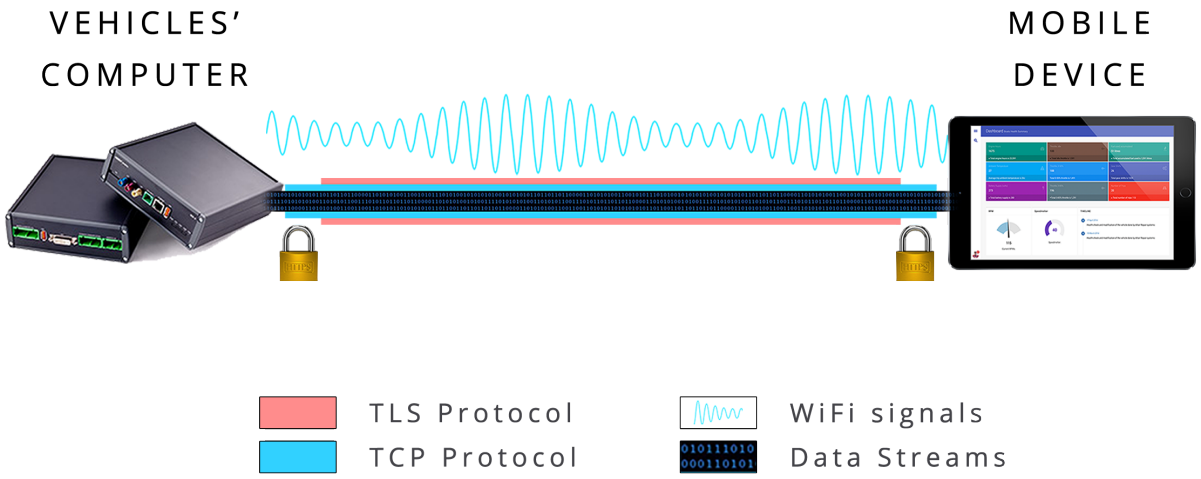
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



The Performance of TLS Protocol in Vehicular Embedded Computers.

Master of Science Thesis in Computer Science

JACKSON ISACK MREMA

Department of Computer science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS INSTITUTE OF TECHNOLOGY
Gothenburg, Sweden June 2016

Master's thesis 2016:NN

The Performance of TLS Protocol In Vehicular Embedded Computers.

A report for the thesis work done at Diadrom Systems AB

JACKSON ISACK MREMA

UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
University of Gothenburg
Chalmers University of Technology
Gothenburg, Sweden June 2016

Industrial Supervisor: Dr. Henrik Fagrell, Diadrom Systems AB
Department Supervisor: Prof. Carlo Furia, Computer Science and Engineering
Examiner: Prof. Alejandro Russo, Computer Science and Engineering

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg Sweden
Telephone + 46 (0)31-772 1000

Cover: Architectural setup of the proposed project. Vehicles ECU wirelessly communicating with mobile device; Transporting HTTP data secured with TLS protocol.

Typeset in LaTeX
Printed by [Name of printing company]
Gothenburg, Sweden June 2016

Abstract

Transport Layer Security (TLS) is a protocol that is widely used to secure Hypertext Transfer Protocol (HTTP) data transported across the Internet. This protocol comes with a set of operations that are for encryption, decryption, sharing keys for encryption etc. Studies have shown that these operations often cause performance degradation. So to implement TLS protocol in an environment with performance limitations (in terms of response time and usability), it becomes necessary to have an idea of performance implication of TLS protocol in that specific environment.

This thesis work has contributed to understanding the performance of TLS in the specific setting of vehicles' embedded computers. Different performance tests were done to understand performance differences between implementations using the TLS protocol and those not using it. Furthermore, this thesis gave an idea about how different TLS parts in the HTTP communication stack perform. Lastly, tests were conducted to understand how different encryption schemes and algorithms supported by TLS protocol perform in this specific setting.

Results of this work demonstrated that TLS protocol induces some delays in response time that remarkably could affect usability of the system. This delay is due to TLS operations that are necessary for agreeing on sets of keys, algorithms, ciphers and protocols to be used when communicating with TLS protocol. Delay in these operations is caused by a number of factors; some found by this thesis work are size of encryption keys, type of cipher and algorithm used and size of the payload to be operated on. Based on these results, there are some recommendations in this work that when followed would help to optimally utilize the power of TLS protocol. Besides these findings, an open-source software program that performs all of the performance tests explained in this thesis work was produced. This program will potentially allow others to reproduce this research in their specific environments.

Keywords: Security, Embedded Computing, Communication, TLS Protocol.

Acknowledgements

I would like to extend my sincere gratitude to my industrial supervisor Dr. Henrik Fagrell and the entire team at Diadrom Systems AB for the constant support and help that they offered during my thesis work. I would also like to thank my university supervisor Prof. Carlo Furia for the genuine support and continuous inputs that he gave for this thesis. Lastly I would like to thank my family and my girlfriend for the encouragement they provided throughout the thesis.

Jackson Isack Mrema, Gothenburg, June 2016

Contents

List of Figures

List of Tables

List of Abbreviations

AES - Advanced Encryption Standard
CBC - Code Block Chaining
CFB - Cipher Feedback Mode
DES - Data Encryption Standard
DSA - Digital Signature Algorithm
ECDH - Elliptic curve Diffie–Hellman
ECDSA - Elliptic Curve Digital Signature Algorithm
HMAC - Hash-based Message Authentication Code
HTTP - Hypertext Transfer Protocol
HTTPS - Secured Hypertext Transfer Protocol
IETF - Internet Engineering Task Force
MD - Message Digest
OSI - Open Systems Interconnection Model
PKI - Public Key Infrastructure
RC - Rivest Cipher
RSA - Rivest, Shamir, and Adleman
SHA - Secure Hash Algorithm
TCP - Transport Layer Security
VEC - Vehicular Embedded Computers
WiFi / Wi-Fi - Wireless Fidelity
WPA - Wi-Fi Protected Access

1
Introduction

Today, automotive products eg. cars, boats and trucks contain a lot of different features that can be controlled from a primitive display that is a part of a vehicular embedded system. When this system is running, these features consume or produce data and parameters that are often used for different purposes, for example diagnostic purposes, various system routines and tasks, identification and tracking purposes like vehicle tracking, route determination and navigation etc. Vehicular embedded systems have limitations in their physical and logical capacities. For example, they usually have small memory, processing power and disk storage. These limitations lead to less ability to perform like how normal computer systems would perform.

A boat, like other transport systems, also has different subsystems with features and sensors that produce data that could be used by its owner or repair person to make relevant decisions. For example, these data can be useful in determining how soon or later a boat needs to have service maintenance routine. Since not all boats contain interactive screendisplays to display these data, Diadrom AB, company based in Gothenburg proposed an implementation that could allow these data to be displayed on mobile devices without a need to modify a boat's primitive display. This implementation allows data to be displayed onto mobile device's web browser after a phone or tablet connects to the boat's WiFi network.

These sensitive data will be transported from one of boat's computer unit commonly referred as Electronic Control Unit (ECU) with WiFi hosting capabilities through WiFi network over HyperText Transfer Protocol (HTTP) to a mobile device. Attackers can eavesdrop communication, make modification to the data or even send their own wrong information to the mobile device. Wrong information can have a direct or indirect effects to users of the system or vehicle. For example can make owner of the vehicle decide to delay maintenance routine in the case when it needs immediate maintenance. This could lead to degradation of the vehicle's performance and in the worst case scenario cause an accident.

One possible security measure to mitigate this problem is to encrypt all communication between vehicles ECU with WiFi hosting capabilities and mobile device. Encryption can be done on the communication channel used (i.e wireless protocols like WPA2 can be used for this) or by end-to-end (i.e Public Key Infrastructure over secure HTTP protocol). For this thesis work, focus will be on the later.

Transport Layer Security (TLS) is a protocol that is widely used to secure HTTP

traffic [12] [8]. When TLS is used over HTTP, it is what is commonly known as HTTP Secure (HTTPS) that is well known for securing website data when doing security critical activities like electronic payments etc. TLS is known to have a big performance overhead especially during initial handshake after Transmission Control Protocol (TCP) connection has been established [7]. TLS handshake requires at least 4 handshake rounds that for very small chunks of requests could turn out to be overheads [12].

Since vehicle's ECUs are embedded computers that are limited in terms of computation power, implementing TLS without having any idea about its performance implications is a problem. Doing this can lead to improper configuration that could enforce weak keys generation due to computers' low entropy. It could also mean that different secure protocols will underperform in terms of security or in other cases provide security in a fairly low level that what it is expected to.

The main purpose of this thesis work is to research and analyse performance implication of Transport Layer Security (TLS) protocol when implemented in vehicular embedded computers. Furthermore, to address tradeoffs that computers in boats have with regards to relationship between security, usability of the system and computational performance when TLS protocol is being used.

To achieve goals for this thesis work, a number of performance tests were done on a prototype system using a simulation program written in Java. These tests aimed at measuring performance of the system for normal usage of the system, for each operation in the HTTP request/response stack and for individual TLS operations involved with the secured version of the system.

Results from these tests have shown how TLS implementation perform as compared to when not using TLS. Together with that results have also shown how each of the TLS encryption schemes and algorithms perform on the host computer. Furthermore results have shown how there is a close relationship between security of the system and usability and how these two needs to be considered when implementing a secured system.

This report is a document presentation of the thesis work that has been done. It has different chapters that talks about different aspects of this thesis work. Chapter called **Background** in this report talks about different technologies that were used to achieve results. Chapter called **Methodology** talks about research methodologies that were used in this work. It gives an idea about research questions paused, hypotheses, system requirements and delimitation towards end of the chapter. **Setup and Testing** chapter talks about how prototype and simulation was built. It also talks about tests that were performed; What were the goals for each test and how they were setup. **Results and Discussion** chapter gave detailed review of the results obtained and analysis of the results for each of the test performed. It also brought a brief discussion about comparisons between the two implementations not that results were obtained. The last chapter is **Conclusion** where conclusion of

the work will be talked about based on this research work and recommendation for future works.

2
Background

2.1 Transport Layer Security

Transport Layer Security (TLS) is a protocol that is widely used to secure HTTP traffic. It is an Internet Engineering Task Force (IETF) standardization initiative aimed at producing an Internet standard version of Secure Socket Layer (SSL) protocol [12]. It is a successor to SSL but often used interchangeably. When TLS is used over HTTP, it is what is commonly known as HTTP Secure (HTTPS) that is well known for securing website data when doing security critical activities like electronic payments etc. TLS has a biggest performance drawback especially during initial handshake after Transmission Control Protocol (TCP) connection has been established.

To get a clearer picture, the following diagram shows the basic architecture of a TLS protocol. It shows the basic parts of TLS protocol packet and how they relate in order of their time of use.

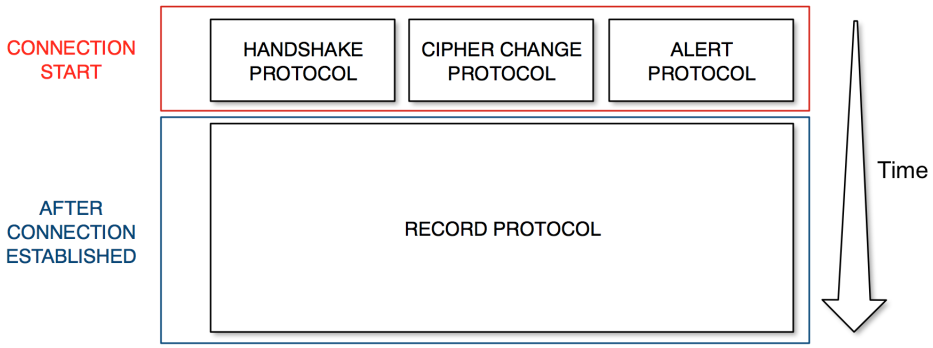


Figure 2.1: TLS Protocol Stack with respect to time each part is being used.

TLS handshake requires at least 2 handshake round trips that for very small chunks of requests could turn out to be overheads [12]. This is because of the cryptographic keys that are being generated and shared between the two communicating parties.

The following diagram shows TLS handshake protocol that involves operations leading to establishment of secure connection between client and server.

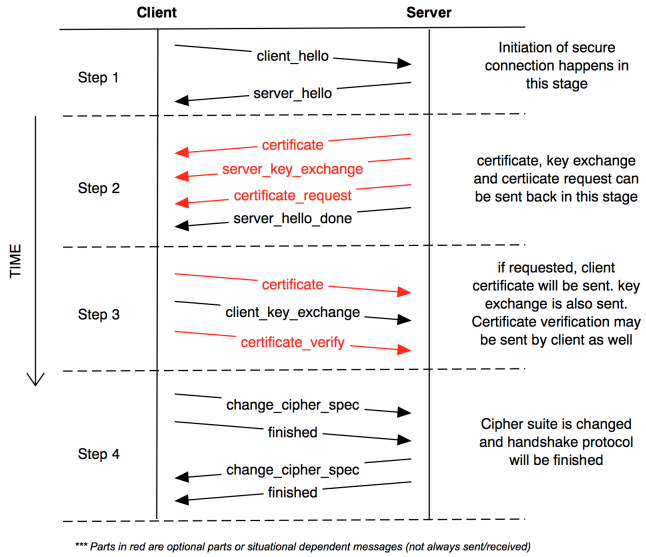


Figure 2.2: TLS handshake operations [12].

All of the operations that involved requests and responses shown above, takes a considerable amount of time that later on, it counts to the total time spent performing these TLS operations.

2.2 Curl Tool

As explained in a curl online manual page [1] "Curl is a tool to transfer data from or to a server, using one of the supported protocols (DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET and TFTP)". This tool is able to make HTTP requests with a command that can work even without user's interaction.

With this tool, there is a number of functionalities that one can use. To list a few, the following are some of the functionalities that one can get from this tool.

- Get the main page from specific web-server:
  Command is :

```
curl http://www.netscape.com/
```

- Get a web page from a server using specific port eg. 8000:
  Command is :

```
curl http://www.weirdserver.com:8000/
```

- To read and write cookies from a netscape cookie file, you can set both -b and -c to use the same file:
  Command is :

```
curl -b cookies.txt -c cookies.txt www.example.com
```

2.3 OpenSSL

OpenSSL is an open source project that provides a robust, commercial-grade, and full-featured tool kit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols [9]. OpenSSL is widely used across different linux distributions for different cryptographic purposes for example TLS operations based functionalities like TLS handshakes and record layer operations. OpenSSL can be used on webservers as well to provide HTTPs functionalities using corresponding openssl extension for that server.

Besides general purpose functionalities discussed above, OpenSSL also provides command line utilities for performing certificate generation, signing, verification, request etc. The following is a command to create certificate signing request (CSR) with OpenSSL in a command line tools.

```
openssl req -new -key fd.key -out fd.csr
Enter passphrase for fd.key: ****************
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:GB
State or Province Name (full name) [Some-State]:.
Locality Name (eg, city) []:London
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Feisty Duck Ltd
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:www.feistyduck.com
Email Address []:webmaster@feistyduck.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

2.4 Related Work

A paper [13] by Li Zhao et al. has addressed anatomy and performance of SSL processing. In this paper, they gave detailed description about secure session and also presented how much time is spent by various cryptographic operations and some of

common algorithms. This paper also addressed the problem that SSL and TLS face of performance and did analysis of SSL processing performance and its execution characteristics.

A paper [3] by Claude Castelluccia et al. also remarks SSL/TLS overload problem and presents different techniques to overcome this problem of expensive public key operations that TLS/SSL have. Their proposed solution is re-balancing the load by allowing some parts of the TLS/SSL handshake to be performed from the client side. The authors of the paper also mark their proposed solution as companion to combat Denial of Service (DOS) or Distributed DOS (DDOS) attacks.

What was done in this thesis work is somehow similar to what Li Zhao and his colleagues did, but with the difference as the main focus in this thesis work was analysing performance of TLS with respect to less performing computers like vehicular embedded computers that generally require more attention to performance. This thesis work presented an academic contribution to the understanding of how TLS performance can trade off with other aspects like computational cost and usability of the system hosted in a vehicular embedded computer. Equally important, how different ciphers and algorithms combination that TLS implementations supports performs in an embedded computer setting.

2.5 Ethical Consideration

This study about performance of TLS protocol in vehicular embedded computers contain no obvious ethical concerns. However, when considering the two implementations, secured (HTTPS) and unsecured (HTTP) that this particular scenario is addressing, one can recognize some ethical concerns regarding the potentiality of sensitive information that is handled.

As users will be accessing sensitive data of the vehicle through mobile application that help them make important decisions, these data in wrong hands would lead to potential risk of loss of integrity and confidentiality and in some cases even availability. It is vital that when making decisions about what implementation to use, to also think about these three main pillars of computer security.

It is important to realize that, despite performance bottleneck that TLS imposes as we have seen in this research work, it is still vital when it comes to protecting sensitive data from security breaches. TLS offers all three pillars, integrity, confidentiality and availability of data when properly configured.

3

Methodology

3.1 Introduction

This thesis work has been conducted as empirical research. Empirical research is a type of research based on empirical evidences. Empirical research allows us to gain knowledge from directly or indirectly observing or experiencing a researched phenomenon [5]. This type of research has the following characteristics: Imposes specific research questions that are to be answered; Defines a case study, a behavior or phenomena that is to be studied; Offers description of the process used to study and in some cases test hypotheses of the case studied or phenomena, taking into account controls, selection criterias, and testing environments (such as simulations). In this thesis work and report, I have addressed all of these characteristics.

To reach intended thesis outcomes, this work was divided into the following parts that all together supported reaching the conclusion.

1. Research on the current security requirements and implementations being used in the industry. Research on general trade offs that such implementations have. Then based on the requirements, write down specifications using assumptions found from the research done. Then hypotheses are to be generated so that they are tested based on the specifications.

2. Prepare hypothesis test environment by developing the prototype of the embedded computer to mobile device HTTP communication. Then implement the same communication test environment with HTTPS and making sure that TLS protocol is used.

3. Write programs that will test for performance from general usability testing to how each of the individual TLS operations performs. Then measure performance of the test environment above by iteratively adjusting the test for the different TLS cipher sets, and payload sizes.

4. Analyse data using the performance results I got and use them to compare performance of each of the combination and decide on optimal implementation. Plot graphs to simplify analysis stage.

5. Based on research findings in steps 1-4, compare and weigh tradeoffs that

each implementation has and based on security requirements in step 1 draw a conclusion for the hypotheses in step 2 and recommendation on the optimal implementation by addressing optimal ciphers combinations to be used and how they imply to the security of the HTTP communication between ECU module and mobile device.

The following is a representational diagram of my thesis's research work procedure.

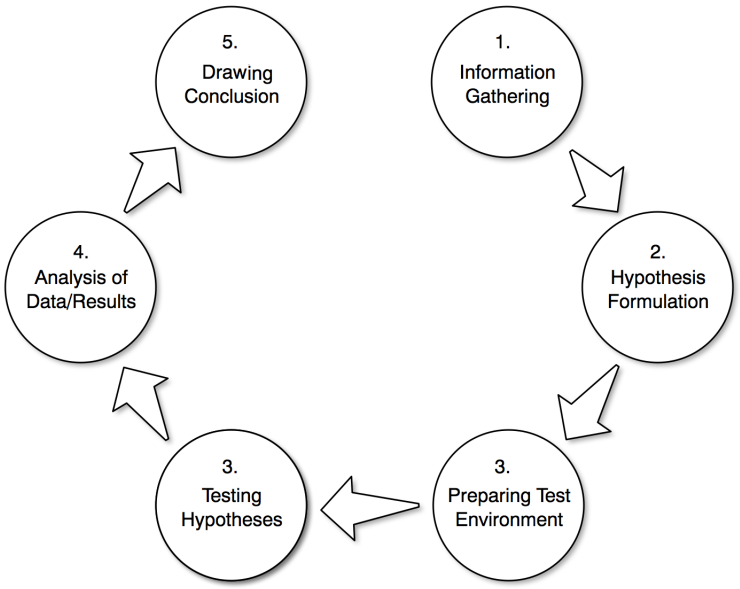


Figure 3.1: TLS Protocol Stack with respect to time each part is being used.

3.2 Research Questions

The following are the research questions that address the research problem and how they have an effect to the user of the system.

- **RQ1:** How long does Transport Layer Security (TLS) Protocol operations take to make a round trip when implemented in Vehicular Embedded System (VES)?
  Since TLS comprises Public Key Infrastructure (PKI) operations during initial handshake process that are expected to take more resources to finish and since VES are limited in terms of resources like computational, memory etc., it will be interesting to find out how long each TLS operation would take to complete when implemented in VES.

- **RQ2:** How does the performance of TLS affect the usability of the system proposed to end-users?

If RQ1 leads to performance degradation, how does it affect the system's usability to end-users? Does it lag a lot to extent that the system is completely unusable to its end-users?

- **RQ3:** How can the TLS protocol stack be simplified so that it impacts less on performance while still providing useful functionalities?
  If at all TLS leads to performance degradation, can it be simplified further so that it improves performance but have less trade-offs on its security?.

3.3 Hypotheses

As a way of refining the focus of this research, the following are hypotheses that I came up with that I will be testing in this research work.

- **H1:** Algorithms requiring long keys degrades performance of the TLS protocol in vehicular computers.

- **H2:** Much more secure implementation of the TLS protocol causes the system to cost more. ie. in terms of power management, network bandwidth and memory.

- **H3:** Security and usability of the system proposed is affected by performance of the vehicles embedded computer.

3.4 Security Requirements of the System

The following are the security requirements that were taken from the targeted system specification. Some of these requirements may be out of scope of my thesis work but significantly identify different security mechanisms that have been put into account when developing the system.

Requirements from the project specifications are as follows:

1. The system is required to give access to data only to authenticated users and in a secure manner.
2. The system is required to encrypt HTTP data transferred between ECU and a mobile device through a web service.
3. The system is required to encrypt wireless communication between ECU and mobile device.
4. The system is required to have a balance between security and its performance without affecting usability.

Additional requirements from the specification evaluation:

5. The system needs to have 2 factor authentication to increase security.
6. The system needs to encrypt HTTP traffic through TLS implementation.

7. The system needs to encrypt wireless communication using standard wireless security protocols like WPA/WPA2.

3.5 Limitations

While the project aims at securing HTTP/s communication between vehicles' ECU and a mobile device in a wireless communication fashion, getting actual usage data was infeasible since there is no system implemented in a vehicle already in usage right now. Instead in this thesis work, simulations of actual usage were created with a Java program and all measurements and readings were based on these simulations. Readings were based on actual usage patterns and run in a span of 3 weeks continuously so as to get much more accurate readings.

4

Setup and Testing

This section gives an in depth description of how testing and simulation was performed to achieve results that helped in analysis stage of the thesis work.

To test hypotheses, a number of tests were performed that were categorized into three stages. Usability testing, TCP/HTTP connection testing and TLS operations testing. All these three tests happened one after another in an order as shown with the diagram below. Results from the first test led to further investigation with the second test. Likewise, results from the second test led to further investigation with the third test. Data for each test was collected and used in the analysis stage that gave deeper understanding of the research area. This understanding was complemented with answers to original research questions and hypotheses that were stated in the previous section.

Stages of testing followed in this research work are as shown in the figure below.

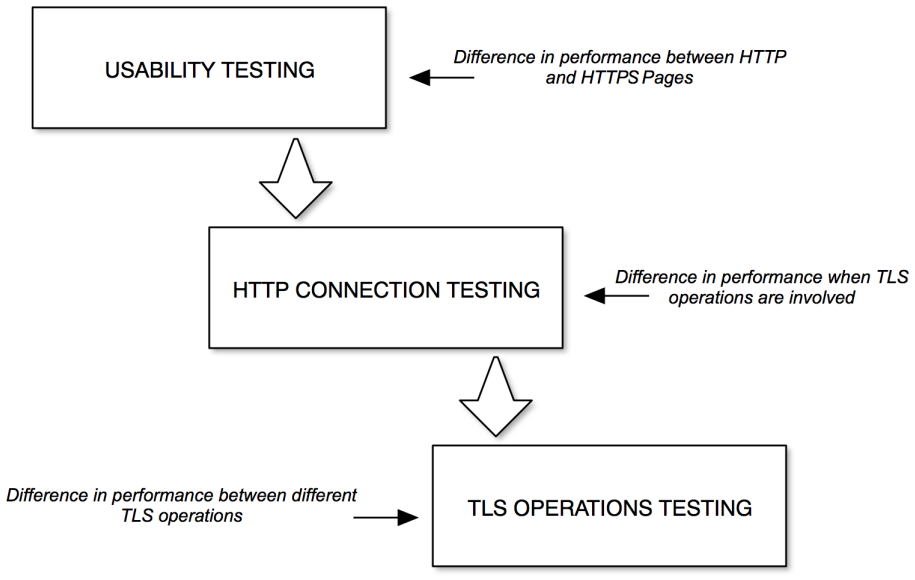


Figure 4.1: Steps followed to perform tests.

The diagram above shows how different tests that led to discoveries and deeper in-

vestigation through other tests.

4.1 Usability Testing

This was the first type of testing that was performed. In this test, it involved usage scenario that a normal user of the system is expected to follow and based on it, the time it took for each of that usage session was collected. A single usage session as defined in this research work is a combination of different web pages that user would visit/open as to achieve one goal(it can be to view data in dashboard, can be to login etc.). For example, if user wants to view dashboard data, would visit a login page, then display page and lastly a dashboard page. Therefore, for a single usage session in this scenario, user visited 3 pages: **/login**, **/display** and **/dashboard**. Each run of this test was aimed at a single usage scenario but it was performed with a number of iterations and average response time was recorded.
Expected outcome of this test was a dataset file containing average time taken to complete one usage session together with times for each of the iterations and per page readings. The figure below shows a sample dataset file that was expected from this test.

SESSION: SafeThread	
NO PAGES: 3	
Iteration no:1	
PAGE: bouer.local/login	TIME:1.342 seconds
PAGE: bouer.local/display	TIME:0.504 seconds
PAGE: bouer.local/dashboard	TIME:0.42 seconds
Iteration no:2	
PAGE: bouer.local/login	TIME:0.763 seconds
PAGE: bouer.local/display	TIME:0.436 seconds
PAGE: bouer.local/dashboard	TIME:0.309 seconds
Iteration no:3	
PAGE: bouer.local/login	TIME:0.918 seconds
PAGE: bouer.local/display	TIME:0.397 seconds
PAGE: bouer.local/dashboard	TIME:0.442 seconds
Iteration no:4	
PAGE: bouer.local/login	TIME:1.3 seconds
PAGE: bouer.local/display	TIME:0.568 seconds
PAGE: bouer.local/dashboard	TIME:0.453 seconds
AVERAGE TIME:	1.08075 seconds

Figure 4.2: Sample output dataset file from the usability test.

4.1.1 Goal

The goal of this test was to measure time difference of web application usage when application is secured with HTTPS as opposed to when it is not secured (i.e when

its just HTTP). This test helped to quantitatively realize if there were differences in performance when two protocols (HTTP and HTTPs) are implemented in this system and if those differences had effects to usability of the system. It also helped get answers to research question 2 (RQ2) and insights to hypothesis 3 (H3) of this research.

4.1.2 Setup

This test involved testing time it takes for one session of usage that combined loading a number of website pages. For example, if a normal user will first visit a ***/login*** page then ***/display*** page and then ***/dashboard*** page; then, get requests to these three pages will be taken as a single usage.

For this test, Java program was written to execute curl command and collect readings from it.

The following figure shows a pseudocode of how request time was collected by this Java test program.

```

/*
Pseudocode for measuring time taken by each web page
*/
$micro_page_start = new java.util.Date().getTime(); //Just before start of the
    request
$URL = [URL TO THE PAGES; CAN BE SECURE OR INSECURE PAGES]
$PARAMETERS = %{time_connect}, %{time_starttransfer}, %{time_total}
curl -o /dev/null --insecure -s -w $PARAMETERS $URL
$micro_page_end = new java.util.Date().getTime(); //Just after the request ended
$TIME_TAKEN = ( $ micro_page_end - $micro_page_start ) / 1000 Seconds
```

Code : How time was recorded before and right after the request call.

As it can be seen from the code written above, time was recorded just before the request was performed and right after the request has returned data. For each page, time it took to for that page to load was collected. And later on, time it took to run that single usage session was calculated as a difference between the two times. For easy reading, time data for this test were set to seconds time resolutions. Furthermore, for data precision purposes, the test was run for measurement for a specified number of times (in this case 100 times), then the average time was obtained.
Figure 4.3 below shows how I did setup the usage testing environment.

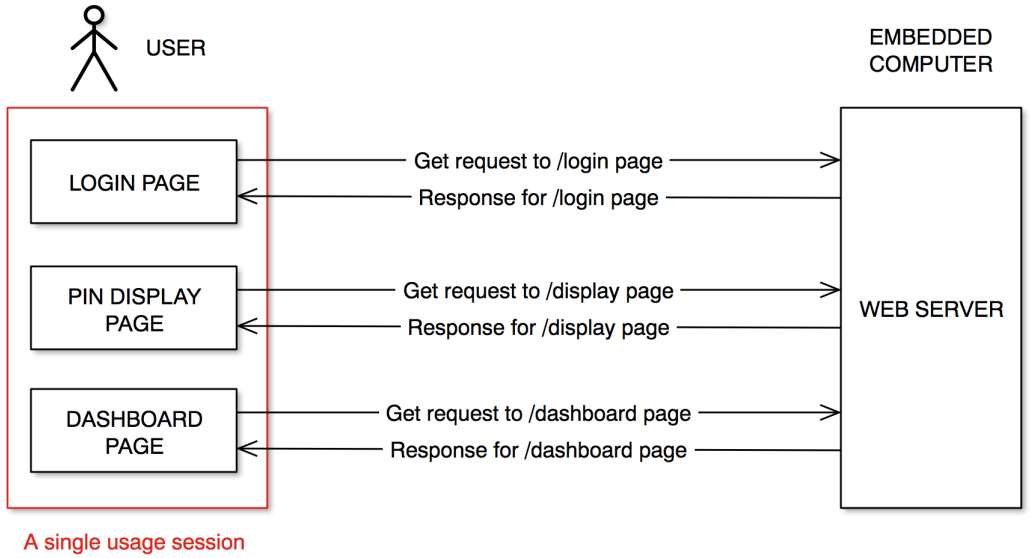


Figure 4.3: Users' single usage requests.

Using the scenario as shown in Figure 4.3 above, simulation was set to run each usage session a number of times that would be specified at run-time. Each round was run for both secured version and unsecured version of implementation (i.e requests going to HTTPS and HTTP).

Source codes of the program used to perform this test can be found in appendix A.3 section of this report.

4.2 HTTP Connection Testing

After time difference between the two implementations was observed, this test was performed to closely see where the delay was experienced in the HTTP request to response operations. Whether it was the TLS implementation or any other reason that might come up due to difference in their respective implementations.

This test was aimed at measuring time it took for each individual operation in the TCP operations stack to complete; From connection setup to data transfer to connection ends. As the test was running, payload of a fixed size was requested and time taken until completion of the requested was recorded. Data was collected with each row containing a specific payload size against time taken by different HTTP connection operations.

The following is a screenshot from sample comma separated values(.csv) file containing data from this test.

ThreadName	t_PayloadSize	t_NameLookup	t_AppConnect	t_Redirect	t_Connect	t_Pretransfer	t_Transfer	t_Total	t_OveralTotal	t_Timestamp
UnsafeThread-1	512	0.00000000	0.00000000	0.00000000	0.00660100	0.00000000	0.06743200	0.06761300	0.11010000	1461344343919
UnsafeThread-2	512	0.00000000	0.00000000	0.00000000	0.00689500	0.00000000	0.06734400	0.06754000	0.11007700	1461344343919
SafeThread-2	512	0.06726200	0.06726200	0.06726200	0.00666500	0.06726200	0.13621100	0.13632800	0.18062800	1461344414219
SafeThread-1	512	0.06790200	0.06790200	0.06790200	0.00685300	0.06790200	0.13852400	0.13864000	0.18309300	1461344416685
UnsafeThread-1	1024	0.00000000	0.00000000	0.00000000	0.00705000	0.00000000	0.08484500	0.08503100	0.13125300	1461344475276
UnsafeThread-2	1024	0.00000000	0.00000000	0.00000000	0.00697900	0.00000000	0.09110900	0.09130200	0.13766900	1461344481648
UnsafeThread-2	1536	0.00000000	0.00000000	0.00000000	0.00687400	0.00000000	0.09189600	0.09212200	0.14282000	1461344624541
UnsafeThread-1	1536	0.00000000	0.00000000	0.00000000	0.00660000	0.00000000	0.10922500	0.10942300	0.16027400	1461344635619
SafeThread-1	1024	0.08151300	0.08151300	0.08151300	0.00657500	0.08151300	0.16969000	0.16979200	0.22037900	1461344637121
SafeThread-2	1024	0.08127100	0.08127100	0.08127100	0.00657800	0.08127100	0.18245000	0.18259000	0.23326900	1461344647548
UnsafeThread-2	2048	0.00000000	0.00000000	0.00000000	0.00675400	0.00000000	0.14329900	0.14350600	0.19660900	1461344821218
UnsafeThread-1	2048	0.00000000	0.00000000	0.00000000	0.00662800	0.00000000	0.15331400	0.15351600	0.20691400	1461344842604

Figure 4.4: Sample dataset file collected from this HTTP connection testing.

4.2.1 Goal

The goal of this test was to understand the performance in terms of time taken by each operation in TCP operations stack for HTTP and HTTPS. It also helped in answering and getting insights on research question 1 (RQ1) and hypothesis 2 (H2) of this research work.

4.2.2 Setup

In this test, web application was designed to return contents of length/size equal to the number specified in the get request. When a test runs, HTTP get request is sent with a header containing size value that represents number of bytes that user wants to get back. An application would automatically return pseudo random string of length equivalent to the size specified. For each run, a call to the webserver can be run multiple times and an average time values are obtained so that to add more precision to the results to be collected. The following diagram shows pseudocode of the web application running on the vehicle that would return contents of size based on the size specified in the HTTP get request headers.

```
/* Pseudocode for HTTP response with contents of size as specified in GET request
    */
HTTP::Request['/get_response', function(){
    $size = HTTP::GET['size']// Specified in the http request
    $random_bytes = OpenSSL_random_pseudo_bytes( $size / 2 )
    $response_contents = bin2hex ( $random_bytes ) // bytes to string
    return $response_contents  //returns the response as http content response
}];
```

Code : Response generated from the input request parameters

For example, if user requests for a URL, response was generated as shown in table 4.1 as follows:

URL Requested	Response	Description
/get_response?size=4	e2cd	Response of size 4 bytes is returned
/get_response?size=8	21d9d93c	Response of size 8 bytes is returned
/get_response?size=10	3f8be27e04	Response of size 10 bytes is returned

Table 4.1: HTTP Get requests with their respective response.

Figure 4.5 below shows how the response looks like if user requested for the following URL http://10.5.1.42/get_response?size=230

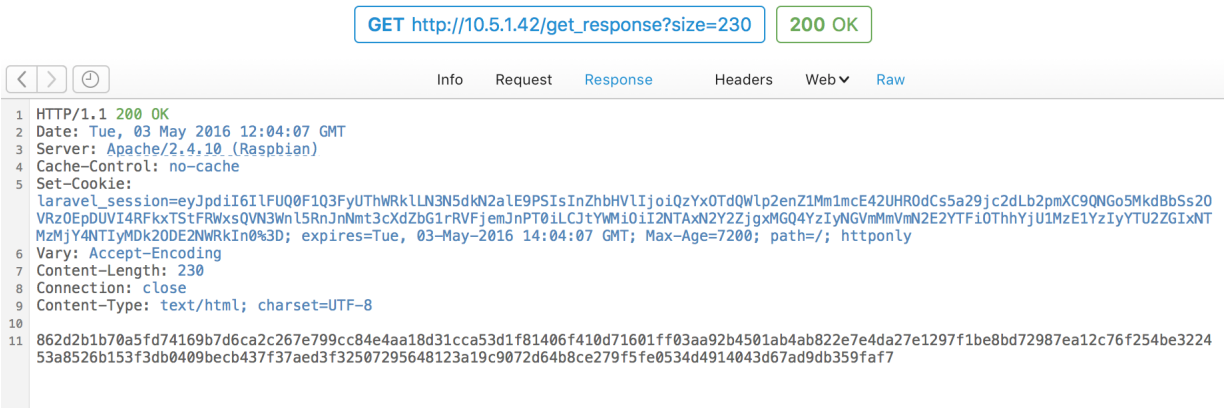


Figure 4.5: HTTP Response for http://10.5.1.42/get_response?size=230

With the above setup, for both HTTP and HTTPS, it became easy to setup a simulation that requests for the page expecting a response of a desired content-length. That led to a test program to be written in Java that made requests for each specified payload size and various time quantities being measured.

Quantities measured in this test are the following:

- **t_PayloadSize** : Payload size in bytes that were transferred from the vehicle to mobile device in a particular request call.
- **t_NameLookup :** Time taken from the start of the request start to the time name resolving was completed.
- **t_Connect :** Time taken from the start until the TCP connect to the remote host was completed.
- **t_Redirect :** Time taken for all redirection steps including name lookup, connect, pretransfer and transfer before the final operation was started.
- **t_Pretransfer :** Time taken from the start until the response contents transfer was just about to begin.

- **t_Transfer :** Time taken from the start until the first byte was just about to be transferred. Includes time_pretransfer and also the time the server needed to calculate the result.
- **t_Total :** The total time that the full operation lasted
- **t_OveralTotal :** Time elapsed from the curl command was just about to start to when it just finished running and results are received.
- **t_Timestamp :** Unix timestamp when data was recorded.

The following diagram show the quantities stack in order of their appearance.

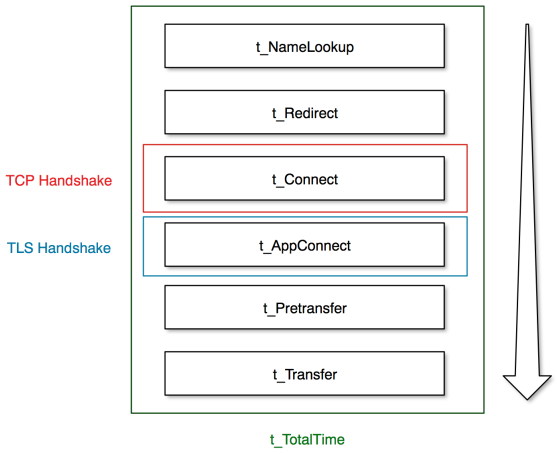


Figure 4.6: Quantities measured in this test in order of appearance in HTTP call.

In this simulation, each test was allowed to run for each payload multiple times (for this research 1000 times) and average time values were recorded to even out random noise. Tests were also run on different payload size and for each of the two implementations; unsecured HTTP and secured HTTP. For this test, payload sizes begun from 1024 bytes (1KB) to about 1 Megabyte with an increment of 1024 bytes between each request.

Figure 4.7 below, shows how test iterations were divided into equal intervals.

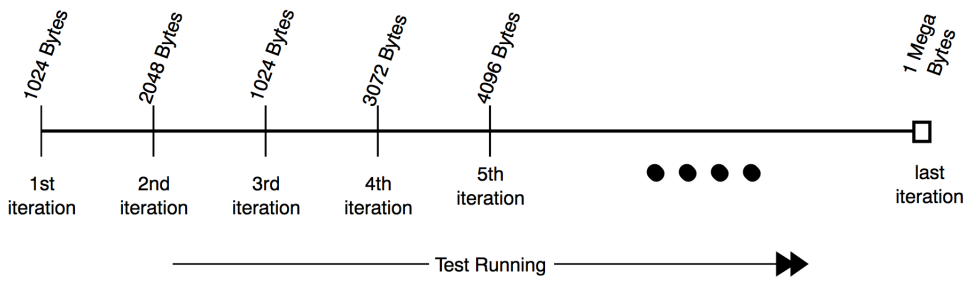


Figure 4.7: How test iterations were divided equally.

Source codes of the program used to perform this test can be found in appendix A.4 section of this report.

4.3 TLS Operations Testing

After test done in the previous subsection giving some insights on what led to HTTPS implementation taking longer time, this test was performed to better understand the reason for it and if there is anything more to learn from the TLS protocol. Discovering that TLS handshake and TLS record layer operations that were causing performance drawback hence affecting user experience, test for the server side OpenSSL operations were performed to see which of the TLS encryption schemes were taking longer; Whether it was TLS record layer or TLS handshake or different cipher sets and key sizes that affects the response time. This test helped in understanding exactly a point in the HTTPS operations that causes notable overhead.

In this test, each of the supported TLS encryption schemes and algorithms were tested against the host system to see how long they take and how many operations can be run per second by each of them. This information will be useful in deciding what cryptographic algorithm would be expensive for the server (in terms of operation time) and what would be the optimal.

Results of this test were expected to be performance measures for each of the TLS supported algorithm and encryption scheme. The following figure shows sample of an expected results from this test.

```
Measurements below are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
md2                 0.00         0.00         0.00         0.00         0.00
mdc2                0.00         0.00         0.00         0.00         0.00
md4              4933.21k    23572.91k    48465.92k   108957.02k   133723.48k
md5              3381.50k    15444.01k    41407.96k    65603.58k   112383.32k
hmac(md5)        4613.87k    13448.73k    48502.44k    67693.23k    88509.10k
sha1             5099.76k    12600.21k    27569.24k    51946.15k    47047.83k
rmd160           3388.98k    12494.02k    23947.39k    30169.64k    45993.01k
rc4             40473.35k    55230.87k    58049.50k    48632.47k    65459.54k
des cbc         10150.09k     9970.81k    13358.06k    11038.31k    11118.11k
des ede3         4532.57k     3771.77k     4692.16k     4274.43k     3827.50k
```

Figure 4.8: Part of the sample of expected output from TLS operation test

4.3.1 Goal

The goal of this test was to measure time taken by each of TLS operations and algorithms when running on a specific host computer. How each of the supported TLS ciphers were performing against a computer to be used. This test also helped me to answer research questions 1 and 2 (RQ1 and RQ2) and hypotheses 1 and 2 (H1 and H2) of this research work.

4.3.2 Setup

This test was performed entirely on the server side since all the TLS operations i.e handshake key generations, encryption and decryption, compression etc. are mainly done from there. This test was performed with the help of OpenSSL test functionality. OpenSSL has a command called speed that tests how specified cipher or algorithm performs in that respective computer.

The following lines of code shows pseudocode of command executed by OpenSSL to benchmark different TLS supported ciphers in a host computer.

```
/* Pseudo-code for OpenSSL benchmarking functionality. Outputs measurement of
    performance of different supported encryption schemes and algorithms. */

$options = [Specific algorithm or cipher or when left blank; tests all ciphers]
OpenSSL speed $options
```

Code : OpenSSL command to test speed of different Ciphers.

To better understand the results from this test, Figure 2.1 shows how TLS stacks with two main parts that occur during the TLS protocol lifetime. The first part is when there is a negotiation of keys, protocol and sets of ciphers to be used. And the second part is when data is being encrypted and transferred using TLS session negotiated in step above.

During connection start, there is a number of operations that happen. These operations include compression of data and encryption and decryption of data. These operations are performed using one or multiple algorithms that each have their own performance implication and directly or indirectly affect total time taken by TLS protocol. Likewise, after connection has been established, because symmetric key has already been established, encryption and decryption makes use of fewer operations as compared to the first step.

In this test, each of the supported algorithms for compression, encryption and decryption where tested against host computer. With OpenSSL command explained above, it allowed this test to get measurements of their performance in terms of execution time and operations per second. This functionality runs on the computer where OpenSSL is installed and it automatically measures the time taken by each of the algorithm.

5

Results and Discussion

5.1 Usability testing

After carefully performing usability testing, a number of interesting findings were found. Below are figures showing charts that were drawn from the data that were obtained. These charts represent two different scenarios; first, was when user used secured system i.e HTTPS protocol implemented. Second, when user used unsecured system i.e HTTP protocol implemented. The first figure contains groups of columns each representing time taken to load a specific web page in both implementations. The second figure represents difference in time taken to use the web application (load all pages in a single usage session) when HTTP is used against HTTPS.

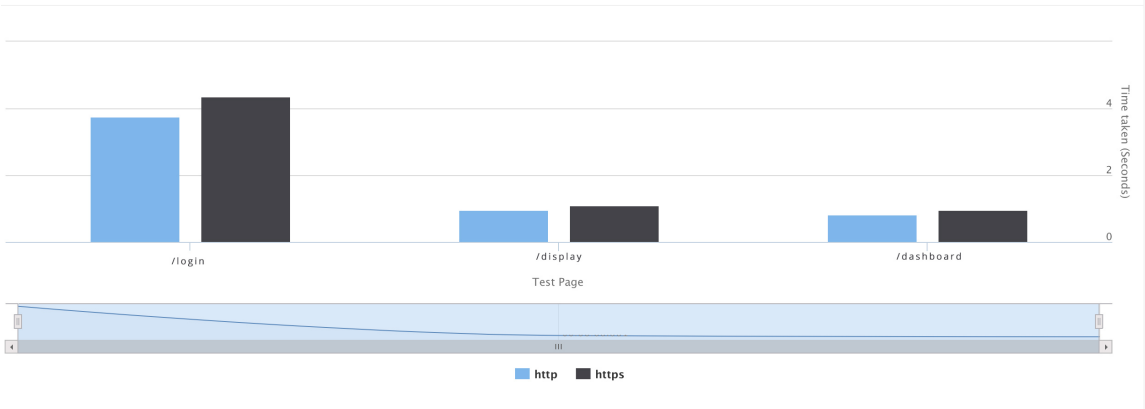


Figure 5.1: Chart of time taken for web pages to load when HTTP is used against when HTTPS is used.

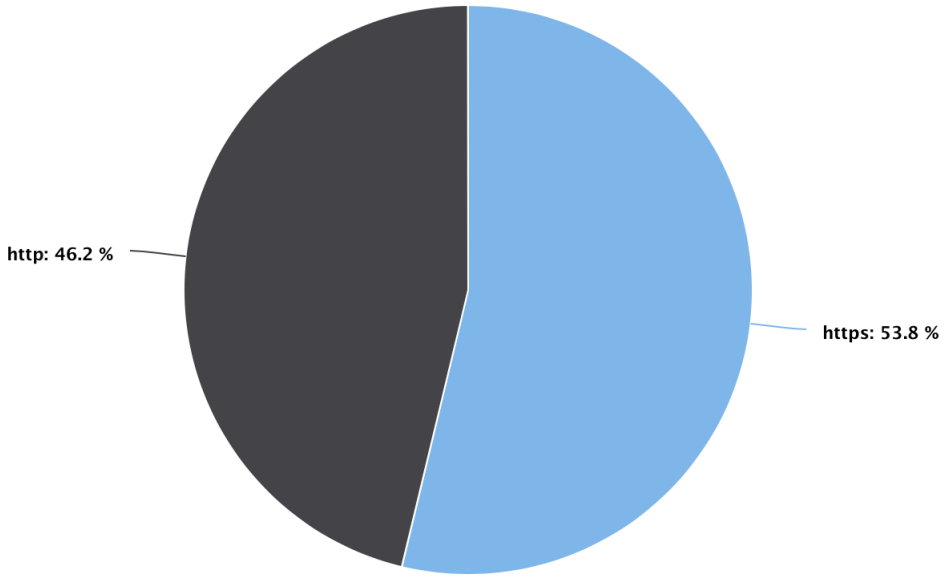


Figure 5.2: Chart of percentage time difference between time HTTP against HTTPS implementation.

As it can be observed from the two figures above, there seem to be a slight difference in performance in terms of time taken when above two different protocols are used. From Figure 5.1, we can see that in a case of loading page **/login** the difference can become as big as about 600 milliseconds, which is, according to human-computer interaction journal [6] considered noticeable to users of the system. Figure **??** shows that more than HTTPS takes about 7.6% more of the time to load web pages.
From these observations, it can be concluded by saying that using the system when secured imposed a remarkable overhead that makes it interesting to dig deeper into this to get clearer picture. These findings led to a much deeper investigation about the exact portion of the HTTPS protocol that takes longer time to execute. It led to execution of the second test with the results in the subsequent section.

5.2 HTTP connection testing

This test took relatively longer time to complete because of how it was set. As this test was running, simulation was requesting for page with same payload size 1000 times then find average of all of the time values obtained so as to increase accuracy and reduce noisy results. Then simulation was picking another payload size and do exactly like explained in step above until it finishes for all the payload size values. This simulation and test was left running for about 15 days continuously and it led to enormous amount of data that gave a much deeper insight of the case in hand.

The following figures are the charts that were obtained from the data collected during simulation run. Some figures show multiple graph lines; these lines represent

results from multiple threads that were run by the test program. Blue lines are for unsafe thread 1 and black lines are for unsafe thread 2. Unsafe threads did perform HTTP calls to pages without TLS protocol. Green and orange lines are for safe threads 1 and 2 respectively. Safe threads did perform HTTP calls to pages with TLS protocol.

Figures below show the correlation between different parts of the HTTP/S connection and how each affects the total response time. Despite this test being prone to high level of data noises, taking a large amount of datasets, helped in interpretation because it allowed a more visible pattern to be observed.

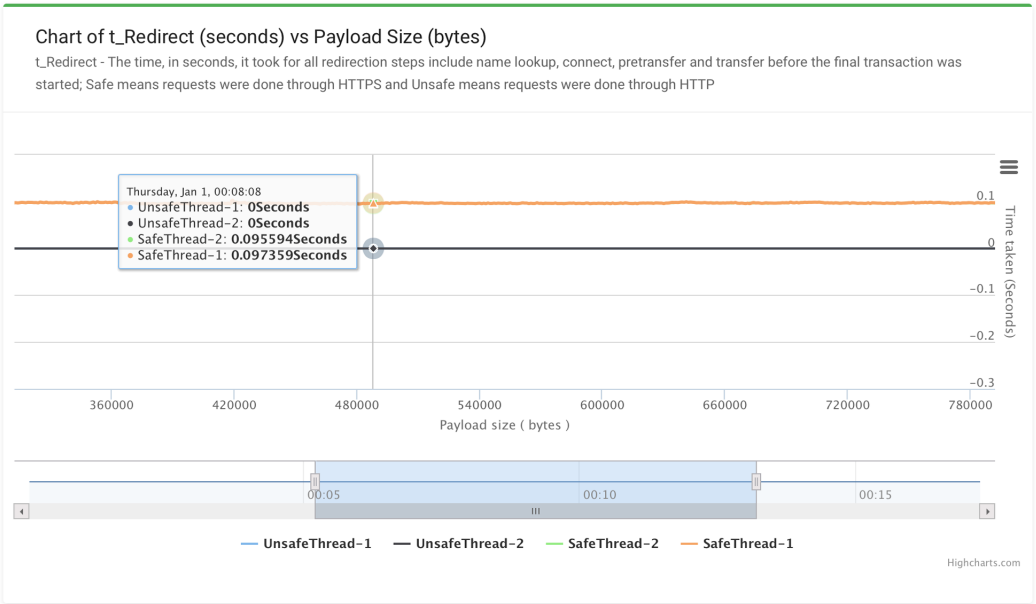


Figure 5.3: Chart of t_Redirect against payload size.

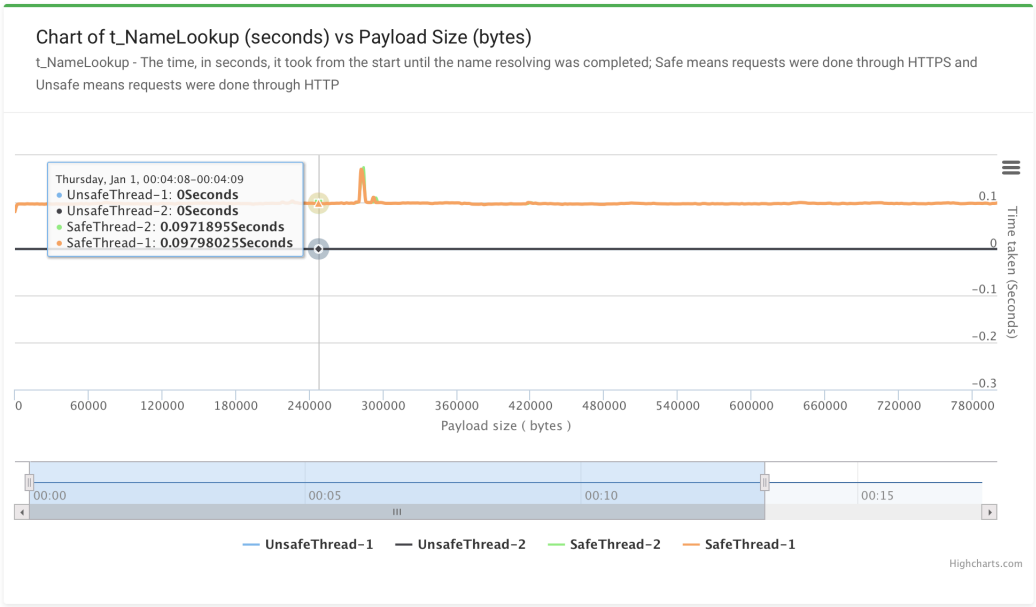


Figure 5.4: Chart of t_NameLookup against payload size.

Figures 5.3 and 5.4 above show difference in redirection and namelookup times when calls where made to the two implementations. Safe implementation registered more time delay as compared to unsafe implementation that registered 0 sec time throughout simulation run. Since t_redirect includes t_namelookup, difference that can be seen in t_redirect should have been expected since t_namelookup had the same difference. t_namelookup however, since it is time taken until name resolution was completed, it can be understood that redirection time difference is due to requests in our tests not being made with proper certificates that have been signed by authorized certificate authority (CA).

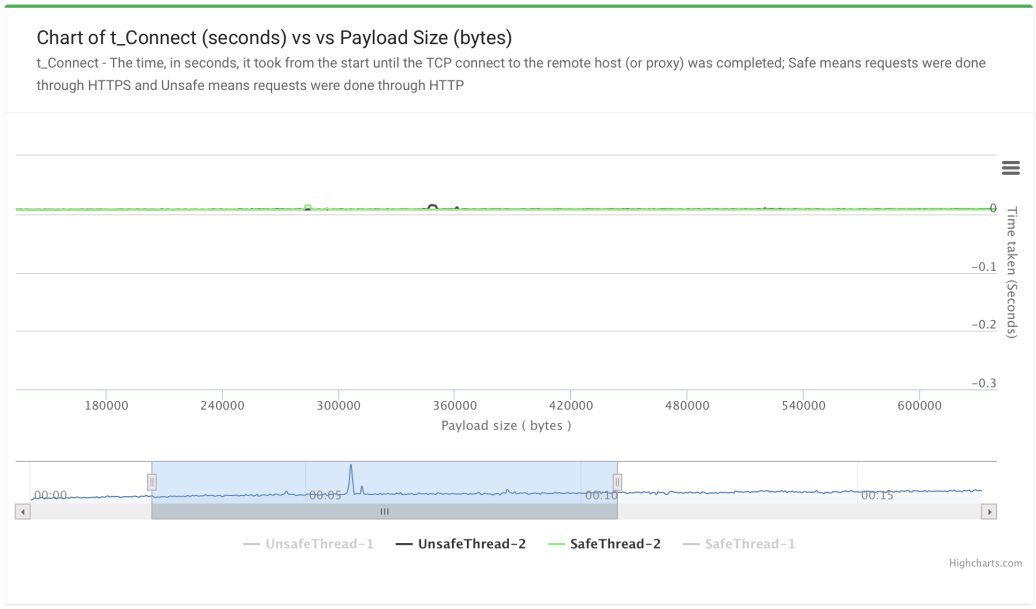


Figure 5.5: Chart of t_Connect against payload size.

Figure 5.5 above has registered almost similar values for both safe and unsafe implementations throughout simulation run. Since t_connect is time taken from start to end of TCP connection [1], this similarity in time can be explained by the fact that TCP handshake happened in both implementations and takes about same time to complete. This means that, TCP handshake time is not affected by TLS implementation. In the OSI network model, TLS is in session layer (layer 5) while TCP is in transport layer (layer 4) [11].

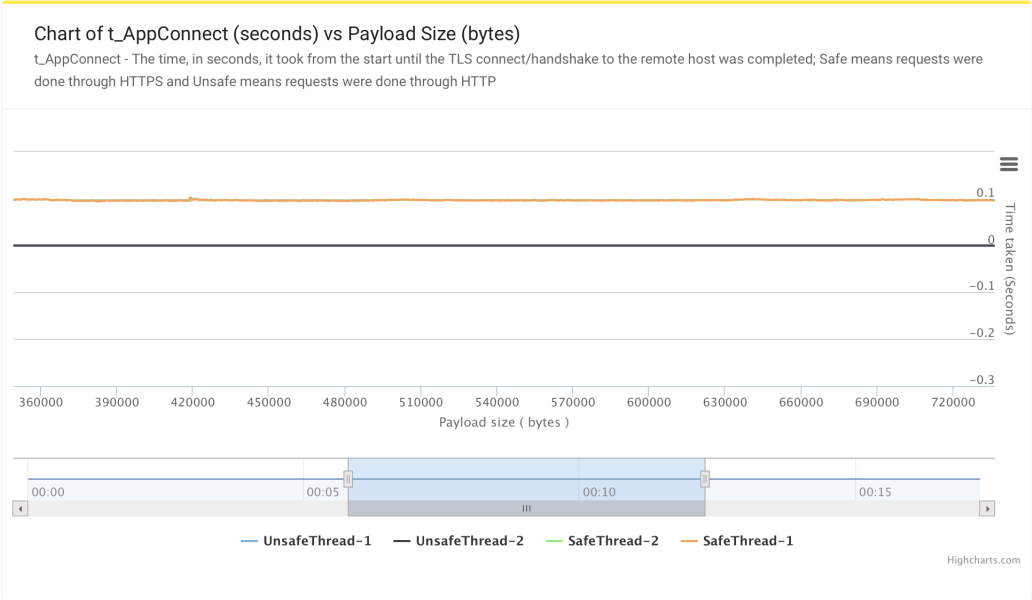


Figure 5.6: Chart of t_AppConnect against payload size.

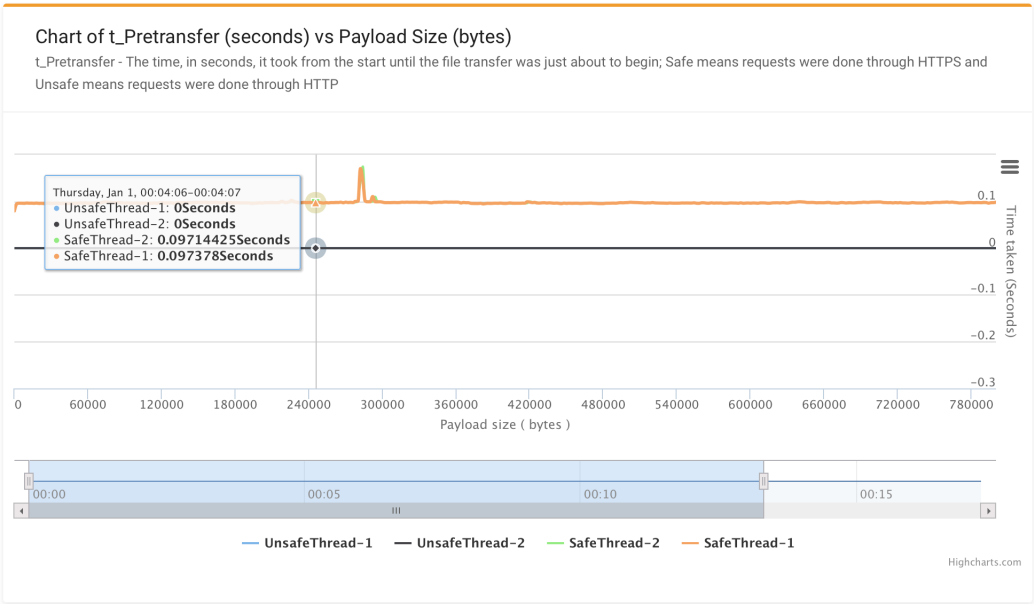


Figure 5.7: Chart of t_Pretransfer against payload size.

Figures 5.6 and 5.7 above, it can be observed that both show remarkable time difference between safe and unsafe implementations. Unsafe registers 0 second throughout the simulation run while safe implementation with TLS protocol operations has registered a noticeable delay. Since t_pretransfer contains t_appconnect and t_appconnect has shown a delay, it was expected of t_pretransfer time to also show a delay. However, since t_appconnect in this TLS setting is basically TLS handshake [1], this is an observable phenomena of TLS handshake delay.

In Figure 5.7 (t_pretransfer vs payload) above, it can be seen that both threads requesting to safe implementation have recorded some time delay contrarily to threads targeting unsafe implementation that all have recorded 0 second throughout the simulation run. Since t_pretransfer is the time taken from the start until the response contents transfer was just about to begin, this supports the fact that the delay observed in safe implementation was due to the fact that there was TLS operations taking place. These TLS operations are like TLS handshake that happens immediately after TCP handshake, TLS encryption and decryption, TLS compression, signing and verifications. All of these operations are said to have impacted to the delay that can be observed in Figure 5.7.

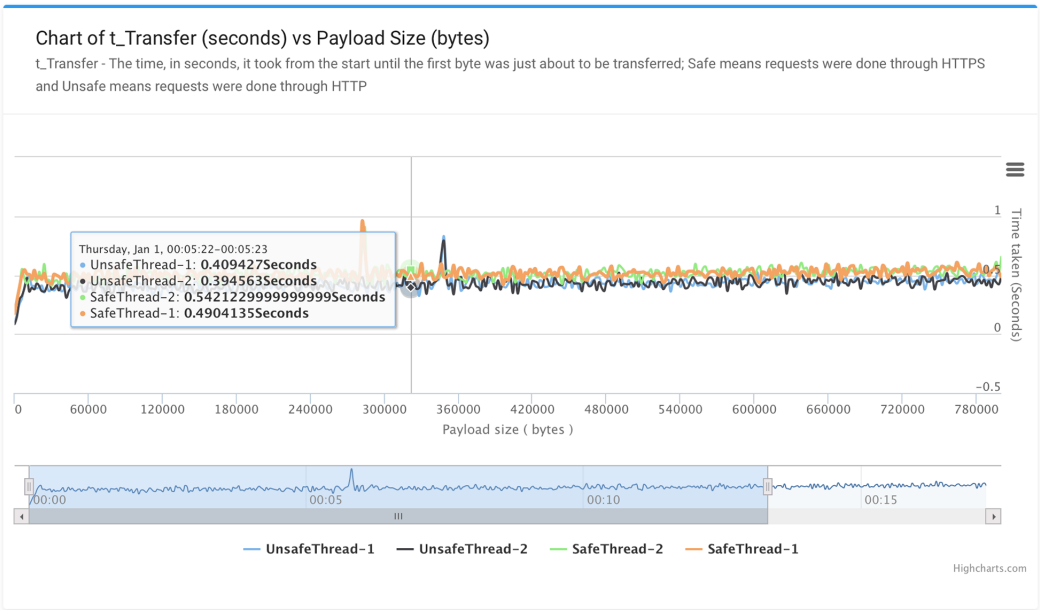


Figure 5.8: Chart of t_Transfer against payload size.

Figure 5.8 above shows how there is a small difference in transfer time between safe and unsafe implementations. The trend shows generally a small bit of similarity between the two implementations. This similarity can be accounted with the fact that in both cases, data transmitted are of the same type and for fixed TCP packet size, they should be transferred around the same time. The only explanation for the slight more time taken by safe implementation could be caused by the size of the data to be transferred. Encrypted data some times tend to be a bit bigger that unencrypted. Bigger data led to slight change in time taken for one complete transfer

of data as it can be observed in the Figure 5.8.

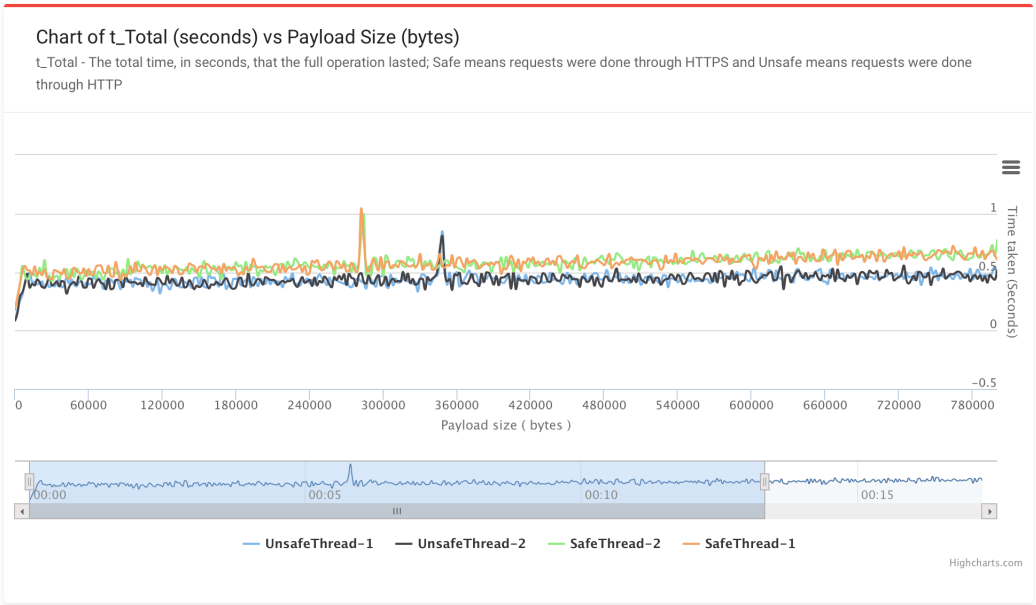


Figure 5.9: Chart of t_Total against payload size.

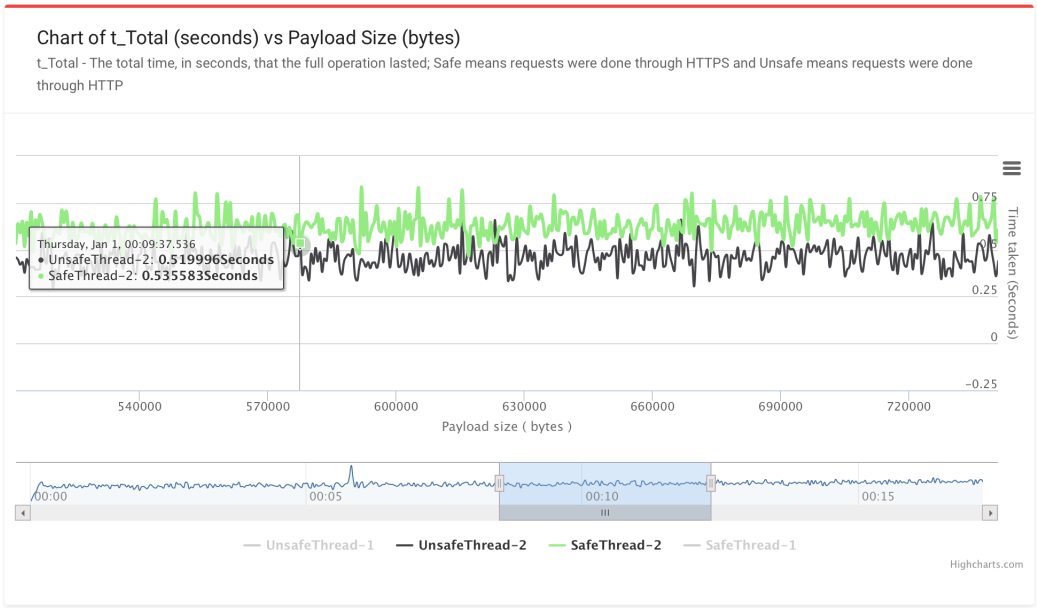


Figure 5.10: Chart of t_Total against payload size.

Both Figure 5.9 and 5.10 above show a chart of t_total against payload size. Figure 5.10 however is a zoomed part of the Figure 5.9 that shows the difference in a much closer perspective. It can be clearly observed in Figure 5.10 that there is a difference in total time spent between safe and unsafe implementation. Safe implementation takes longer total time to complete its requests. This difference is a result of the difference we saw in the figure 5.6 which means that the difference in the total time

requests were taking between two implementations were directly influenced by the t_appconnect.

In general, there is a remarkable difference between safe and unsafe implementation in the following areas: t_Namelookup, t_Redirect, t_Appconnect, t_pretransfer where unsafe recorded 0 seconds throughout. There is also a big difference in total time taken to complete requests that the two implementations have exhibited is directly influenced by t_appconnect. As TLS handshake and TLS record layer comprises of different encryption schemes and algorithms [12], findings from the test results in a previous results subsection of this test made it necessary to dive deeper into TLS operations so as to understand the difference that each of the algorithm takes. Having an idea of how each of the TLS operations perform would help in making a conclusion about performance implications of TLS in a more specific way.

Third test that was performed with the aim of investigating how much exactly does TLS operations perform in the host computer and also what is expected to take longer. Results and analysis are in the next subsection.

5.3 TLS operations testing

This test involved testing different OpenSSL algorithms and operations that are used by TLS protocol. The aim of this test was to see how long each of the algorithms/operations will take so that we can understand exactly what TLS operations are expected to take longer than the other. Results of this test involved performance measure of each of the supported crypto ciphers and algorithms that are directly or indirectly used in different TLS operations.

The tables below contain results that found after performing TLS operations test.

Algorithm Name	16 Bytes	64 Bytes	256 Bytes	1024 Bytes	8192 Bytes
MD2	0.00k	0.00k	0.00k	0.00k	0.00k
MDC2	0.00k	0.00k	0.00k	0.00k	0.00k
MD4	4933.21k	23572.91k	48465.92k	108957.02k	133723.48k
MD5	3381.50k	15444.01k	41407.96k	65603.58k	112383.32k
HMAC(MD5)	4613.87k	13448.73k	48502.44k	67693.23k	88509.10k
SHA1	5099.76k	12600.21k	27569.24k	51946.15k	47047.83k
RMD160	3388.98k	12494.02k	23947.39k	30169.64k	45993.01k
RC4	40473.35k	55230.87k	58049.50k	48632.47k	65459.54k
DES(CBC)	10150.09k	9970.81k	13358.06k	11038.31k	11118.11k
DES(EDE3)	4532.57k	3771.77k	4692.16k	4274.43k	3827.50k
IDEA(CBC)	0.00k	0.00k	0.00k	0.00k	0.00k
SEED(CBC)	11085.63k	15852.63k	13997.38k	13436.32k	16896.34k
RC2(CBC)	8182.12k	11515.25k	9678.60k	10471.78k	11126.33k
RC5-32/12(CBC)	0.00k	0.00k	0.00k	0.00k	0.00k
BLOWFISH(CBC)	15556.35k	21506.88k	21274.80k	18319.67k	23527.42k
CAST(CBC)	14185.50k	18784.30k	19582.97k	16837.09k	18383.73k
AES-128(CBC)	25597.38k	23708.29k	25623.04k	30664.02k	24159.50k
AES-192(CBC)	22600.68k	20857.90k	21972.45k	25955.83k	20602.88k
AES-256(CBC)	18485.85k	19992.23k	23227.85k	19971.78k	19038.87k
CAMELLIA-128(CBC)	16347.81k	20819.99k	17064.33k	21087.53k	19619.70k
CAMELLIA-192(CBC)	11746.75k	17398.62k	13292.26k	16238.25k	15836.04k
CAMELLIA-256(CBC)	11762.40k	17329.19k	13563.39k	15689.54k	17096.70k
SHA256	6808.69k	15184.62k	24066.65k	38517.47k	34417.32k
SHA512	1996.16k	8606.40k	10185.38k	17491.97k	18276.35k
WHIRLPOOL	698.18k	1860.49k	2482.77k	2917.54k	3701.37k
AES-128(IGE)	22429.12k	21244.61k	28857.90k	24914.34k	22820.18k

Table 5.1: Performance results for different compression and encryption algorithms. Numbers are in 1000s of bytes processed per second.

From Tables 5.1 and 5.2 above, it can be observed that as the key size increases, the number of operations that can be performed decreases. This signifies our finding and tells us that more secured/bigger key sizes leads to longer time to complete each operation that in turns leads to worse performance. This is a very important point to note, since key size most of the times implies how hard or easy it is to break the system. That being said, there has to be balance between security and usability of the system. A very much secured web application would also mean that the system responsiveness would be poor hence lead to bad usability.

Algorithm Name	Payload	Operations	Operations/time
ECDH(SECP160r1)	160bit	0.00k26s	378.4
ECDH(NISTP192)	192bit	0.00k38s	262.3
ECDH(NISTP224)	224bit	0.00k52s	191.1
ECDH(NISTP256)	256bit	0.00k66s	150.8
ECDH(NISTP384)	384bit	0.0149s	67.3
ECDH(NISTP521)	521bit	0.0343s	29.1
ECDH(NISTK163)	163bit	0.00k60s	165.9
ECDH(NISTK233)	233bit	0.0106s	94.8
ECDH(NISTK283)	283bit	0.0199s	50.2
ECDH(NISTK409)	409bit	0.0459s	21.8
ECDH(NISTK571)	571bit	0.1095s	9.1
ECDH(NISTB163)	163bit	0.00k65s	154.7
ECDH(NISTB233)	233bit	0.0120s	83.0
ECDH(NISTB283)	283bit	0.0219s	45.7
ECDH(NISTB409)	409bit	0.0524s	19.1
ECDH(NISTB571)	571bit	0.1299s	7.7

Table 5.2: Performance results for Elliptic curve Diffie–Hellman (ECDH) often used for secure key agreement (happens in TLS handshake).

Algorithm Name	Payload	Sign	Verify	Signs/time	Verify/time
RSA	512bits	0.00k1369s	0.00k0123s	730.5	8157.4
RSA	1024bits	0.00k7269s	0.00k0357s	137.6	2799.2
RSA	2048bits	0.043203s	0.00k1292s	23.1	773.9
RSA	4096bits	0.317500s	0.00k4655s	3.1	214.8
DSA	512bits	0.00k1208s	0.00k1378s	827.8	725.7
DSA	1024bits	0.00k3413s	0.00k4032s	293.0	248.0
DSA	2048bits	0.012575s	0.014239s	79.5	70.2
ECDSA(SECP160r1)	160bits	0.00k09s	0.00k35s	1123.1	286.0
ECDSA(NISTP192)	192bits	0.00k11s	0.00k44s	929.4	227.8
ECDSA(NISTP224)	224bits	0.00k14s	0.00k61s	694.0	163.0
ECDSA(NISTP256)	256bits	0.00k18s	0.00k75s	549.4	133.4
ECDSA(NISTP384)	384bits	0.00k38s	0.0192s	265.9	52.0
ECDSA(NISTP521)	521bits	0.00k71s	0.0404s	141.2	24.7
ECDSA(NISTK163)	163bits	0.00k28s	0.0112s	355.6	89.2
ECDSA(NISTK233)	233bits	0.00k57s	0.0225s	175.4	44.4
ECDSA(NISTK283)	283bits	0.00k91s	0.0396s	110.1	25.2
ECDSA(NISTK409)	409bits	0.0238s	0.0936s	42.0	10.7
ECDSA(NISTK571)	571bits	0.0596s	0.2104s	16.8	4.8
ECDSA(NISTB163)	163bits	0.00k28s	0.0131s	354.6	76.3
ECDSA(NISTB233)	233bits	0.00k56s	0.0243s	177.2	41.1
ECDSA(NISTB283)	283bits	0.00k91s	0.0470s	110.0	21.3
ECDSA(NISTB409)	409bits	0.0238s	0.1060s	42.0	9.4
ECDSA(NISTB571)	571bits	0.0595s	0.2493s	16.8	4.0

Table 5.3: Performance results for different Public Key Infrastructure (PKI) ciphers and algorithms.

From the Table 5.3 above it can also be observed that for RSA algorithm, it takes longer time to sign than to verify signature of data. This is contrary to other algorithms like DSA and Elliptic Curve DSA that shows that verification operation takes longer that signing operations. This finding tells us that when implementing deciding to use TLS implementation, there has to be a clear understanding about where to put more computational load, is it to users mobile application or vehicles' embedded computer. Which of the two computers can bear more computational load?. What algorithm should be used for encryption and decryption? These questions will help better formulate requirements in a way that would reflect overcoming performance bottleneck that TLS protocol imposes. As it was proposed in the paper [3], there is a possibility of flexibly balancing the load by adding more computational work on the client side as opposed to the current implementation.

In general, from the Tables 5.1, 5.3 and 5.2, it can be seen that different algorithms and key sizes have different performance implications. Some algorithms take longer while others take considerably less time. For example Table 5.2 shows that ECDH(NISTB571) takes a long time to operate when payload is of 571 bytes or

more. This means that, when implementing TLS protocol, this algorithm should be least expected to be used for Key agreement in PKI.

5.4 Hypothesis and Research answers

After evaluating and analysing test results, hypotheses and research questions addressed in the **Methodology chapter** were answered. The following are the answers to each of the hypotheses and research questions:

5.4.1 Hypotheses

1. **H1:** Algorithms requiring long keys degrades performance of the TLS protocol in vehicular computers.
   Answer: Yes. Results from TLS Operations tests has shown that when longer encryption and decryption are used, they take longer time to complete their respective TLS operations. This means that longer keys degrade performance of the system in terms of responsive time.

2. **H2:** Much more secure implementation of the TLS protocol causes the system to cost more. ie. in terms of power management, network bandwidth and memory.
   Answer: Yes. Results from TLS Operations tests has shown that when longer keys are used, they causes TLS operations to take longer to complete. This can be directly associated with system cost in terms of computational cost. Network bandwidth and memory can be implicitly associated as well.

3. **H3:** Security and usability of the system proposed is affected by performance of the vehicles embedded computer.
   Answer: Still Unknown. Tests conducted in this thesis work have reflected on relationship between the security and usability and not the security and usability against VECs performance.

5.4.2 Research Questions:

1. **Question RQ1:** How long does Transport Layer Security (TLS) Protocol operations take to make a round trip when implemented in Vehicular Embedded System (VES)?
   Answer: Each of the TLS protocol operations take different times that for each specific operation vary depending on different factors. Factors that were observed in this thesis work based on the results from different tests are: algorithm used, key size chosen and payload size to be operated on. There may be more factors, but these three are the ones observed in this thesis work. Different algorithms have different efficiency and operation time. Same applies to different key sizes and different payload of data to be operated on. Because

of this, it is not correct to question about performance of TLS protocol on its own without involving these factors.

2. **Question RQ2:** How does the performance of TLS affect the usability of the system proposed to end-users?
   Answer: TLS imposes noticeable difference in response time when implemented in VEC. In this thesis work, results have shown some cases when TLS registered time difference above 300 milliseconds. This is delay is above average time for delay to be noticed by humans using computer system according to Miller in his paper [6]. As it was later observed in the TLS Operations test results, this effect on usability depends on different factors like algorithms used, key sizes and payload size of the data to be operated on.

3. **Question RQ3:** How can the TLS protocol stack be simplified so that it impacts less on performance while still providing useful functionalities?
   Answer: As it has been seen in this thesis work results, what influences performance of TLS is not its architectural stack but rather other factors like algorithms used, key size and payload size of the data to be operated on. This thesis work has seen a huge difference in time taken for different operations; some cases took long time (Table 5.2 shows ECDH(NISTB571) taking 129.9 milliseconds to operate on 571bit payload data) while other cases took less time (Table 5.2 shows ECDH(SECP160r1) taking only 2.6 milliseconds to operate on 160bit payload data).

5.5 Discussion

As we have seen in this research work, despite these two protocols offering the same goal of transporting and delivering hypertext data contents over the network, there still seem to be quite a lot of differences between them. From the architecture of each protocol to how each are implemented to how they perform. A much more considerable difference is in the primary reason for HTTPS; securing http protocol. Despite all that, in this research work, we have seen how HTTP and HTTPS perform and has given us some points to note.

The following are main takeaways from this research work with regards to comparisons between HTTP and HTTPS implemented with TLS protocol:

- HTTP is better in terms of performance relative to HTTPS (with TLS implementation). Results from usage tests (Figures 5.1 and 5.2 and HTTP connection test results (Figures 5.6, 5.7 and 5.9) shows that HTTP performs better than HTTPS with TLS implementation. Figure 5.10 shows that this difference in performance is noticeable to human users of the system.

- TLS can be a direct cause to usability problems when longer keys are used. Results of this thesis work as seen in Tables 5.1, 5.3 and 5.2 has shown that

longer encryption keys take longer time to complete operation hence directly imposing delay to the system response time that becomes a bottleneck to usability of the system.

- TLS protocol performs better for shorter/small key sizes as compared to bigger/longer keys. Results of this thesis work as seen in Tables 5.1, 5.3 and 5.2 has shown that longer encryption keys take longer time to complete operation hence directly imposing delay to the system response time that becomes a bottleneck to usability of the system.

- TLS handshake operations generally takes longer than TLS record layer operations. Figures 5.6 and 5.7 shows t_Appconnect which implies TLS handshake has been observed to take much longer time as compared to difference between t_pretransfer and t_Appconnect which implies TLS record layer.

- There has to be a balance between security and usability as the two are closely related. This thesis work has found that usability of the system is directly affected by security imposed. To implement a much stronger security, usability has to be sacrificed since stronger encryption keys and algorithms take much longer time to finish.

5.6 Threats to Validity

Despite of conducting this thesis work with full caution about uncertainties in outcomes of the simulations, there are threats that would still affect validity of this thesis work and conclusions derived from its results. This section has grouped these threats according to papers [10] and [4] into 2 groups.

5.6.1 Construct Validity

This threat is concerned with the capacity at which the studied performance measures reflect what the researcher intended to investigate. To minimize this threat, problem definition, hypotheses and research questions of this thesis work were used so that as their solutions were found, a deeper understanding of the research subject would have been attained. Likewise, tests were performed to check solutions for these research questions and hypotheses.
Workshop was organized at the beginning of my thesis work so at to interact with the company employees and get their ideas and suggestions on different technological and methodological approaches currently being used in the industry. After applying ideas from employees and results achieved, these results were compared with the research questions, hypotheses and requirements to make sure that all of the intended goals were reached.

5.6.2 Reliability Validity

This validity aspect questions about an extent that results of the research work are dependent on the researcher. This poses a question, would results change if a different researcher conducted the same research?. This thesis work was performed under simulation environment, but it was done with a lot of iterations. Average values were recorded after thousands of repetitions so as to reduce incorrect readings. Also, these simulations were conducted on the same local machine so as to reduce network latency that has a possibility of causing uncertainties to the outcomes.

6
Conclusion

6.1 Summary

The aim of this thesis work was to research and analyse performance implication of TLS protocol when implemented in VEC. This work was aimed at developing a prototype of the system as a proof of concept and based on security requirement that TLS protocol has to be used; assess its performance implication in this specific setting of VEC. The main performance concern in this thesis work was to check difference in response time between implementations with and without TLS protocol configured. By the end of this thesis work, different performance tests were conducted, results were obtained and these results helped in understanding performance implication of TLS protocol when implemented in VEC.

6.2 Conclusions

TLS protocol is a very robust and proven to work protocol that offers security when transferring data across the Internet with a wide range of choices of algorithms and ciphers. But its flexibility has made it to be susceptible perception that it has performance drawbacks even in cases when it is not actually the source of the problem. This thesis work has seen how TLS protocol can perform really well or poorly depending on the **algorithm** used, **key size** used or **payload size** to be operated on. Results from this thesis work has shown how there are cases when TLS perform really well to the extent of having almost no effect to the total TCP response time.

Together with the above findings, it is necessary to have an idea on performance implication that TLS protocol has to the target system. This idea will inspire configuration and development of the system that has optimal performance in terms of response time. An open-source software like the one used in this this work could be a great tool to use to get this idea if it is developed further.

6.3 Future Work

There is a variety of things that could have been done in this project, but for various reasons like narrowing the scope, they had to be left out. The following are few propositions for future works based on this thesis work so as to make this contribution more robust. These things may either have an influence on the project as a whole or may be applicable to other people using the work and the ones that affects the results for:

- It would have been interesting to see how a specific cipher perform in terms of actual time set it takes for each unit operation. For example, how encryption comprises of a number of different iterations of XOR operations that are either done in electronic Code Block Chaining (CBC) mode or Cipher Feedback mode (CFB) and how long each takes so that can provide a much deeper understanding of these crypto algorithms.

- It would also have been interesting to find out relationship between security and power efficiency. In most cases, these embedded computers are very much expected to be power efficient, so it is vital to measure how does the two relate and see if there might be any conclusions to make regarding this.

- To complement an open source project [2] that came out as a result of this thesis work, it would have been very interesting to find other sets of test that could be added with other tests and bundle them up to make one complete software that performs tests for a anyone interested in knowing how their computers and web applications performs.

Bibliography

[1] CURL - online manual page. `https://curl.haxx.se/docs/manpage.html`. [ONLINE; accessed: 2016-04-26].

[2] HyperText Profiler - opensource http/s performance profiling tool. `https://github.com/bychwa/HyperTextProfiler`. [ONLINE; accessed: 2016-05-26].

[3] Claude Castelluccia, Einar Mykletun, and Gene Tsudik. Improving secure server performance by re-balancing ssl/tls handshakes. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 26–34. ACM, 2006.

[4] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research-an initial survey. In *SEKE*, pages 374–379, 2010.

[5] Chakravanti Rajagopalachari Kothari. *Research methodology: Methods and techniques*. New Age International, 2004.

[6] Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277. ACM, 1968.

[7] Farhad Moghimifar and Douglas Stebila. Predicting tls performance from key exchange performance: short paper. In *Proceedings of the Australasian Computer Science Week Multiconference*, page 44. ACM, 2016.

[8] Eric Rescorla. *SSL and TLS: designing and building secure systems*, volume 1. Addison-Wesley Reading, 2001.

[9] Ivan Ristic. *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. Feisty Duck, 2014.

[10] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131–164, 2009.

[11] William Stallings. Data and computer communications. *Prentice Hall*, 2005.

[12] William Stallings. *Cryptography and Network Security, 4/E*. Pearson Education India, 2006.

[13] Li Zhao, Ravi Iyer, Srihari Makineni, and Laxmi Bhuyan. Anatomy and performance of ssl processing. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 197–206. IEEE, 2005.

A

Testing Program Codes

A.1 Main Program: MainProfiler.java

```
1  package diadrom.profiler;
2
3  /**
4   *
5   * @author jaxonisack
6   */
7  import java.util.*;
8
9  public class MainProfiler {
10
11     public static void main(String args[]){
12
13         Scanner scan=new Scanner(System.in);
14         System.out.println("Welcome␣to␣Requests␣Profiler!,␣\n\n␣Choose␣Type:\n␣1:
       ␣Apache␣Benchmark␣\n␣2:␣Curl␣HTTP␣Profiler␣\n␣3:␣Usage␣Profiler␣\n");
15         System.out.print("My␣choice␣is:␣");
16         String profiler_type=scan.nextLine();
17
18         switch(profiler_type){
19             case "1":
20                     new ApacheProfiler();
21                 break;
22
23             case "2":
24                     new HyperTextProfiler();
25                 break;
26
27             case "3":
28                     new UsageProfiler();
29                 break;
30
31             default:
32
33                 System.out.println(profiler_type+"␣is␣a␣wrong␣choice!␣\n␣Bye!");
34                 System.exit(1);
35
36                 break;
37
38         }
39
```

```
    }
}
```

A.2 Runnable Program (Allows multi-threading): RunnableProfiler.java

```
import java.io.*;
import java.util.Date;

class RunnableProfiler implements Runnable {

      private Thread t;
      private String threadName,page;
      private int accuracy,payload_from,payload_to, interval;
      private String ofile,site,type,https_protocol,cipher_file;
      private int threadNumber=1;
      private Boolean secured;
      private String [] pages;
      private double tcp_start=0,tcp_end=0,tcp_time=0,micro_page_start=0,
    micro_page_end=0;
      private double[] tcp_micro_time;
      private File fout;
      private FileOutputStream fos;
      private OutputStreamWriter osw;

      RunnableProfiler(String type, String threadName,int accuracy, int
    threadNumber,Boolean secured, String[] pages, String ofile){

            //for Apache Benchmark
            this.threadNumber=threadNumber;
            this.type=type;
            this.threadName = threadName;
            this.accuracy=accuracy;
            this.pages=pages;
            this.ofile=ofile;
            this.secured=secured;

      }

      RunnableProfiler(String type, String threadName,int accuracy, int
    threadNumber, String site, Boolean secured, String https_protocol,String
    cipher_file, String ofile){

            //for Apache Benchmark
            this.threadNumber=threadNumber;
            this.type=type;
            this.threadName = threadName;
            this.accuracy=accuracy;
            this.site=site;
            this.https_protocol=https_protocol;
            this.cipher_file=cipher_file;
```

```
        this.ofile=ofile;
44      this.secured=secured;

46    }
      RunnableProfiler(String type, String threadName,int accuracy, int
    payload_from,int payload_to, int interval, String site, String ofile){
48
        this.type=type;
50      this.threadName = threadName;
        this.accuracy=accuracy;
52      this.payload_from=payload_from;
        this.payload_to=payload_to;
54      this.interval=interval;
        this.site=site;
56      this.ofile=ofile;


58
    }
60
    public void run() {
62
        TerminalCommandExecutor te=new TerminalCommandExecutor();
64
        switch(type){
66
            case "USAGE":
68
                    try {
70
                        fout = new File(ofile);
72                      fos = new FileOutputStream(fout);
                        osw = new OutputStreamWriter(fos);
74                      osw.write("SESSION:\t"+threadName+"\nNO␣PAGES:\t"+pages
    .length+"\n");

76                      for (int i=1; i <= accuracy; i++) {
                            osw.write("\tAccuracy␣Level:"+i+"\n");
78
                            tcp_start = new java.util.Date().getTime(); //start
     of the request
80
                            for(int j=0; j < pages.length; j++){
82                              micro_page_start = new java.util.Date().getTime
    (); //start of the request
                                page=secured?"https://"+pages[j]:"http://"+
    pages[j];
84                              String command="curl␣-o␣/dev/null␣--insecure␣-s
    ␣-w␣%{time_connect},%{time_starttransfer},%{time_total},%{time_appconnect},%{
    time_namelookup},%{time_pretransfer},%{time_redirect}␣"+page;
                                String results=te.runCommand(command,true);
86                              micro_page_end = new java.util.Date().getTime()
    ; //start of the request
                                osw.write("\t\tPAGE:␣"+pages[j]+"\t\t\t␣TIME:"
    +(micro_page_end-micro_page_start)/1000+"␣seconds\n");
88
                            }
```

```
90
                            tcp_end = new java.util.Date().getTime();    //end
    of the request
92
                            tcp_time +=(tcp_end - tcp_start);
94                          osw.write("\n");
                        }
96
                        System.out.println(threadName+","+pages[0]+",␣"+String.
    format("%.8f",(tcp_time/accuracy)/1000)+"␣seconds,␣"+new java.util.Date().
    getTime());
98
                        osw.write("AVERAGE␣TIME:\t\t\t"+String.format("%.8f",(
    tcp_time/accuracy)/1000)+"␣seconds\n");
100
                        osw.close();
102
                    }catch(Exception e){
104
                        System.err.println("An␣error␣has␣occured␣while␣opening␣
    /␣writing␣to␣the␣file␣"+ofile);
106
                    }
108
                break;
110
            case "CURL":
112
                try {
114
                        fout = new File(ofile);
116                     fos = new FileOutputStream(fout);
                        osw = new OutputStreamWriter(fos);
118
                        System.out.println("ThreadName,␣t_PayloadSize,␣
    t_NameLookup,␣t_AppConnect,␣t_Redirect,␣t_Connect,␣t_Pretransfer,␣t_Transfer,
    ␣t_Total,␣t_OveralTotal,␣t_Timestamp");
120                     osw.write("ThreadName,␣t_PayloadSize,␣t_NameLookup,␣
    t_AppConnect,␣t_Redirect,␣t_Connect,␣t_Pretransfer,␣t_Transfer,␣t_Total,␣
    t_OveralTotal,␣t_Timestamp"+"\n");
122                     for(int i=Integer.valueOf(payload_from); i <= Integer.
    valueOf(payload_to); i+=interval){
124                         String num_bytes=String.valueOf(i);
126                         String command="curl␣-o␣/dev/null␣--insecure␣-s␣-
    w␣%{time_connect},%{time_starttransfer},%{time_total},%{time_appconnect},%{
    time_namelookup},%{time_pretransfer},%{time_redirect}␣"+site+"/get_response?
    size="+num_bytes;
128                         double t_connect=0,t_transfer=0,t_total=0,
    t_appconnect=0,t_namelookup=0,t_pretransfer=0,t_redirect=0;
130                         for(int j=1; j<=accuracy;j++){
```

```
132                                     tcp_start = new java.util.Date().getTime
    (); //start of the request
                                    String results=te.runCommand(command,true);
134                                     tcp_end = new java.util.Date().getTime();
       //end of the request

136                                     tcp_time+=(tcp_end - tcp_start);

138                                     String[] resultsArray=results.split(",");

140                                     t_connect+=Double.valueOf(resultsArray
    [0]);
                                        t_transfer+=Double.valueOf(resultsArray
    [1]);
142                                     t_total+=Double.valueOf(resultsArray[2]);
                                        t_appconnect+=Double.valueOf(resultsArray
    [3]);
144                                     t_namelookup+=Double.valueOf(resultsArray
    [3]);
                                        t_pretransfer+=Double.valueOf(
    resultsArray[3]);
146                                     t_redirect+=Double.valueOf(resultsArray
    [3]);

148                                 }

150                                 System.out.println(threadName+",␣"+num_bytes+",␣"
    +String.format( "%.8f",(t_namelookup/accuracy))+",␣"+String.format( "%.8f",(
    t_appconnect/accuracy))+",␣"+String.format( "%.8f",(t_redirect/accuracy))+",␣
    "+String.format( "%.8f",(t_connect/accuracy) )+",␣"+String.format( "%.8f",(
    t_pretransfer/accuracy))+",␣"+String.format( "%.8f",(t_transfer/accuracy))+",
    ␣"+String.format( "%.8f",(t_total/accuracy))+","+String.format("%.8f",(
    tcp_time/accuracy)/1000)+",␣"+new java.util.Date().getTime());
                                    osw.write(threadName+",␣"+num_bytes+",␣"+String.
    format( "%.8f",(t_namelookup/accuracy))+",␣"+String.format( "%.8f",(
    t_appconnect/accuracy))+",␣"+String.format( "%.8f",(t_redirect/accuracy))+",␣
    "+String.format( "%.8f",(t_connect/accuracy) )+",␣"+String.format( "%.8f",(
    t_pretransfer/accuracy))+",␣"+String.format( "%.8f",(t_transfer/accuracy))+",
    ␣"+String.format( "%.8f",(t_total/accuracy))+","+String.format("%.8f",(
    tcp_time/accuracy)/1000)+",␣"+new java.util.Date().getTime()+"\n");
152
                            }
154
                            System.out.println("------END-----");
156
                            osw.close();
158
                        }catch(Exception e){
160                         System.err.println(e);
                        }
162                 break;

164             case "APACHE":

166                 if(secured){
```

```
168                         System.out.println("\n Safe HTTP Test:\n ");

170                         try{

172                             String cipher;

174                             BufferedReader br = new BufferedReader(new
    InputStreamReader(new FileInputStream(cipher_file)));

176                             while ((cipher = br.readLine()) != null)   {

178                                 String command="ab -d -k -c "+this.threadNumber+"
     -n "+accuracy+ " -f "+https_protocol+" -Z"+cipher+" "+site;

180                                 String results=te.runCommand(command,true);

182                                 String[] lines = results.split(System.getProperty
    ("line.separator"));

184                                 if(lines.length<=30){

186                                     System.out.println(cipher+" : "+" Error");

188                                 }else{
                                        System.out.println(cipher+" :");
190                                     System.out.println("\t"+lines[8]);
                                        System.out.println("\t"+lines[9]);
192                                     System.out.println("\t"+lines[10]);
                                        System.out.println("\t"+lines[11]);
194                                     System.out.println(" ");
                                        System.out.println("\t"+lines[31]);
196                                     System.out.println("\t"+lines[32]);
                                        System.out.println(" ");

198
                                    }

200

202                             }
                                br.close();
204
                        }catch(Exception e){
206
                            System.err.println("An error occurred due to reading
    the cipher file!");
208
                        }
210
                    }else{
212
                        System.out.println("\n UnSafe HTTP Test:\n ");
214
                        String command="ab -d -k -c "+this.threadNumber+" -n "+
    accuracy+ " "+site;
216
                        String results=te.runCommand(command,true);
218
```

```
                        String[] lines = results.split(System.getProperty("line.
    separator"));
220
                        if(lines.length<=30){
222
                            System.out.println(">>"+"␣Error");
224
                        }else{
226
                            System.out.println("\t"+lines[8]);
228                         System.out.println("\t"+lines[9]);
                            System.out.println("\t"+lines[10]);
230                         System.out.println("\t"+lines[11]);
                            System.out.println("␣");
232                         System.out.println("␣");
                        }
234
                    }
236
                    break;
238
                }
240
        }
242
        public void start ()
244     {
           if (t == null){ t = new Thread (this, threadName); t.start (); }
246     }

248 }
```

A.3 Usage Testing Program : UsageProfiler.java

```
2 package diadrom.profiler;

4 /**
   *
6  * @author jaxonisack
   */
8 import java.util.*;

10 public class UsageProfiler {

12      private final String RESULTS_FOLDER_PATH="results/";
        private final String TEST_TYPE="USAGE";
14
        public UsageProfiler(){
16
            System.out.println("Welcome␣to␣Usage␣Profiler");
18
            Scanner scan=new Scanner(System.in);
```

```
20
        System.out.print("Number␣of␣Site␣Pages␣per␣normal␣usage:␣"); int
    num_pages=Integer.valueOf(scan.nextLine());
22
            String [] pages=new String[num_pages];
24          for (int i=0;i < num_pages;i++ ) {
                System.out.print("Page␣"+i+":"); pages[i]=scan.nextLine();
26          }

28      System.out.print("Test␣both␣http␣and␣https␣(Y/y␣or␣N/n):␣"); String
    test_both=scan.nextLine();
        String https_protocol="", cipher_file="";
30      System.out.print("Number␣of␣Threads:␣"); int num_threads=Integer.valueOf(
    scan.nextLine());
        System.out.print("Accuracy␣Level␣(1-100):␣"); int accuracy=Integer.
    valueOf(scan.nextLine());
32      System.out.print("Output␣filename:␣"); String outputfilename=scan.
    nextLine();

34      RunnableProfiler s_thread,u_thread;
        String results_file_name="untouched.csv";
36      Boolean secured=false;

38      if(test_both.equals("Y")|| test_both.equals("y")){

40          for(int p=1; p<=num_threads; p++){
                //unsafe thread
42              secured=false;
                results_file_name=RESULTS_FOLDER_PATH+outputfilename+"
    _unsafe_thread_"+p+"_.csv";
44              u_thread = new RunnableProfiler(TEST_TYPE,"UnsafeThread",
    accuracy,num_threads,secured,pages,results_file_name);
                u_thread.start();
46          }
            for(int p=1; p<=num_threads; p++){
48              //safe thread
                secured=true;
50              results_file_name=RESULTS_FOLDER_PATH+outputfilename+"
    _safe_thread_"+p+"_.csv";
                s_thread = new RunnableProfiler(TEST_TYPE,"SafeThread", accuracy,
    num_threads,secured,pages,results_file_name);
52              s_thread.start();
            }
54
        }else{
56          for(int p=1; p<=num_threads; p++){
                //unsafe thread
58              secured=false;
                results_file_name=RESULTS_FOLDER_PATH+outputfilename+"
    _unsafe_thread_"+p+"_.csv";
60              u_thread = new RunnableProfiler(TEST_TYPE,"UnsafeThread",
    accuracy,num_threads,secured,pages,results_file_name);
                u_thread.start();
62          }
        }
64  }
```

```
}
```

A.4 HTTP Connection Testing Program : HyperTextProfiler.java

```
package diadrom.profiler;

/**
 *
 * @author jaxonisack
 */
import java.util.*;

public class HyperTextProfiler {
    private final String RESULTS_FOLDER_PATH="results/";
    private final String TEST_TYPE="CURL";

    public HyperTextProfiler(){

        System.out.println("Welcome to HTTP Profiler (with CURL)");

        Scanner scan=new Scanner(System.in);

        System.out.print("Website: "); String website=scan.nextLine();
        System.out.print("Test both http and https (Y/y or N/n): "); String
    test_both=scan.nextLine();
        System.out.print("Number of Threads: "); int num_threads=Integer.valueOf(
    scan.nextLine());
        System.out.print("Accuracy Level (1-100): "); int accuracy=Integer.
    valueOf(scan.nextLine());
        System.out.print("Payload From (Bytes): "); int payload_from=Integer.
    valueOf(scan.nextLine());
        System.out.print("Payload To (Bytes): "); int payload_to=Integer.valueOf(
    scan.nextLine());
        System.out.print("Interval Payload (Bytes): "); int interval=Integer.
    valueOf(scan.nextLine());
        System.out.print("Output filename: "); String outputfilename=scan.
    nextLine();

        String safe_site="https://"+website;
        String unsafe_site="http://"+website;
        String results_file_name="untouched.csv";

        RunnableProfiler r_thread;

        if(test_both=="y" || test_both=="Y"){

            for(int i=1; i<=num_threads; i++){

                results_file_name=RESULTS_FOLDER_PATH+outputfilename+"
    _unsafe_thread_"+i+"__.csv";
```

```
                r_thread = new RunnableProfiler(TEST_TYPE,"UnsafeThread-"+i,
    accuracy,payload_from,payload_to,interval,unsafe_site,results_file_name);
40              r_thread.start();

42          }
            for(int i=1; i<=num_threads; i++){
44
                results_file_name=RESULTS_FOLDER_PATH+outputfilename+"
    _safe_thread_"+i+"__.csv";
46              r_thread = new RunnableProfiler(TEST_TYPE,"SafeThread-"+i,
    accuracy,payload_from,payload_to,interval,safe_site,results_file_name);
                r_thread.start();
48
            }
50
        }else{
52
            for(int i=1; i<=num_threads; i++){
54
                results_file_name=RESULTS_FOLDER_PATH+outputfilename+"
    _unsafe_thread_"+i+"__.csv";
56              r_thread = new RunnableProfiler(TEST_TYPE,"UnsafeThread-"+i,
    accuracy,payload_from,payload_to,interval,unsafe_site,results_file_name);
                r_thread.start();
58
            }
60
        }
62  }

64 }
```