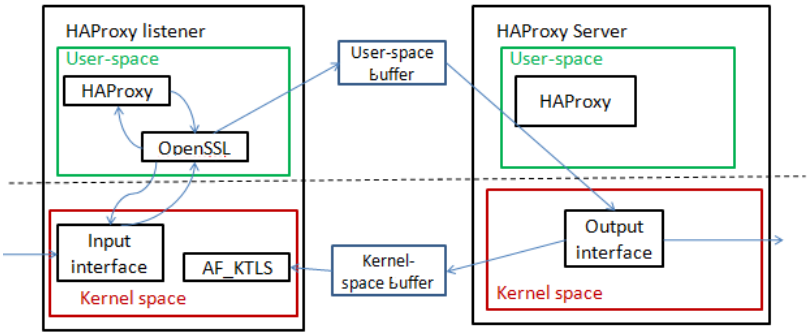

Figure 4.1: Diagram of the transmission in implemented integration.

HAProxy operates in the non-blocking mode, every time the SSL handshake returns an error, or the data read/write return with "Operation would be blocked" error, or the handshake needs to be renegotiated, the control has to be passed back to OpenSSL using the following function:

```
getsockopt(ktls_sock,AF_KTLS,KTLS_GET_IV_RECV,
readSeqNum, &optlen);
getsockopt(ktls_sock,AF_KTLS,KTLS_GET_IV_SEND,
writeSeqNum, &optlen);
close(ktls_sock);
```

The IV list is stored sequentially for the AF_KTLS socket, hence the next IV can be set using sequence numbers.

A diagrammatic representation of the data transmission in the implemented AF_KTLS socket in the HAProxy can be seen in Figure 4.1. Here, it can be seen that HAProxy consists of two sockets: one for listening to and accepting client connections, and the other socket is for connecting to the web servers. The client requests come to the TCP socket over OpenSSL and they are transferred to the TCP socket for HAProxy server mode. The responses from the server arrive at the HAProxy server socket and they are written back to the newly

created AF_KTLS socket. The AF_KTLS socket encrypts and sends this response back to the requesting client. Without the use of the `splice(2)` system call, this implementation replaces the `SSL_write()` OpenSSL function with:

```
write(ktls_sock,input_buffer,TLS_PAYLOAD_MAX_LEN)
```

The `TLS_PAYLOAD_MAX_LEN` is the maximum transfer unit (MTU) of the AF_KTLS socket. In the above case, the data from the client-side HAProxy TCP socket is sent over SSL to the server-side HAProxy TCP socket over OpenSSL. The responses from the servers are sent to the newly created AF_KTLS socket, that is bound to the client-side HAProxy TCP socket.

Now the data transfer operation is ready to be optimized using the `splice(2)` system call as defined by the AF_KTLS module. HAProxy has a provision for using the `splice()` system call for raw sockets that are not operated over the TLS protocol. These functions are defined as `raw_sock_to_pipe()` and `raw_sock_from_pipe()` in the source file `raw_sock.c`. Analogously, we have defined functions `ssl_sock_to_pipe()` and `ssl_sock_from_pipe()` for data transfer over the TLS protocol. In the function `ssl_sock_from_pipe()`, two `splice(2)` system calls have been used. The first one uses the server-side HAProxy TCP socket as source and writes to a kernel-space buffer, and the second one uses the kernel-space buffer as source and writes to the AF_KTLS socket as:

```
splice(server_fd, NULL, pipe[1], NULL, payload,0)
splice(pipe[0], NULL, ktls_sock, NULL, payload,0)
```

The `pipe()` system call creates an array `pipefd` that returns two file descriptors `pipe[0]` and `pipe[1]`. They are used for reading and writing, respectively.

Not all socket descriptors support the `splice(2)` system call. For our implementation, we are considering the sockets to be either the data source or data destination. This means that there has to be a `splice_read` operation and a `sendpage` operation for both scenarios. The AF_KTLS implementation of `splice_read` call is known as `tls_splice_read`. This operation is invoked from the HAProxy code when `splice(2)` is called.

The prototype of the operation is as follows:

```
static ssize_t tls_splice_read( struct socket
*sock, loff_t *ppos, struct pipe_inode_info *pipe,
size_t len, unsigned int flags)
```

To support copy-less writing to the AF_KTLS socket, `tls_sendpage` operation defined by the AF_KTLS module has also been integrated. The prototype of the implemented function is as follows:

```
static ssize_t tls_sendpage(struct socket *sock,
struct page *page, int offset, size_t size,
int flags)
```

Once the AF_KTLS socket is in use, then the TCP socket it is bound with cannot be used by user-space, unless the control is passed back to the user-space library. After the implementation was completed, it was time to test it using a client-server environment. The next section describes the setup used for experimentation for the verification of the implementation.

5 Experimental Setup

The implementation was tested and benchmarked using the following set-up:

- 3 Lighttpd web servers configured to serve static files of periodically increasing sizes. The files are filled with random data.
- The HAProxy load balancer, which serves the files from either of the three lighttpd servers. HAProxy has been configured to provide SSL Termination **[20]** with SSL-only redirection. In this configuration, the load balancer is tasked with terminating/decrypting an SSL connection, and sending unencrypted connections to the configured lighttpd servers. For fulfilling this requirement we created a self-signed certificate `server.pem`. The SSL connection is handled and terminated by HAProxy and hence unencrypted HTTP requests can be sent to the backend lighttpd servers. For the SSL-only redirection, we added the redirect directive, which redirects `http` to `https` for connections which are not made over the TLS protocol. The HAProxy `ssl_fc` variable is used to create this conditional configuration. Firewall rules also need to be added to the clients, servers and HAProxy to allow clients to connect to the HAProxy load balancer, and to allow HAProxy to connect to the lighttpd web servers.
- A client machine running the siege tool that has been configured to simulate periodically increasing number of concurrent clients connecting to the load balancer HAProxy for getting the different files that are being served by the lighttpd server.

The HAProxy load balancer communicates with the backend lighttpd servers and conducts the file transfers by putting up a frontend for the clients, which are then served a file from one of the 3 servers.

One of the aims was to ensure minimal interference during the benchmarking process to provide replicated environment for all the benchmarks. The machines were connected in an internal network with no access from the outside. All CPU intensive processes such as rendering of X environment, file transfers, file creations, multimedia applications were restricted from the machines of this configuration.

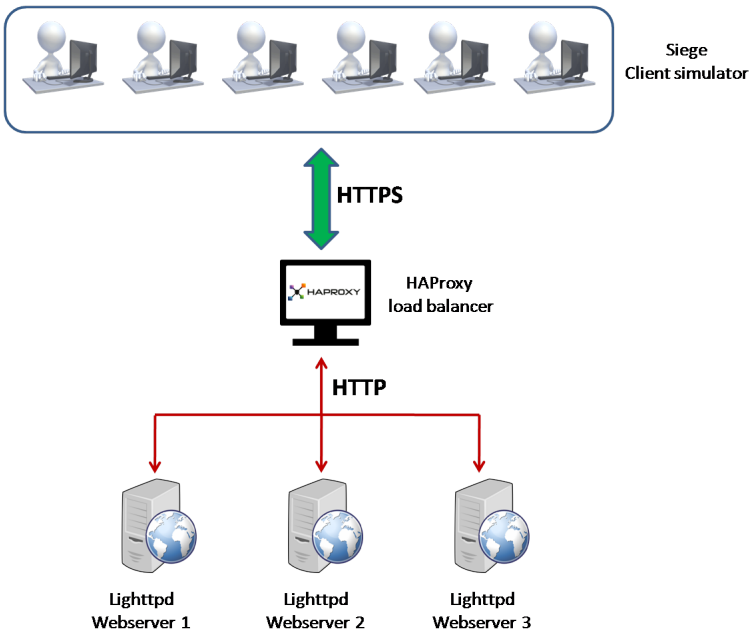

Figure 5.1: Diagram of the experimental setup.

This ensured a clean and easily clone-able environment for all subsequent benchmarking activities.

The benchmarking utility provided by the siege tool have been used for generating human-readable benchmarks. The performance measures provided by the tool include elapsed time, first response time, throughput, number of successful transactions, concurrency percentage, amongst others. The benchmarks have been carried out for the following cases:

Case I. HAProxy with no AF_KTLS.

Case II. HAProxy with AF_KTLS and no SSL splice.

Case III. HAProxy with AF_KTLS and SSL splice.

The splice over raw TCP sockets has been integrated implicitly into the HAProxy configuration. The schematic representation of the experimental setup is depicted in Figure 5.1.

Bash scripts on the lighttpd server, the siege client, and the HAProxy load balancer are used to set up the environment required for minimum interference benchmarking. The parameters for benchmarking each case is as follows:

- Bandwidth: The random data files that are being served by the lighttpd servers to the siege clients through the HAProxy load balancer are of sizes ranging from `100 MB` to `1 GB`. This is measured over the range of file sizes for a single client connection.
- Latency: This is measured for all the cases with the same file size, `100 MB`, and a single client connection.
- Concurrency: The scripts are altering the configuration of siege to incrementally change the number of concurrent client connections. This is measured for the same file size, `100 MB`.

The timing metadata for completion of each client-server session through HAProxy, i.e, the success of file transfer from server to client, is recorded by the scripts for later analysis.

The benchmarking for Case I was initially tried out on HP Compaq desktop PC with Intel Core 2. The results were as expected for Case I, but Case II could not be tried out with the same configuration because the processor did not support AESNI optimization. AF_KTLS relies heavily on AESNI optimization by the processor, hence we had to use a different machine for all the benchmarks. The final benchmarking for all the cases has been carried out on a Dell PC with Intel i5 processor that supports AESNI optimization. The OpenSSL version on both the client machine and the HAProxy load balancer is 1.0.2k.

The results from the benchmarking have been plotted in the next section so that we can check for possible optimizations and bottlenecks.

6 Experimental Results and Analysis

The results have been compiled in Section 5.1 according to the three cases defined in the previous chapter, i.e, HAProxy with no AF_KTLS support, HAProxy with AF_KTLS support, and HAProxy with AF_KTLS with splice support. Comparative analysis of the obtained results is provided in Section 5.2.

6.1 Results

An observation was made while benchmarking Case III; the response times for larger files were too high to be usable. The optimization effect of using the `splice()` system call with AF_KTLS on the HAProxy load balancer could only be observed while transferring smaller files. The results measuring bandwidth for the first two cases with large file sizes are depicted below:

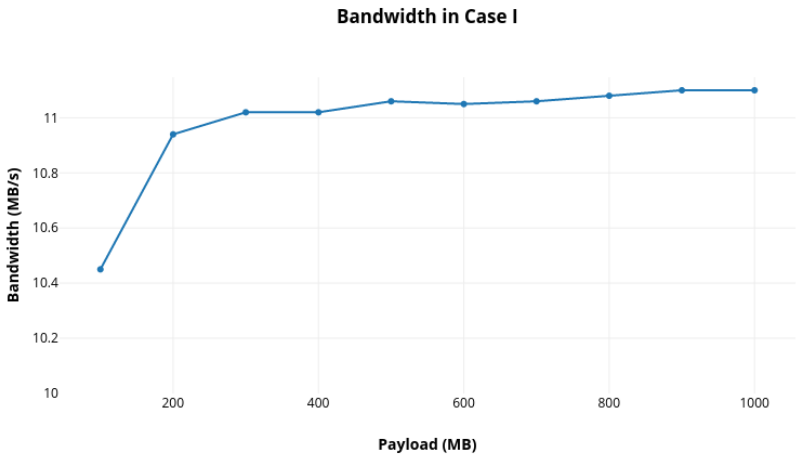

Figure 6.1: Bandwidth measurement for Case I.

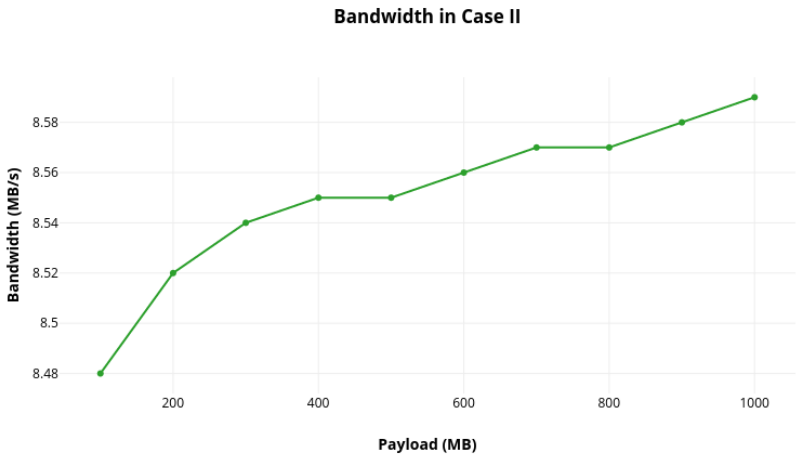

Figure 6.2: Bandwidth measurement for Case II.

The results measuring the bandwidth of the three cases for small file sizes are depicted below:

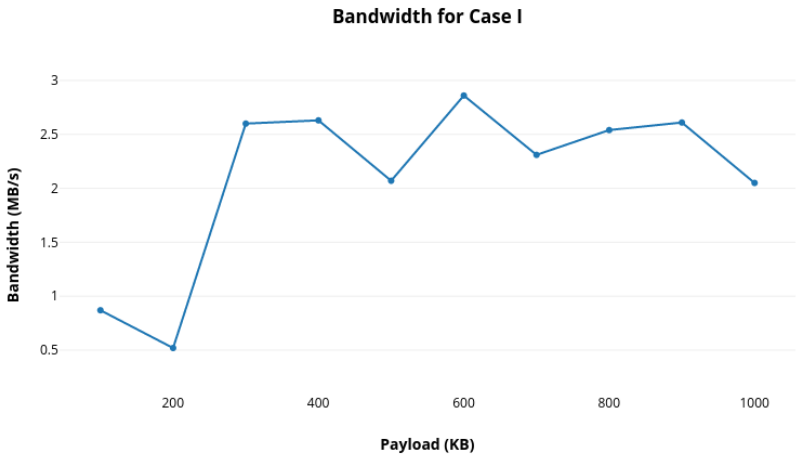

Figure 6.3: Bandwidth measurement for Case I.

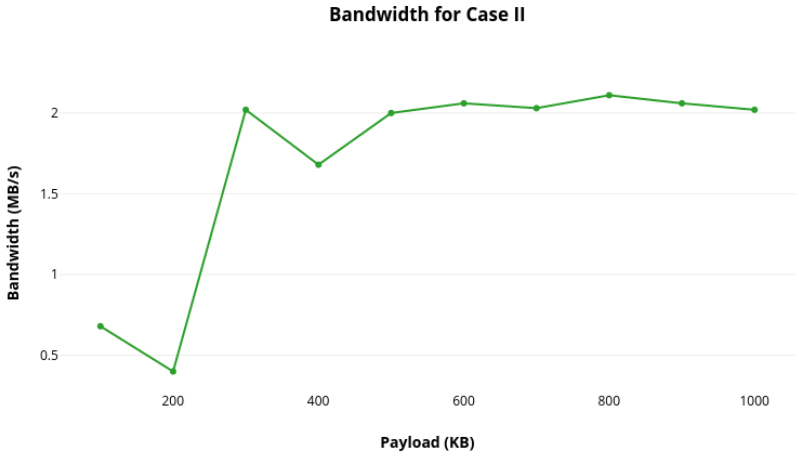

Figure 6.4: Bandwidth measurement for Case II.

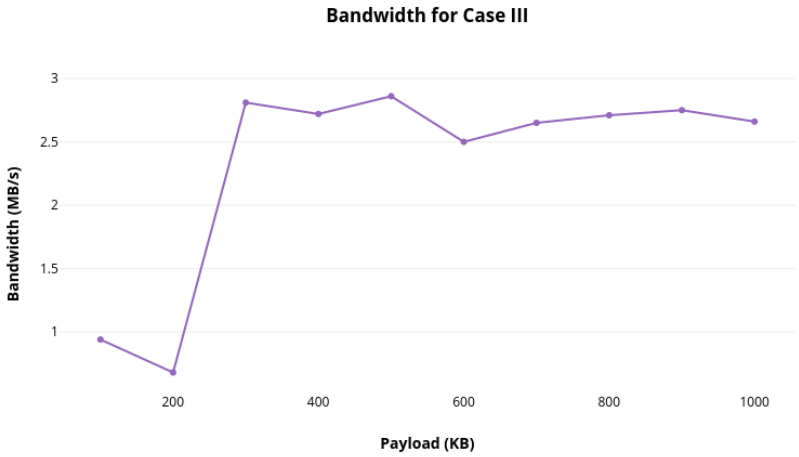

Figure 6.5: Bandwidth measurement for Case III.

These results show that the behavior of HAProxy for all three cases is similar but the values of the performance are different. The comparisons between the three sets of values will be made in the next section.

The results measuring latency for the three cases are depicted below:

Table 6.1: Response times for all the cases (100 MB).

Cases	HAProxy	HAProxy-AF_KTLS	HAProxy-AF_KTLS-splice
Latency(s)	8.60	10.50	95.50

These values have been measured for the same file of size `100 MB`, for 10 iterations. The average of the 10 obtained values have been taken as the average latency result. Here, it can be seen that the latency for the third case is significantly higher than the other two cases for the file of size 100MB.

The results measuring latency for the three cases for small file sizes are depicted below:

Table 6.2: Response times for all the cases (100 KB).

Cases	HAProxy	HAProxy-AF_KTLS	HAProxy-AF_KTLS-splice
Latency(s)	0.07	0.085	0.064

These values have been measured for the same file of size `100 KB`, for 10 iterations. The average of the 10 obtained values have been taken as the average latency result.

In this result, we can see a similar behavior to the results of bandwidth measurements for small file sizes for the three cases.

The results measuring the throughput for increasing number of concurrent accesses for all three cases are depicted below:

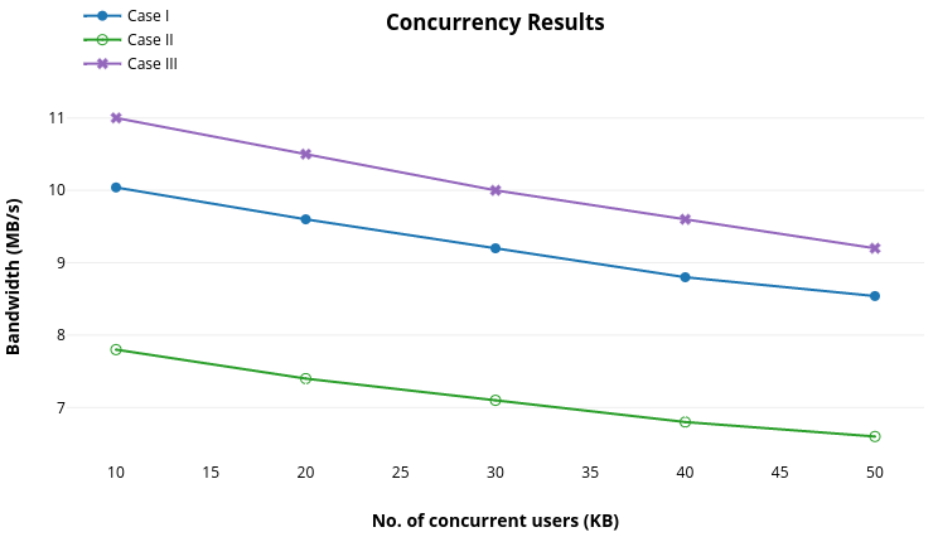

Figure 6.6: Throughput for concurrent access.

These values have been measured for the same file of size `10 MB`, for increasing number of concurrent client connections in the range 10 to 50. We can see that throughput decreases with increase in number of concurrent connections, for all cases.

Similar to the observations made on the bandwidth measurement, here it was observed that the response times for larger files were too high to be usable in Case III. The optimization effect of using the `splice()` system call with AF_KTLS on the HAProxy load balancer could only be observed while transferring smaller files. The comparative analysis is done in the next section.

6.2 Analysis of Results

In this section we compare and analyze the results of the experiments.

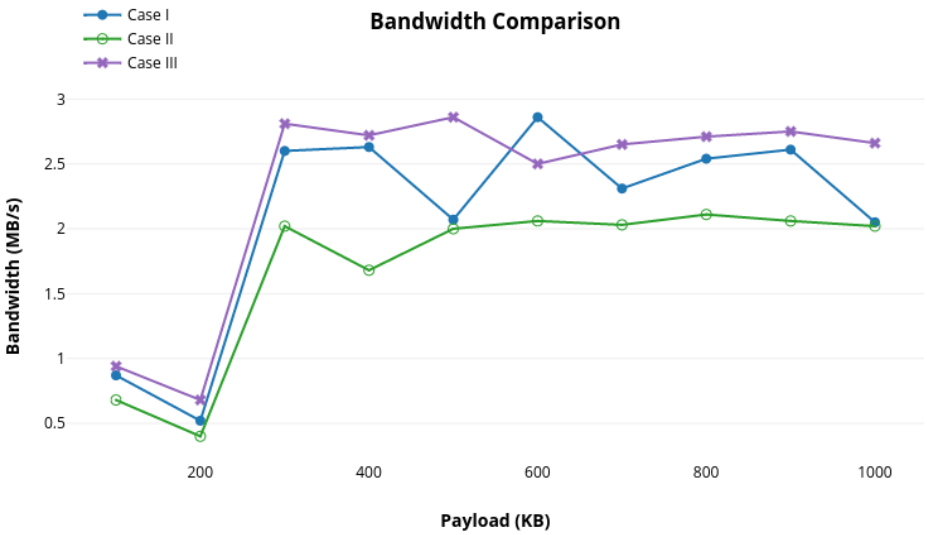

Figure 6.7: Bandwidth comparison for all the cases.

Figure 6.7 shows the comparison for bandwidths in the three cases for different file sizes. Here we can see that using AF_KTLS with simple `recv()` and `send()` calls on HAProxy is slower than using the default `SSL_read()` and `SSL_write()` calls. The Case II is approximately 20-22% slower than the Case I. This result verifies the claims by the AF_KTLS developers, that data transfer over AF_KTLS without splice is slower than that by user-space libraries due to certain bottlenecks which have been discussed and analyzed extensively by the developers [] . In Case III, where AF_KTLS is used with the `tls_read_splice()` call, the operation is much faster for small file sizes. It is seen to be nearly 9-10% faster than Case I.

From the Table 6.2, we can see the same results for latency of the data transfer. In this case also, HAProxy with AF_KTLS is slower than OpenSSL implementation. The Case III, that is HAProxy with

AF_KTLS and splice improves the latency of data transfer for small file sizes.

From the Figure 6.6, we can see similar comparisons between the three cases. Throughput behavior for the three cases follows the same pattern in case of concurrency as well.

This is in accordance with the results of benchmarking of AF_KTLS by the developers. This proves that implementation of data transfer over AF_KTLS socket using splice in HAProxy improves and optimizes the performance of the HAProxy load balancer by at least 9-10% for small file sizes.

6.2.1 Bottlenecks and Limitations

It has been observed that Case III shows significantly worse performance for file sizes larger than 10 MB. A possible reason for this observation could be that the HAProxy load balancer operates in the non-blocking mode. This leads to frequent handshake renegotiations and data read and write errors when resources are unavailable or locked up. The HAProxy load balancer keeps polling for data from the servers using the `epoll()` system call. The connections to the servers are opened in non-blocking mode and every time a server is ready to transfer data, SSL handshake takes place between the HAProxy and the corresponding server. For every connection, renegotiation, and error, the control needs to be passed to OpenSSL and AF_KTLS socket needs to be unattached from the bound TCP socket. Consequently, every time the problems are resolved and control passes back to AF_KTLS socket, the socket has to be bound again with the TCP socket and the TLS communication configuration, such as IVs, keys and sequence numbers, need to be maintained and updated. This occurs quite frequently in HAProxy and adds an increasing overhead to the data operations for large file sizes.

According to the AF_KTLS developers, AF_KTLS socket with `recv()` and `send()` system calls are 18-20% slower than user-space SSL data operations. In our case, HAProxy with AF_KTLS implementation without splice is approximately 20-22% slower.

For the purpose of this thesis, the data coming from the server-side TCP socket has been optimized with AF_KTLS as the destination. The implementation could be faster if the client requests coming from

the client-side TCP socket can be optimized as well with AF_KTLS as the source. Both configurations have to be over the `splice()` call. In this case, the AF_KTLS socket will bind the listening TCP socket before accepting client requests. The client requests will be accepted by the configured AF_KTLS socket and they will be passed on to the kernel buffer using the `splice()` system call, before being passed on to the HAProxy server-side TCP socket. This will lead to optimization using AF_KTLS as the source and the destination, in both sides of the communication between the clients and the HAProxy load balancer.

7 Conclusions and Future Work

This thesis provides an analysis of the AF_KTLS module and identifies possible use-cases which can be optimized using this module. The main advantage of using AF_KTLS is in applications that serve static content that can be sent using the `splice()` system call or by applications that do not necessarily need decrypted content available in user-space. The identified application that fulfilled this criteria is the HAProxy load balancer. HAProxy was used as the user-space application to be optimized using AF_KTLS.

The scheme for TLS communications in HAProxy were identified and analyzed to come up with a solution design that would lead to possible optimization using the AF_KTLS module. The changes identified by the design were implemented in AF_KTLS module and the HAProxy load balancer. The implementation was evaluated and benchmarked.

We found that using HAProxy with AF_KTLS over user-space buffers slowed down the data operations by approximately 22%. This was around 2% higher than the analysis done by the AF_KTLS developers. The use of kernel-space buffers with AF_KTLS in HAProxy showed about 10% improvement in performance for small file sizes. This value was about 1% lower than the analysis done by the AF_KTLS developers. Further investigation revealed a bottleneck which was unavoidable due to the non-blocking nature of the HAProxy application.

Considering the benchmarks it was found that in our case, the implemented changes in the HAProxy application outperforms the available file transfer mechanism over TLS in HAProxy for small file sizes. Link to implementation is `https://github.com/459203/MS_Thesis`.

The HAProxy configuration used for this thesis is using SSL termination. For a more secure implementation, the HAProxy load balancer should terminate the SSL connections coming from the client, and re-initiate another SSL connection for sending and receiving data from the server. In this scenario, HAProxy receives encrypted client requests, decrypts them and re-encrypts them before sending the response to the server over a new SSL connection. The same logic is used in the reverse direction too. In this case, AF_KTLS socket communication

can be used on both client and server sides of HAProxy. This would lead to more security and higher optimization in HAProxy.

Bibliography

[1] Membrey P, Hows D, Plugge E. *SSL Load Balancing*, In Practical Load Balancing, Apress, 2012. URL:

[2] Dierks T. *The transport layer security (TLS) protocol version 1.2.*, RFC 5246, 2008.

[3] Rescorla E, Modadugu N. *Datagram transport layer security version 1.2.*, RFC 6347, 2012.

[4] Mavrogiannopoulos N, Josefsson S. *GnuTLS: The GnuTLS transport layer security library.*, 2014. [Online]. URL: https://www.gnutls.org.

[5] Young EA, Hudson TJ, Engelschall R. *Openssl: The open source toolkit for ssl/tls.*, 2011. [Online]. URL: http://www.openssl.org.

[6] Watson D. *Kernel TLS (Transport Layer Security) Socket*, In NETDEV 1.2, Tokyo, Japan, 2016. URL: https://netdevconf.org/1.2/papers/ktls.pdf.

[7] Randall S, Gurney J.M, Long S. *Optimizing TLS for High–Bandwidth Applications in FreeBSD.*, In Proceedings of AsiaBSDCon Conference, 2015. URL: https://people. freebsd. org/rrs/asiabsd_2015_tls.

[8] Moellenkamp J. *Less known Solaris Features: kssl*, 2009. [Online]. URL: http://www.c0t0d0s0.org/archives/5575-Less-known-Solaris-Features-kssl.html.

[9] Watson D. *Crypto kernel TLS socket*, 2015. [Online]. URL: https://lwn.net/Articles/665602.

[10] Herbert T. *Kernel connection multiplexer*, 2015. [Online]. URL: https://www.kernel.org/doc/Documentation/networking/kcm.txt.

[11] Struk T. *Linux Kernel Patch: crypto: af_alg: add TLS type encryption*, 2016. [Online]. URL: https://lwn.net/Articles/410536.

[12] Pokorný F. *Linux VPN Performance and Optimization*. Master's thesis, Brno University of Technology, Faculty of Information Technology, 2016.

[13] Tarreau W. *HAProxy-the reliable, high-performance TCP/HTTP load balancer*, 2012. [Online]. URL: http://www.haproxy.org.

[14] Tarreau W. *Using Linux TCP Splicing with HAProxy*, 2007. [Online]. URL: http://www.haproxy.org/download/1.3/doc/tcp-splicing.txt

[15] Watson D. *KTLS patches on top of net_next kernel repository*. [Online]. URL: https://github.com/ktls/net_next_ktls.

[16] Pokorný F. *Linux Kernel TLS/DTLS Module Tool*. [Online]. URL: https://github.com/ktls/af_ktls-tool.

[17] Lancerchao. *Tests suite for af_ktls using Google Test*. [Online]. URL: https://github.com/ktls/af_ktls-test.

[18] Bogus A. *Lighttpd*, Packt Publishing Ltd, 2008.

[19] Fulmer J. *Siege*, 2012. [Online]. URL: https://www.joedog.org/siege-home.

[20] Bagepalli N, Patra A. *Load balancing approach for scaling secure sockets layer performance*, In U.S. Patent No. 7,111,162. Washington, DC: U.S. Patent and Trademark Office, 2006.