SECURITY ANALYSIS OF COAP AND DTLS PROTOCOLS FOR INTERNET OF THINGS APPLICATIONS

ALİ TUNCA GÜRKAN
B.S., Computer Engineering, IŞIK UNIVERSITY, 2015

Submitted to the Graduate School of Science and Engineering
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Engineering

IŞIK UNIVERSITY
2019

IŞIK UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

SECURITY ANALYSIS OF COAP AND DTLS PROTOCOLS FOR INTERNET OF THINGS APPLICATIONS

ALİ TUNCA GÜRKAN

APPROVED BY:

Assist. Prof. Dr. Ayşegül TÜYSÜZ ERMAN Işık University
(Thesis Supervisor)

Prof. Dr. Olcay Taner YILDIZ Işık University

Assist. Prof. Dr. Yonca BAYRAKDAR Çanakkale Onsekiz Mart University

APPROVAL DATE: 26/08/2019

SECURITY ANALYSIS OF COAP AND DTLS PROTOCOLS FOR INTERNET OF THINGS APPLICATIONS

Abstract

Internet of Things is a very fast growing area. Its requirements and related technologies are changing from day to day. In Internet of Things, devices can communicate with each other with different messaging protocols. The latest messaging protocols are well developed, but they are too heavy to be run on devices developed with old technology. Therefore, these devices have to be operated with old-fashioned protocols. This makes devices vulnerable to security risks.

CoAP is a newly developed messaging protocol for constrained devices used in Internet of Things applications. The protocol is a variant of HTTP, so it has similar specifications. CoAP does not have an embedded security mechanism. Therefore, another protocol called DTLS is used on top of it to provide security. DTLS has powerful functions like handshaking and session processes; however, it is weak against DoS attacks.

In this study, we develop a security extension for Internet of Things devices using CoAP with DTLS for secure messaging. DTLS applies handshaking process for every received request. The handshaking process is the most time and resource consuming part of the communication. We propose a security extension to prevent unnecessary messaging during handshaking process of an attacker device that sends a lot of unauthenticated requests. When a client sends requests to a server that has the proposed security extension, the server counts unsuccessful handshaking processes for each client. If the count passes a limit of suspicious requests, the security extension on server adds the client's IP address into a banned IPs list. Until the expiration time, the server does not accept any request from the banned IP address.

Our proposed security extension is tested in different scenarios to examine the effects on the network. The results of the experiments show that the enhanced security extension decreases delays on the network and it is helpful for communication between authenticated devices.

Keywords: IoT, CoAP, DTLS, DoS, security

NESNELERİN İNTERNETİ UYGULAMALARI İÇİN COAP VE DTLS PROTOKOLLERİNİN GÜVENLİK ANALİZİ

Özet

Nesnelerin İnterneti çok hızlı büyüyen bir alan. Bu alanın gereksinimleri ve ilgili teknolojiler günden güne değişiyor. Nesnelerin İnterneti'nde, cihazlar birbirleri ile farklı mesajlaşma protokolleri ile iletişim kurabilir. En yeni mesajlaşma protokolleri iyi geliştirilmiştir, ancak eski teknolojiyle geliştirilen cihazlarda uygulanamayacak kadar ağırdırlar. Bu yüzden bu cihazlar eski tür protokoller ile çalıştırılmak zorunda kalıyorlar. Bu durum, cihazları güvenlik risklerine karşı savunmasız hale getiriyor.

CoAP, Nesnelerin İnterneti uygulamalarında kullanılan kısıtlı cihazlar için yeni geliştirilmiş bir mesajlaşma protokolüdür. Protokol HTTP'nin bir çeşididir, bu yüzden benzer özelliklere sahiptir. CoAP'ın yerleşik bir güvenlik mekanizması yoktur. Bu nedenle, DTLS adında başka bir protokol, güvenliği sağlamak için kullanılır. DTLS, el sıkışma ve oturum işlemleri gibi güçlü işlevlere sahiptir, ancak DoS saldırılarına karşı zayıftır.

Bu çalışmada, güvenli mesajlaşma için DTLS ile CoAP kullanan Nesnelerin Interneti cihazları için bir güvenlik uzantısı geliştirdik. DTLS, alınan her istek için el sıkışma işlemini uygular. El sıkışma süreci, iletişimin en çok zaman alan ve kaynak harcayan kısmıdır. Kimliği doğrulanmamış ve birçok istek gönderen bir saldırganın bu işlem sırasında oluşturduğu gereksiz mesajlaşmayı önlemek için bir güvenlik uzantısı öneriyoruz. Bir istemci önerilen güvenlik uzantısına sahip bir sunucuya istek gönderdiğinde, sunucu her istemci için başarısız el sıkışmalarını sayar. Sayı bir şüpheli istek sınırını geçerse, sunucudaki güvenlik uzantısı istemcinin IP adresini bir yasaklanmış IP adresleri listesine ekler. Zaman aşımına kadar, sunucu yasaklanan IP adresinden herhangi bir istek kabul etmez.

Önerilen güvenlik eklentisinin ağ üzerindeki etkilerini incelemek için eklenti farklı senaryolarda test edilmiştir. Bu testlerin sonuçları, geliştirilmiş güvenlik protokolünün ağdaki gecikmeleri azalttığını ve kimliği doğrulanmış cihazlar arasındaki iletişimde yardımcı olduğunu göstermektedir.

Anahtar kelimeler: IoT, CoAP, DTLS, DoS, güvenlik

Acknowledgements

I would like to thank my thesis supervisor Assist. Prof. Dr. Ayşegül TÜYSÜZ ERMAN, who has been standing by me since the beginning of preparing my graduate thesis and continues to help me patiently in my mistakes.

I am thankful and fortunate enough to get constant encouragement, support and guidance from Büşra Şen and Onur Açıkgöz.

To My Family…

Table of Contents

List of Tables

List of Figures

List of Abbreviations

IoT	**I**nternet **o**f **T**hings
CoAP	**Co**nstrained **A**pplication **P**rotocol
DTLS	**D**atagram **T**ransport **L**ayer **S**ecurity
MQTT	**M**essage **Q**ueuing **T**elemetry **T**ransport
AMQP	**A**dvanced **M**essage **Q**ueuing **P**rotocol
DoS	**D**enial **o**f **S**ervice
DDS	**D**ata **D**istribution **S**ervice
REST	**RE**presentational **S**tate **T**ransfer
TLS	**T**ransport **L**ayer **S**ecurity
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
HTTP	**H**ypertext **T**ransfer **P**rotocol
6LowPan	IPv**6** over **Low**-Power Wireless **P**ersonal **A**rea **N**etworks
DNS	**D**omain **N**ame **S**ystem
NTP	**N**etwork **T**ime **P**rotocol

Chapter 1

Introduction

IoT (Internet of Things) is connecting different kinds of devices with each other by Internet or any other means of networking to monitor and control an environment remotely. Low-power computers, sensors, embedded devices are example for these devices. IoT is one of the fastest growing area in computer science. It reaches many different kinds of market in the world such as environmental monitoring [1], healthcare [2], smart cities [3], etc. According to J. Jeon, 75 billion IoT devices will be get connected in 2020 [4].

One of the critical issues about IoT is to ensure that IoT devices, the network, and data are secure. These commonly used devices would be under attacks of hackers. In IoT ecosystem, devices connect to each other. If a hacker machine can reach a single device, it can connect (send/receive fake messages) to all the other devices in the same network and can hack other devices, thus the whole network.

There are many powerful and commonly used network security protocols and standards for popular computing devices such as PCs and smartphones. Day by day, computer and Internet security companies update their security products (antivirus, firewall etc.) or develop new methods for new threats. However, IoT devices are not able to adapt these developments rapidly due to their constraints. By their nature, IoT devices are low-cost products and have limited resources such as limited memory, battery and processor. Therefore, their hardware specialities are insufficient to run intensive security protocols or software. The IoT technology follows the recent security related developments from behind. In order to close the gap, security protocols for every IoT devices/applications should be standardized.

On the other hand, there are many standardized communication protocols when we examine IoT from the networking perspective. CoAP [5], MQTT [6], AMQP [7] and DDS [8] are four major IoT protocols providing mechanisms for asynchronous communication between IoT devices. CoAP and MQTT are very dominantly used in most of the IoT applications. A server and a client communicate as one-to-one with each other in CoAP. In MQTT, multiple clients communicate with a central server as many-to-many communication. AMQP is enterprise-scale asynchronous messaging protocol. It consists of publisher, consumer and broker. Publisher and consumer communicate with each other through a queue established on a broker [9]. DDS is a middleware protocol that uses publisher-subscriber data transfer model [10]. CoAP (Constrained Application Protocol) uses REST [11] software architecture. Since the REST architecture is useful for web services, CoAP has an advantage over other protocols. Therefore, it is highly preferable by developers. However, none of those protocols have any build-in security mechanisms.

While examining studies on IoT application security, it is seen that there are two most commonly compared methods: TLS [12] and DTLS [13]. TLS is the first protocols proposed for providing security of a network traffic with transparent connection-oriented channel. It requires a reliable transport channel as TCP. DTLS is a variant of TLS. The aim of DTLS is to provide a secure communication over an unreliable datagram traffic with UDP [13]. These protocols provide authentication, key exchange and secure communication between IoT devices. They are suitable to be implemented on resource constrained IoT devices since they can work in low-power consumption mode and do not need powerful hardware specifications to run.

In this thesis, we focus on security flaw of the most commonly used IoT communication protocols: DTLS running on top of CoAP. The protocol CoAP is efficient for nowadays constrained devices; therefore, it is expected to be the most preferable protocol at the near future. For different kinds of IoT applications, different communication protocols can be used. However, the protocol, which needs to be run on IoT devices, can not be a heavy protocol like HTTP with its security plug-in due to their limited resources. Although CoAP is a popular light-weight protocol for IoT devices, it needs an efficient security solution. DTLS is suitable security protocol to provide security for CoAP. However, DTLS has a weak point against DoS attacks according to its specification. In a DoS attack, attackers send too many requests to overwhelm server resources and decrease the server's ability to reply legitimate requests. The attack creates an inoperative server as a result of overloading.

In our study, we overcome the previously mentioned problem with an enhanced security extension that is implemented on top of DTLS. The extension blocks specific requests similar to DoS attacks. It measures incoming requests to detect DoS attackers on the server side. We adapt our extension on a well-implemented library called Californium [14]. The Eclipse Californium platform provides an open source implementation of CoAP and DTLS 1.2 [13]. We extend DTLS implementation in Californium library with a new IP checking mechanism. The Californium library is suitable for running on devices such as Raspberry Pi and Arduino. Thus, we test our proposed extension on real devices in a real testbed using Raspberry Pi. As performance metrics, the delay of secured communication and the effects of unsuccessful handshaking processes during DoS attacks for every devices are considered. Our implementation has adaptable parameters to test different cases. Thus, we examine performance of devices with the enhanced security extension for different attacking scenarios.

This thesis is organized as follows. In Chapter 2 we will introduce related works, CoAP and its comparison with other protocols, DTLS, and the Californium platform. We will give the details of our proposed security extension in Chapter 3, including details of the proposed algorithms, and the implementation of the test platform. We will present our experimental results in Section 4 and conclude our work in Section 5.

Chapter 2

Background

2.1 Related Works

In a survey conducted by Bagci et al. [15], it was found that describes the Fusion architecture and application which is developed on IPsec and DTLS systems. They are described as Fusion's implementation and evaluate its storage overheads, communication performance, and energy consumption. According to the research's results, applying the coalesced solution to use the standardized IoT security protocols IPsec and DTLS is usable. Their implementation and evaluation show that it is applicable on implemented solutions in constrained devices, and the combined solution has better performance than performing cryptographic operations separately.

In their research by Sitenkov [16], investigate the access control solution for the IETF standard draft CoAP, using the DTLS protocol for transport security. They used the centralized approach to save access control information in the framework. Public key cryptography operations is computationally heavy for constrained devices. Therefore, they build the new solution based on symmetric cryptography. Evaluation results show that the access control framework increases the computational effort of the handshake by 6.0%, increases the code footprint of the Datagram Transport Layer Security implementation by 7.9% and has no effect on the overall handshake time.

Raza S. et. al. [17] researched about integration of DTLS and CoAP for the IoT which name is Lithe. They proposed a new implemented DTLS header compression scheme. The new header has gains to reduce energy consumption by leveraging the 6LoWPAN standard. The proposed DTLS header compression scheme does not compromise the end-to-end security properties provided by DTLS. Simultaneously, it considerably reduces the number of transmitted bytes while maintaining DTLS standard compliance. According to evaluated results, the new approach get precious achievements in terms of packet size, energy consumption, processing time, and network-wide response times with compressed DTLS. They quantitatively show that DTLS can be compressed and its overhead is significantly reduced using 6LoWPAN standardized mechanisms. Their implementation and evaluation of compressed DTLS is successful to reduce the CoAPs overhead as the DTLS compression is efficient in terms of energy consumption and network-wide response time when compared with plain CoAPs. The difference between compressed DTLS and uncompressed DTLS is very significant if the use of uncompressed DTLS results in 6LoWPAN fragmentation.

Rahman R et al. [18] examined IoT protocols and their security risks. Especially, they are concentrated on CoAP protocol. They analyzed architecture, implementation and comparison with HTTP for CoAP. Nevertheless, they worked on CoAP's security challenges and solutions. According to the analysis, providing security by DTLS with high performance is a considerable challenge for CoAP.

Kothmayr et al. [19] research first fully implemented two-way authentication security with DTLS for IoT. Their proposed security scheme is based on RSA which is one of the most used public key cryptography algorithms. According to the results, DTLS is achievable security solutions for IoT systems. The handshake process with RSA is consumed 488 mj. Also, the system needs less than 20 KB memory. The value is under their sensor nodes which require 48 KB.

2.2 Constrained Application Protocol (CoAP)

CoAP (Constrained Application Protocol) [5] was proposed by Shelby et al. in 2015. HTTP and HTTPS are not proper for IoT and M2M applications [20] due to their high messaging overheads. That's why CoAP was developed for constrained devices and networks to achieve lower communication overheads [5]. Indeed, CoAP is a lightweight variant of HTTP. They have the same principles; however, CoAP is simplified for resource constrained devices such as sensor nodes that need to consume lower energy to operate longer [18]. CoAP protocol is relatively new in communication networks and it has some weak points such as security issues. It does not have a standardized security mechanism yet [18].

In the rest of this section, CoAP design details, features and comparison with similar protocols are given.

2.2.1 Design Principles

CoAP protocol has the following design principles:

- It is designed to decrease the size of message overhead
- It can communicate asynchronously and uses UDP
- It has different message types: Confirmable, Non-confirmable, Acknowledgement and Reset. Reliability of a message is controlled by the first two message types.

2.2.2 Features

The most important features of CoAP protocol are listed below:

- Supporting UDP integration/binding for unicast and multicast requests

- Content-type is available in the request header

- Uniform Resource Identifier (URI) is supported

- Proxy and caching mechanism abilities

- Supporting Datagram Transport Layer Security (DTLS) to provide security

- CoAP has a simple binary format. The format includes header (fixed-size 4-bytes), variable-length token values (between 0 to 8-bytes).

- CoAP shares same response code definitions with HTTP

2.2.3 CoAP vs. HTTP

HTTP is the main protocol to provide data delivery on the Internet (expanding its interface to support HTTP/HTTPS protocol). CoAP is developed as an alternative to HTTP for constrained devices. Therefore, they use very similar infrastructures and processes. These protocols uses the same methods of REST structure [18]. Besides that, HTTP is old and widely known but CoAP is very new and still continue to develop.

The priority in CoAP design is being lightweight for both constrained devices and networks. Therefore, CoAP has significantly low communication overhead than HTTP. For a same type of transaction, CoAP with UDP creates a transaction with 8 to 10 times less bytes than HTTP with TCP creates. [21].

2.2.4 CoAP vs. MQTT

There are different kinds of message exchanging protocols designed based on requirements of IoT systems. MQTT (Message Queuing Telemetry Transport Protocol) is also very popular and used in lots of IoT applications. MQTT [6] is another M2M communication protocol developed in 1999. The protocol is based

Criteria	**CoAP**	**MQTT**
Transport Layer	UDP	TCP
Architecture	Client/Broker	Client/Server Client/Broker
Header Size	4 Byte	2 Byte
Message Size	Max 256 MB	Max dependent to programming technology
Methods	Connect, Disconnect, Publish, Subscribe, Unsubscribe, Close	Get, Post, Put, Delete
Cache and Proxy	Partial	Yes
Security	TLS/SSL	UDP, SCTP
Default port	1883/883 (TLS/SSL)	5685,5684 (UDP/DTLS)

Table 2.1: Comparison of CoAP and MQTT.

on publish-subscribe messaging and is also designed to mitigate the communication overhead in M2M communication for constrained environments.

Table 2.1 shows the comparison of CoAP and MQTT protocols. One of the important differences between CoAP and MQTT is about transport protocols used underneath. MQTT uses TCP and CoAP uses UDP. The differences affect the reliability of these messaging protocols [22]. TCP can guarantee packet delivery to target; on the other hand, UDP can not [23] guarantee message delivery.

Both CoAP and MQTT are binary protocols. MQTT has 2-byte header size and it has the half size of CoAP's header. The message payloads can be maximum 256 MB for CoAP, and the maximum payload value can change according to the web server or programming technology for MQTT. Although header size is lower in MQTT than CoAP, TCP connection increases overall overhead and message size in MQTT [23]. Moreover, CoAP achieves better values for bandwidth requirements and round trip time (RTT) when it is compared with MQTT [24].

type	version	epoch	sequence number	length	fragment
1 byte	2 bytes	2 bytes	6 bytes	2 bytes	variable

Table 2.2: DTLS Record Format.

2.3 Datagram Transport Layer Security (DTLS)

TLS is a well-developed security protocol, which is generally used on web traffic and e-mail protocols [13]. TLS run on top of TCP, so it can achieve reliable connection. DTLS is an alternative to TLS designed for applications running on top of unreliable datagram protocols such as UDP [25]. DTLS commonly uses the infrastructure of TLS. As mentioned above, TLS does not support requirements of an unreliable communication. DTLS completes these requirement with least possible changes [13].

Table 2.2 shows the record format of DTLS. The fields type, version, length include basic information about a record. Epoch and sequence number are significant for handshake functionality. Epoch starts from 0, and increases. Sequence number is a unique value, and it is used with Epoch to calculate Message Authentication Code for each DTLS record.

2.3.1 Handshaking

In Figure 2.1, handshake process of DTLS between a client and a server is shown [19]. First of all, the client sends "*ClientHello*" message that contains the protocol version. If the cipher suite is one of the supported version on the server, the process will continue. If the process is successful, the server sends a response with "*ServerHello*" message with the supported cipher suite. After that, the server continues to send a message called "*CerfiticateRequest*" with X.509 certificate. After this message, the server sends "*ServerHelloDone*" message to complete the flight. In Flight 5, the client sends encrypted keys. it starts to send its certificate information on "*ClientKeyExchange*". After the message containing

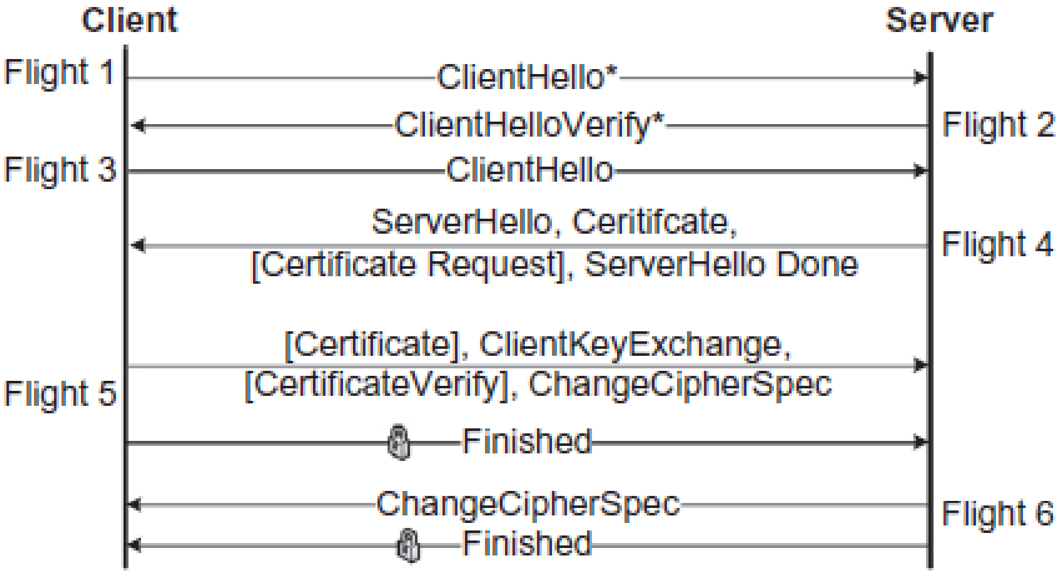
Client
Server
Flight 1
ClientHello*
ClientHelloVerify*
Flight 2
Flight 3
ClientHello
ServerHello, Ceritifcate,
[Certificate Request], ServerHello Done
Flight 4
[Certificate], ClientKeyExchange,
[CertificateVerify], ChangeCipherSpec
Flight 5
Finished
ChangeCipherSpec
Finished
Flight 6

Figure 2.1: DTLS Handshake Process.

the encrypted pre-master secret with keys is received by the server, the client sends "*CertificateVerify*" message to provide authentication itself. The message shows that the server's private key matched the client's public key. Afterward, the client sends "*ChangeCipherSpec*" message, and it shows that the client encrypted all messages successfully with the cipher suite and these keys. Finally, the client sends "*Finished*" message, and it contains the encrypted version of all previous handshaking messages. In last flight, the server sends "*ChangeCipherSpec*" and "*Finished*" messages. Finally, they complete the handshake process with this message.

Handshaking in DTLS protocol is very important to start a secure communication between a client and a server. If this procedure fails, they can not communicate with each other.

2.4 Eclipse Californium Platform

CoAP has been implemented in different programming languages. Californium [14] platform contains one of the well-developed Java implementations of CoAP protocol. Indeed, Californium is a very wide range library and still under development.

The library is under Eclipse Distribution License and Eclipse Public License.

The library has both server side and client side implementations. Unlike other CoAP implementations, Californium library has sub-projects to support all features of the protocol. When the source directory is examined, Californium-core, Scandium-core, and sub-directories of the projects can be seen. Californium-core includes core implementations of CoAP and related implementations. Scandium-core is sub-project of Californium to provide security and DTLS version 1.2 implementation resides in this package.

Californium platform has some certain benefits when it is compared with other implementations of CoAP. It has many well-implemented example codes for testing. Setting up a scenario and running it is very easy and quick. For testing purposes, the platform also provides a useful plugin called Copper [26], which is designed to act as a CoAP user-agent to test CoAP communication. It runs on Firefox Browser; however, it is only available in Linux. Moreover, according to Kovatsch et al. in [27], Californium implementation achieves three times higher throughput than other CoAP implementations. This is one of the most significant reasons to use Californium platform in this work.

Urkia et al. compare up-to-date open source CoAP implementations in [28]. They consider technical specifications, target platforms, extensions and other dioristic properties. The features of different implementations are summarized in Table 2.3. The authors set an experiment with the given CoAP implementations. Server and client codes are run on RaspberryPi devices. Each client and server, which are deployed for every implementation, send 50 requests to each other. According to their experiment, Californium has less RTT values than Python-based libraries have, but longer RTT values than C-based libraries achieves. On the other hand, Californium is the library that consumes the RAM the most. C-based libraries has better values on CPU and RAM usage to compile and execute the protocol than the others have. In larger scenarios, libraries implemented by Java, Python and Nodejs achieve similar performance to C-based libraries.

Library	**Language**	**Target Platform**	**Client/Server**	**Extension**
libcoap	C	POSIX, Contiki, IwIP, TinyOS	Client & Server	Observe, Block-wise, Resource directory
smcp	C	Embedded devices, Linux	Client & Server	Observe, Multicase
microcoap	C	Arduino, POSIX	Server	-
FreeCoAP	C	GNU or Linux	Client & Server	-
Californium	Java	JVM	Client & Server	Observe, Blockwise Resorce Directory
h5.coap node-coap CoAPython	JS JS Python	Nodejs Nodejs Nodejs	Client Client & Server Client & Server	Observe, Blockwise resource, Core-link multicast
CoAPy	Python	Python	Client & Server	Multicast, Blockwise

Table 2.3: Features of CoAP libraries.

Chapter 3

DTLS Protocol with Proposed Security Extension

3.1 Structure

As it is mentioned in the previous chapter, Californium includes an open-source implementation of CoAP protocol. The library has a modular structure. It includes five sub-packages: californium-core, californium-osgi, californium-proxy, element-connector and scandium-core. In our work, we have modified some classes on Californium-core and Scandium-core packages to improve security perspective of DTLS version 1.2 protocol.

Californium has many skills for CoAP based projects and related progress and protocols. Californium-core includes basic functionalities about CoAP. In addition, when a server or a client is created, it is initialized by californium-core. Scandium-core is completely about security and includes DTLS implementation. It applies 1.2 version of DTLS. Therefore, we improve the security against DoS attacks on that version. Since the modules include detailed and well-implemented demo classes, it is an advantage to test our proposed model in such a complex project.

Californium has maven as project structure. It provides convenience to check downloaded libraries' version and keep the project up-to-date.

3.2 CoapServer

CoapServer is a class, which includes all CoAP implementations of server side in californium-core module. The class has network and security configurations, listening port information and message deliverer to income CoAP request from a source. If adding multiple end point information to *CoapServer*, the server can listen multiple ports of these end points.

3.3 CoapClient

CoapClient is a class, which includes all CoAP implementations of client side in californium-core module. Unlike *CoapServer*, it does not has message deliverer. In addition, it has timeout parameter to control send request status. The client can send request to just a single resource. Therefore, it does not support to add multiple end point information.

3.4 Configuration

When we add DTLS to our CoAP server or CoAP client, we need to define security configuration on these nodes. There are four settings to make successful DTLS configuration: Psk Store, Trust Store, Identity, and Supported Cipher Suites.

In scandium-core module, there are helper classes to configure security properties. *KeyStore* is one of these classes. It provides to store cryptographic keys and certificates. These stored keys and certificates are used to set identity on security configuration. The helper class is used to read and store certificates on Trust Store, Supported Cipher Suites and Identity properties.

Psk Store configuration sets pre-shared key for authentication process. It has identity and key parameters. The parameters are stored in text format. The parameters are same for both server and client during the communication of each other. Trust Store includes root certificates on X.509 certification format. X.509 is a standard for public key certificates. It contains public key and an identity [29]. Identity provides private key and the definition of used certification chain to connector. The parameter is used to convince nodes identity server to client and client to server. Therefore, the values must be same to ensure this process on both nodes. Supported Cipher Suites, as it is understood form its name, include list of supported cipher suites. To secure communication, server and client must support at least one mutual cipher suite and the parameter can not be empty in connector.

After making all these configurations on a builder object, the builder is added to *DTLSConnector*. It creates a configuration object to be used on server or client nodes. At the end, the nodes are ready for secure communication after the configuration.

3.5 Implementation

Our proposed security extension is about protecting DTLS 1.2 against DoS attacks. Therefore, we develop the solution on *DTLSConnector* class in scandium-core module. The class includes all main functions about DTLS in Californium. As a result, both server and client implementations, which are used mutually by *DTLSConnector* class, are affected from the development. We create a new *DTLSConnector* class that is named *DTLSConnectorClient*. Therefore, the client side implementation does not affect the development. The created class maintains the previous functionalities. We focus on *DTLSConnector* to improve and it is used by our server.

We examine the received packets on the server side to enhance the secure of DTLS protocol. In the original version, all received packets are forwarded to related progresses in *DTLSConnector*. If there is no procedure defined for a specific packet, the reception of the packet is only put into a log information file and the information is shown. Indeed, all packets of same type are included in all this process without exception. This process is a time-consuming one, especially for packets received before handshake. Therefore, we decide to develop a new procedure to decrease this time consumption. There is no change for successful handshake packets, but the new implementation catches multiple received packets without successful handshake from the same resource.

Aim of the enhanced security extension is providing protection for a CoAP server from DoS attacks. There is no control mechanism about IP address on DTLS version 1.2 [13]. Therefore, controlling received packet from clients eliminates the risks on security of the communication.

When we develop the security extension, we implement a main algorithm that is shown in Algorithm 1. The other three algorithms are helper methods of the main algorithm. Algorithm 1 includes the processes applied to received requests to provide security. In the algorithm, processes are shown step by step.

3.5.1 Algorithms

In Algorithm 1, the records are prepared by processing received data included in datagram packets at the beginning. The records include the client's IP address, the content of the message and some additional information. We keep track of the IP address for DoS attack control. Then, the client's port information is checked. If an attacker uses a DNS or NTP server, all messages sent by the attacker will have 53 or 123 as the source port number in the received packet. Therefore, if the client's port is one of these ports, the algorithm calls the method *insertBannedIpAddressList* for inserting the IP address into *bannedIpAddressList* table in the database (called DB in the algorithm). After that, the algorithm checks the IP address with the port in Banned IP Addresses List. If the address exists in the list, the IP address is inserted to *receivedPacketDelay* table in the database. This table includes all records of received requests and their reception time to analyze delay values of all received requests after all. If the IP address is not in the list, the record is forwarded to one of the related processes: "HANDSHAKE" massage, "APPLICATION DATA" massage, "ALERT" massage or "CHANGE CIPHER SPEC" massage. Except for "HANDSHAKE" massage process, other processes are not critical for the security extension. Therefore, we do not keep track of these types of massages.

Algorithm 2 shows the details of *processHandshakeRecord* function. The function is called if the record type equals to "HANDSHAKE" massage in Algorithm 1. The algorithm tries to examine the handshake process for the record. If any exception occurs such as non-matching keys, the count of an unsuccessful handshake is increased by one point. After the incrementation, if the count exceeds the limit on banned IP addresses, the IP address of the record is added to *BannedIpAddressList* in the database. If this happens, the server does not accept any request from this client anymore until an expiration time. The expiration time is set to a month after the insertion. In the end of the forwarded processes, if there is no error, the request is added to *receivedPacketDelay* table as it is a successful request.

Algorithm 3 is a simple but significant process. It checks the address in *bannedIpAddressList* table. If the address in the list, the algorithm returns true. Otherwise, it returns false.

Algorithm 4 checks the address of the client for its unsuccessful requests. If the requests count exceeds the limit on banned IP addresses, the IP address sending the request is inserted into *bannedIpAddressList* table in the database. Thus, the server does not accept any request from this IP address until expiration time. The limit of received unsuccessful request is a parameter. The value can be determined at the start of the server process.

Algorithm 1

```
$packet \leftarrow getNextPacketFromBuffer()$            // Read datagram packet
$IPAddress \leftarrow packet.IPAddress$         // Get IP address of the packet
$portNumber \leftarrow packet.portNumber$      // Get port number of the packet
$record \leftarrow packet$              // record keeps IPAddress, portNumber and
 bannedCount that is initially zero
// Check portNumber if the request is from DNS (53) or NTP (123)
if $portNumber == 53$ OR $portNumber == 123$ then
    $insertBannedIpAddressList(IPAddress)$              // call function
end
if isBannedIPAddress(IPAddress) then
    table $ReceivedPacketDelay \leftarrow record$         // Insert the request with
     unsuccessful handshake to ReceivedPacketDelay table in DB
else
    forall records of received packet do
        if record.recordType equal 'HANDSHAKE' then
            $processHandshakeRecord(record)$                // call function
        end
        if record.recordType equal 'APPLICATION DATA' then
            $processApplicationDataRecord(record)$          // call function
        end
        if record.recordType equal 'ALERT' then
            $processAlertRecord(record)$                    // call function
        end
        if record.recordType equal 'CHANGE CIPHER SPEC' then
            $processChangeCipherSpecRecord(record)$         // call function
        end
        table $ReceivedPacketDelay \leftarrow record$ // Insert the request with
            successful handshake to ReceivedPacketDelay table in DB
    end
end
```

Algorithm 2 processHandshakeRecord(record)

```
if any process is failed then
    record.bannedCount + + ;    // Increase count of unsuccessful handshake by one for the request IP address
end
if isExceedTheBannedLimit(record) then
    table bannedIpAddressList ← record.IPAddress ;    // Insert the address to bannedIpAddressList table in DB
end
```

Algorithm 3 isBannedIPAddress(IPAddress)

```
if IPAddress $\in$ bannedIpAddressList table in DB then
    return true
else
    return false
end
```

Algorithm 4 isExceedTheBannedLimit(record)

```
if record.bannedCount > limit for unsuccessful handshake then
    insertBannedIpAddressList(record.IPAddress)
end
```

3.6 Experimental Setup

In our research, we have implement a security extension to DTLS running on CoAP. First of all, three classes are created that are named *SecureServer*, *SecureClient* and *DosDtlsAttackerToSecureServer*. *SecureServer* plays server-role, and other classes play as a client-role in the experiment. Figure 3.1 shows that our experiment setup. Every scenarios use the same setup. We only change some specific parameters such as banned Limit and number of DoS attack threads.

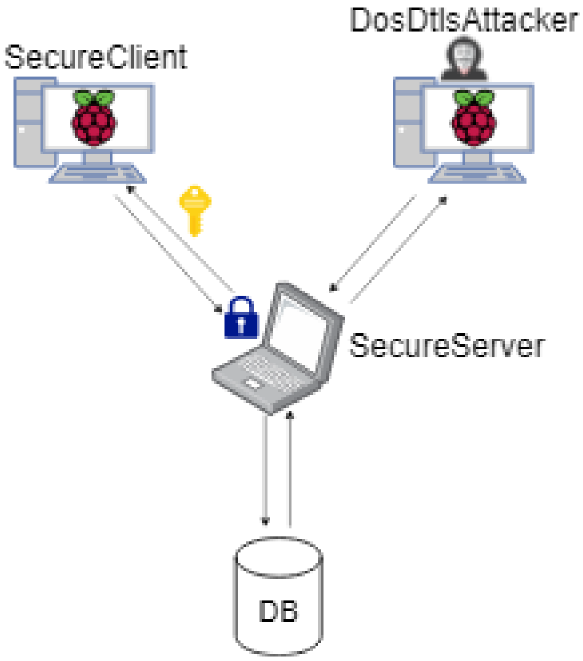

Figure 3.1: Experiment Setup.

The server is initialized by *CoapServer* class and configured by DTLSConnector-Config class. The config class includes security properties: KeyStore, TrustStore, PskStore, identity properties, and cipher suites properties. If we do not add these security properties, the server runs as Coap without DTLS.

SecureClient class is initialized by *CoapClient*, and it has the same features of *DTLSConnector* configuration with *SecureServer*. The configuration must be the same for secured communication between parties.

The client class, *DosDtlsAttackerToSecureServer*, is initialized by *CoapClient*, but its' security settings are different than others. It does not include supported cipher suites by the server. Therefore, when the client sends any type of request to the server, the server does not response to any request because the requests are not valid.

We need one of the two things to create a real DoS attack: sending a request to push capacity of a server or sending a lot of requests to keep server busy. In the experiments, we apply the second attack type. Therefore, the client performs many threads. The number of attack threads is dynamic, we change the value according to the scenario.

Chapter 4

Experimental Results and Discussion

Our experimental work is about creating network scenarios with multiple different conditions. We do not use simulation approach. Instead, we test the proposed solution with real devices in a real environmental conditions. In every scenario, there are two clients and one server, and each one is running on a separate device: *SecureServer*, *SecureClient* and *DosDtlsAttacker*. *SecureServer* runs on laptop, and each client runs on a different RaspberryPi. RaspberryPi is an experimental IoT device. It is more powerful than an average constrained device, but it is good to observe changes of performance values when the device runs our client implementation.

Californium is completely developed with Java. Therefore, we generate jar files to upload codes to RaspberryPi devices. Server codes runs on laptop, so there is no need to generate its jar file. After generated jar files for clients, we upload them to devices with a ftp server.

Order of running devices are important and fixed in every scenario. First of all, server is run before clients. It is essential to receive request from client since in opposite case, clients may send request to unavailable server address. After the server is ready to run, we can run the clients for a specific time that can differ from server time.

```
java -cp cf-secure.jar
org.eclipse.californium.MasterThesis.MasterThesisSecureClient
M 192.168.1.28 5684
```

Figure 4.1: Call SecureClient Jar File Example.

```
java -cp cf-secure.jar
org.eclipse.californium.MasterThesis.DosDtlsAttackerToSecureServer
192.168.1.28:5684 60 100
```

Figure 4.2: Call DosDtlsAttacker Jar File Example.

In the experiment, we change two parameters to call *DosDtlsAttacker* jar file. *Exceed limit of banned list* is used to detect DoS attacker and block it. It is an useful parameter to observe attacker's pressure on the network performance. When the parameter is bigger, attack detection time is increased. Other changeable value is *number of threads* of *DosDtlsAttacker*. The parameter has opposite effect than exceed limit of banned list because it decreases the attack detection time with a same exceed limit value, but it causes to a higher response delay values during an attack. Therefore, we input these parameters to *DosDtlsAttacker* when jar file is called from a terminal as shown in Figure. It is the easiest way to test with different parameters on scenarios.

In Figure 4.1 and Figure 4.2 figures show how to call client jar files. Figure 4.1 is for *SecureClient* and the other figure is for *DosDtlsAttacker*. Each jar file is needed to call with different parameters. In every jar call, mean of first four parameters are same. These jar files have same names, but its main class names are different.

In Figure 4.1, fifth parameter can be 'S' or 'M'. Meaning of 'S' is sending a single request, the other option is sending multiple requests to target address. We use 'S' parameter on development process. In our test scenarios, we always use 'M' parameter to run *SecureClient* for multiple request. Last two values are address and port of the target. The client runs for 300 seconds. The run time is not critical because we want to observe effects of *DosDtlsAttacker* on the network performance. Therefore, the time is enough to show the effects before and after the attack.

In Figure 4.2, after main class name, order of these parameters are target address with port, running time limit and number of thread. As it is seen, the client has less running time limit than *SecureClient*. Thus, we can examine the changes between before and after a DoS attack. Like the running time limit of client, number of thread is the other most effective parameter for performances of devices in the scenarios. Because of this reason, we run *DosDtlsAttacker* after *SecureServer* and *SecureClient*.

After uploading jars, we connect devices remotely with Putty from a PC, and call jar files with parameters. When *SecureServer* received first packets from the *SecureClient*, it shows log messages on the console. The logs are about handshake processes, and at end of the logs we can see ”successfully completed” message. When the log messages are seen in the console, *SecureServer* and *SecureClient* can communicate with each other securely. After communication proceeds for a specific time, we can run other client as *DosDtlsAttacker*. First running process of the attacker client takes time, because it tries to send as many message as the number of threads. The number of thread, run time are changeable parameters for *DosDtlsAttacker*. *SecureClient* has constant run configuration.

Every scenario, we get three delay graphs. First graph which is named as delay graph is about delay for successfully received requests by server. The requests are came from *SecureClient* to *SecureServer* because *DosDtlsAttacker* does not complete handshake with *SecureServer*, and its requests are not reach to deep into *SecureServer*. In the graph, we can examine effects of *DosDtlsAttacker* on the delay values between *SecureClient* and *SecureServer*. Other two graphs are about delay of handshake and forward process for each client in server. These values include both successful and unsuccessful requests.

In all delay graphs of scenarios, the X-axis is elapsed time in seconds, and the Y-axis is the related delay value in milliseconds.

4.1 Scenario 1: Effects of Low Thread Number and High Banned Limit

In this scenario, the number of thread parameter is set to 100, and the exceed limit on inserted banned IP addresses list is set to 50. Packet delay of *SecureClient* on *SecureServer* on Figure 4.3, communication process delay of *SecureClient* on Figure 4.4 and communication process delay of *DosDtlsAttacker* on Figure 4.5 are shown.

In Figure 4.3, first and maximum delay value is 8380 ms at 1.13 seconds of run time for an success handshake between *SecureClient* and *SecureServer*. There is no development for the handshake process by the security extension. Upto 90 seconds, *SecureServer* and *SecureClient* are successfully communicating with each other, and *DosDtlsAttacker* does not send any request to *SecureServer*. After that, when *SecureServer* starts to receive *DosDtlsAttacker*'s requests, delay increases obviously. The situation continues approximate upto 125 seconds. In that point, *SecureServer* inserts to the IP address in banned list in the database, and it starts to reject requests coming from the banned IP as *DosDtlsAttacker*. After the insertion, delay is decreased, but it is not same before the DoS attack. Nevertheless, there is a good improvement for delay values between *SecureServer* and *SecureClient*.

Figure 4.4 shows that received requests' delays from *SecureClient* to *SecureServer*. The delay values are calculated to each request. First and max value of the figure is about handshake process. The value is 75ms. After the process, there are small values until 90s. In there, delay values increases to 20-25ms because *DosDtlsAttacker* starts to send request to server at the same time.

Figure 4.5 shows that delays of record process to abort handshake. These process values are bigger than *SecureClient* because *SecureServer* checks these requests and try to handshake with each other. Until 110s, there are no values because the client runs with 100 threads, so begin of all threads takes time.

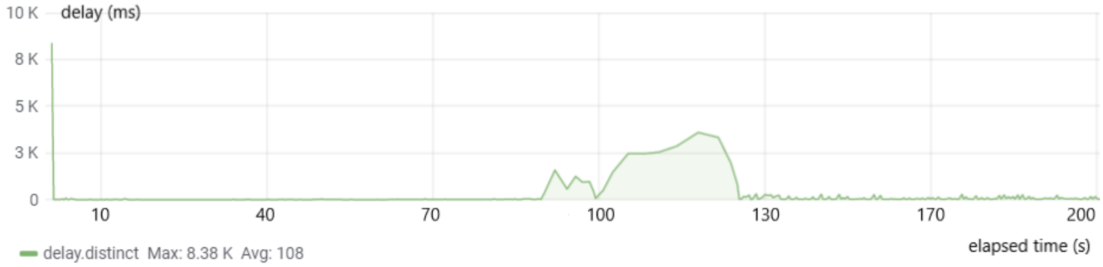

Figure 4.3: Delay on Secure Server for scenario 1.

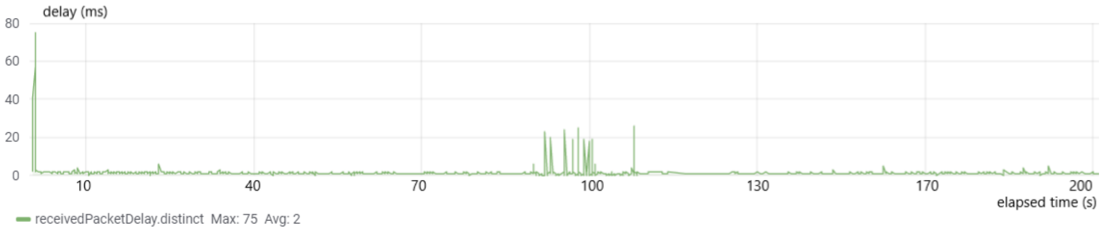

Figure 4.4: Packet delay of Secure Client for scenario 1.

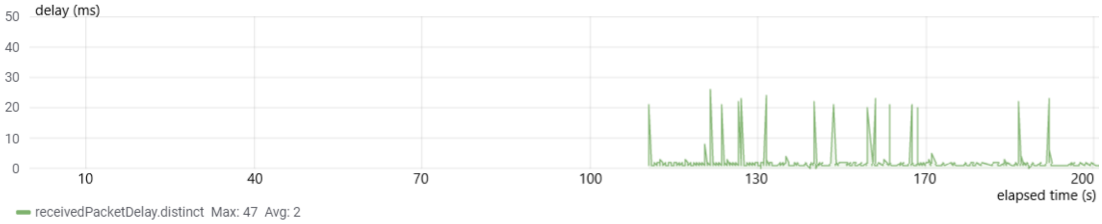

Figure 4.5: Packet delay of DoS Attacker Client for scenario 1.

When we compare values of delay processes between *SecureClient* and *DosDtlsAttacker*, we can easily observe different values. *SecureClient* has smaller values than the other because it completes handshake during the communication with server. However, *DosDtlsAttacker* has higher values than *SecureClient* because it tries to handshake with server on each request.

4.2 Scenario 2: Effects of High Thread Number and High Banned Limit

In this scenario, number of thread parameter is set to 1000, and exceed limit to insert banned IP address list to 50. Packet delay of *SecureClient* on *SecureServer* on Figure 4.6, communication process delay of *SecureClient* on Figure 4.7 and communication process delay of *DosDtlsAttacker* on Figure 4.8 are shown.

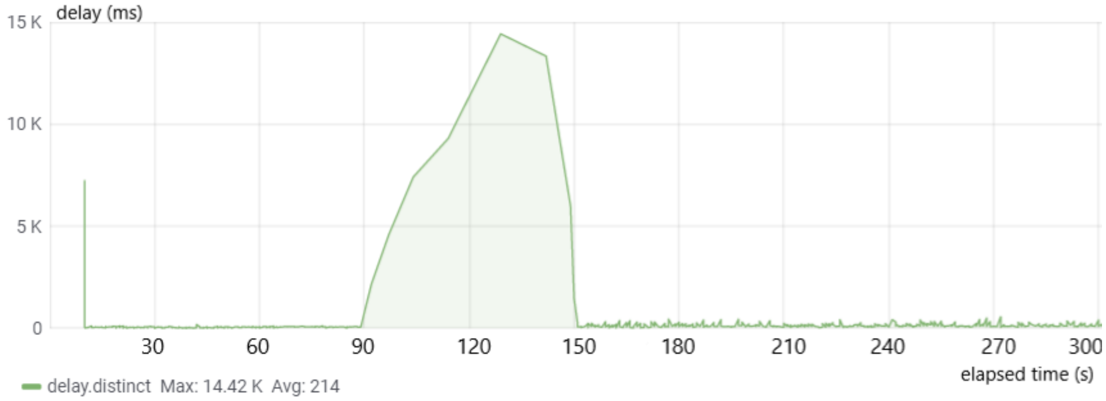

Figure 4.6: Delay on Secure Server for scenario 2.

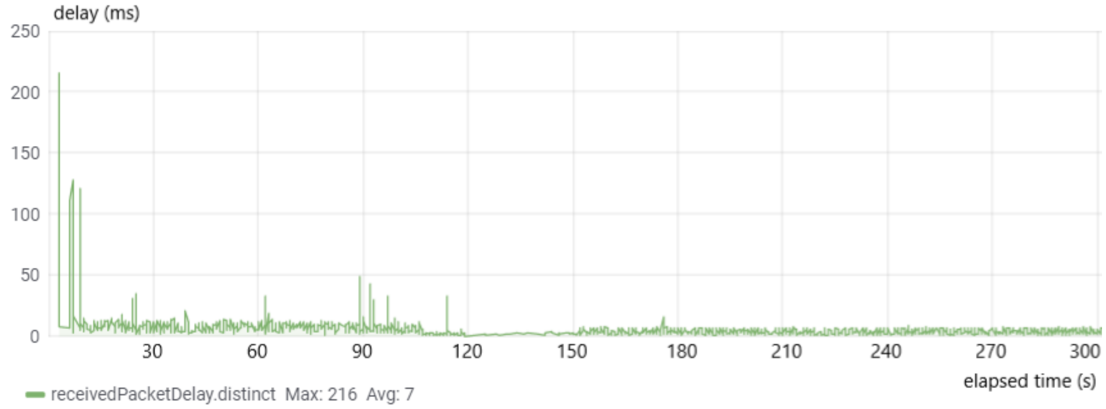

Figure 4.7: Packet delay of Secure Client for scenario 2.

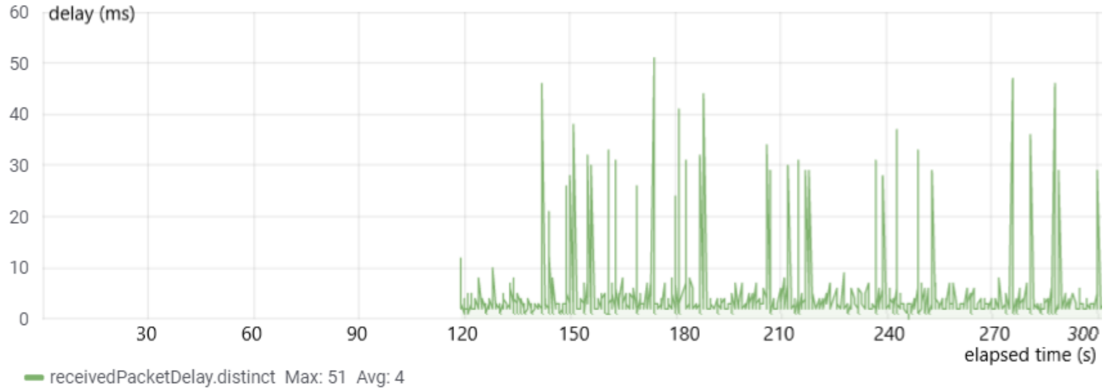

Figure 4.8: Packet delay of DoS Attacker Client for scenario 2.

Figure 4.6 shows that delay graph of the scenario. First delay is about handshake process which value is 7.6 seconds. In the scenario, number of thread parameter value increases 100 to 1000. Therefore, delay values are obviously bigger than previous scenario during under the DoS attack. Max delay value is 14.42 seconds due to the attack, and the value is approximately two times more than the scenario's delay value of handshake process. The comparison is significant to network performance because handshake process is occurred once and again the delay value is unexpected during communication. The unexpected situation increases overall communication time. In this point, our security extension provides communication quality and it decreases delay values average 137ms. Of course, the average value is more than delays of non-under attack but the value is acceptable to provide communication.

When the network is under DoS attack, *SecureClient* can send less request than normal because *DosDtlsAttacker* keeps busy the network. After the attacker is inserted to banned list, *SecureClient* continues to send request as before the attack.

4.3 Scenario 3: Effects of Low Thread Number and Medium Banned Limit

In this scenario, number of thread parameter is set to 100, and exceed limit to insert banned IP address list to 25. Packet delay of *SecureClient* on *SecureServer* on Figure 4.9, communication process delay of *SecureClient* on Figure 4.10 and communication process delay of *DosDtlsAttacker* on Figure 4.11 are shown.

When we compare results of the scenario with scenario 1, changing limit on banned IPs list value is more influence than number of thread values for elapsed time of during attack. The elapsed time of attack is decreased to 18s from 36.5s than scenario 1. The changes shows that the limit on banned IPs list parameter is directly affected to elapsed time of attack.

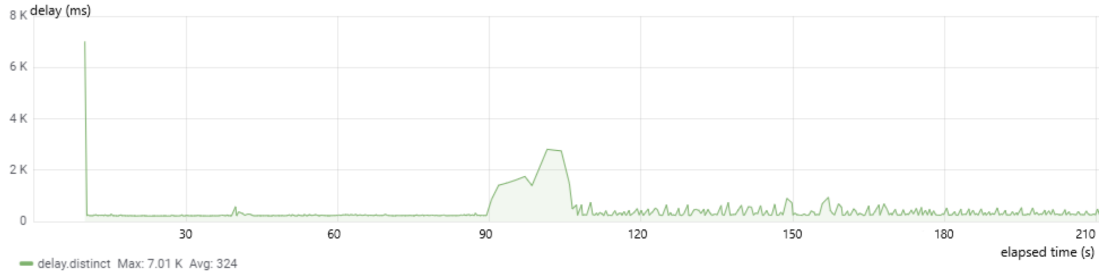

Figure 4.9: Delay on Secure Server for scenario 3.

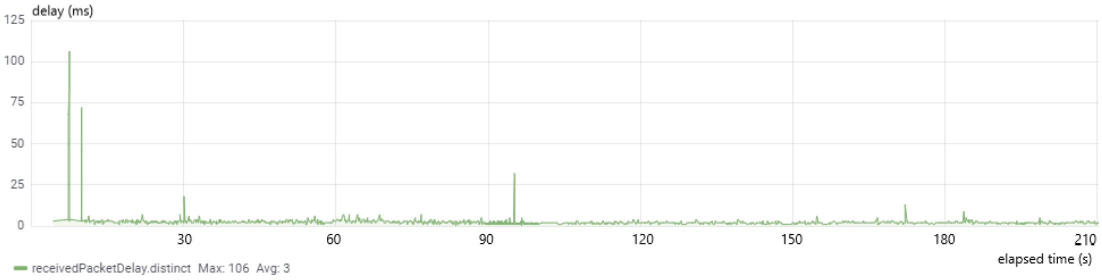

Figure 4.10: Packet delay of Secure Client for scenario 3.

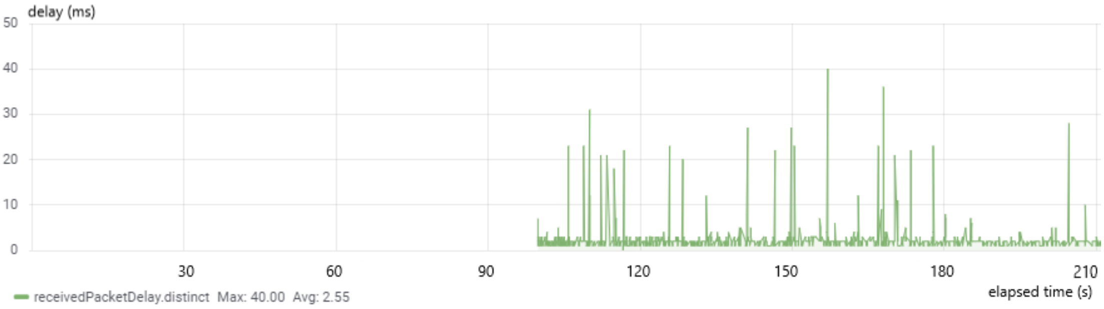

Figure 4.11: Packet delay of DoS Attacker Client for scenario 3.

4.4 Scenario 4: Effects of Medium Thread Number and Medium Banned Limit

In this scenario, number of thread parameter is set to 500, and exceed limit to insert banned IP address list to 25. Packet delay of *SecureClient* on *SecureServer* on Figure 4.12, communication process delay of *SecureClient* on Figure 4.13 and communication process delay of *DosDtlsAttacker* on Figure 4.14 are shown.

The scenario is suitable to compare effects of number of threads. In scenario 4, number of threads is increased 100 to 500 than scenario 3. When we compare these results, number of threads parameter is effective for both. Max delay during attack is increased 2.81s to 10.48s and attack elapsed time is increased 18s to 41.6s. These differences between results are significant to network performance and network security.

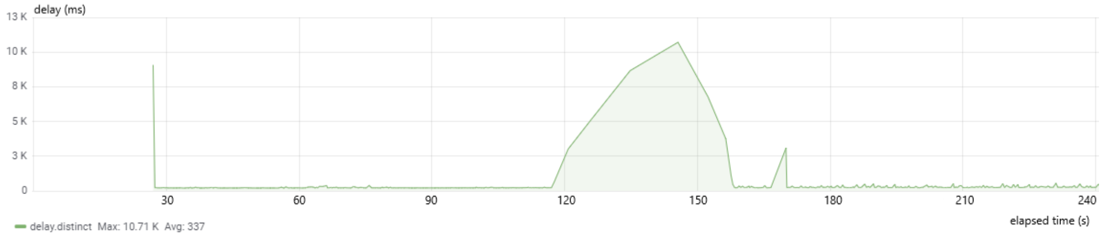

Figure 4.12: Delay on Secure Server for scenario 4.

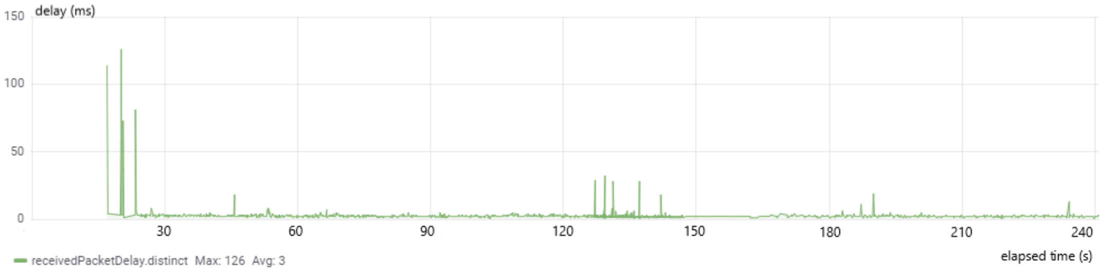

Figure 4.13: Packet delay of Secure Client for scenario 4.

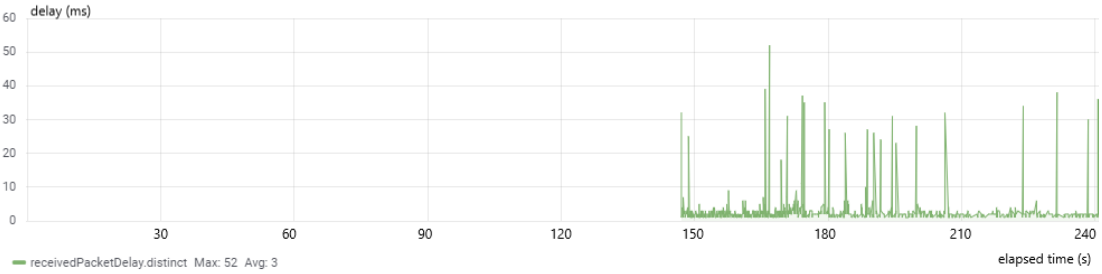

Figure 4.14: Packet delay of DoS Attacker Client for scenario 4.

4.5 Scenario 5: Effects of High Thread Number and Medium Banned Limit

In this scenario, number of thread parameter is set to 1000, and exceed limit to insert banned IP address list to 25. Packet delay of *SecureClient* on *SecureServer* on Figure 4.15, communication process delay of *SecureClient* on Figure 4.16 and communication process delay of *DosDtlsAttacker* on Figure 4.17 are shown.

In the scenario, we increase number of threads to 1000. We compare the scenario with scenario 3 and 4 because they have same limit on banned IPs list value. Thus, we can compare correctly effects of number of threads values. According to results of these scenarios, number of thread parameter is effective until 500. When the parameter is increased to 500 from 100, elapsed time of the attack is increased to 41.6s from 18s. However, if the parameter is changed 500 to 1000, the elapsed time of attack is increased to 47.4s.

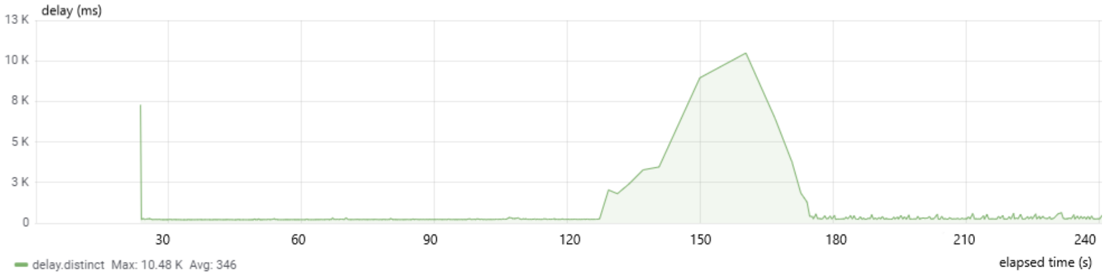

Figure 4.15: Delay on Secure Server for scenario 5.

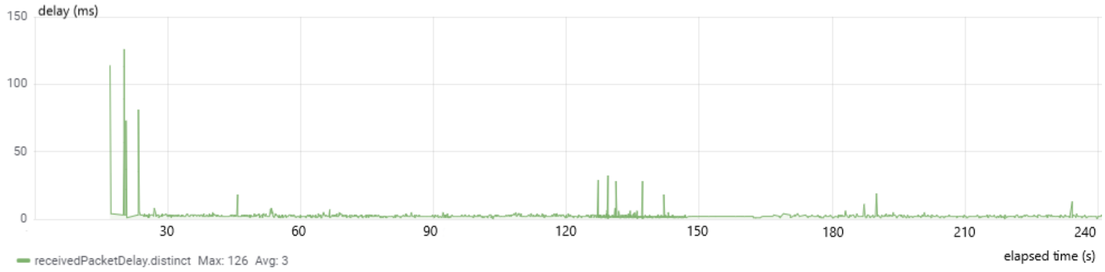

Figure 4.16: Packet delay of Secure Client for scenario 5.

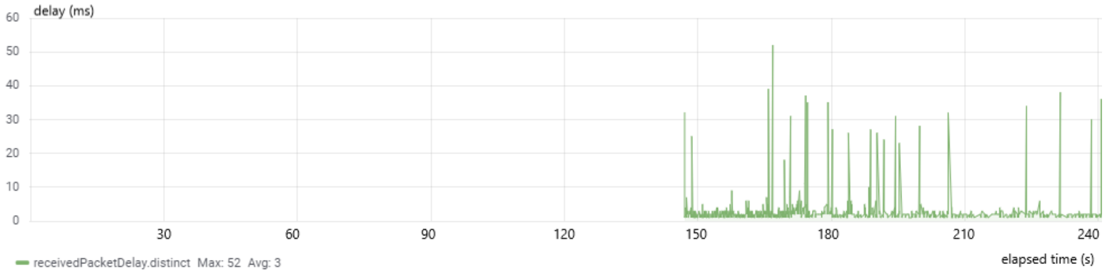

Figure 4.17: Packet delay of DoS Attacker Client for scenario 5.

4.6 Scenario 6: Effects of Low Thread Number and Low Banned Limit

In this scenario, number of thread parameter is set to 100, and exceed limit to insert banned IP address list is set to 5. Packet delay of *SecureClient* on *SecureServer* on Figure 4.18, communication process delay of *SecureClient* on Figure 4.19 and communication process delay of *DosDtlsAttacker* on Figure 4.20 are shown. Delay values are less than previous scenarios because exceed limit parameter is less ten times, and it directly affected the delay values and unprotected attack time. Attack time decreases approximately 60 seconds to 30 seconds than previous scenarios. In additionally, max delay values as 3.65s under the attack is less than first delay as handshake process.

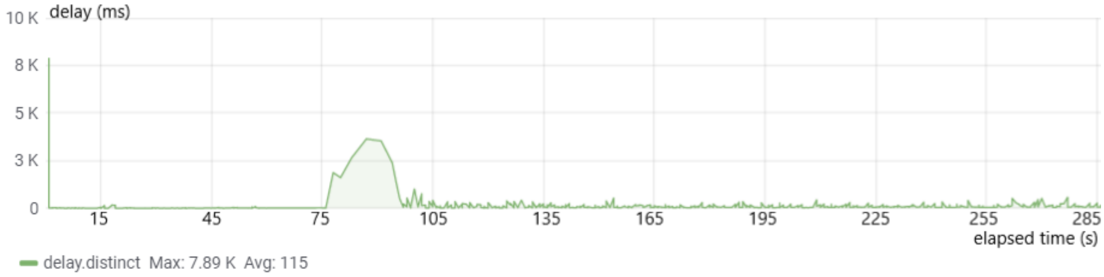

Figure 4.18: Delay on Secure Server for scenario 6.

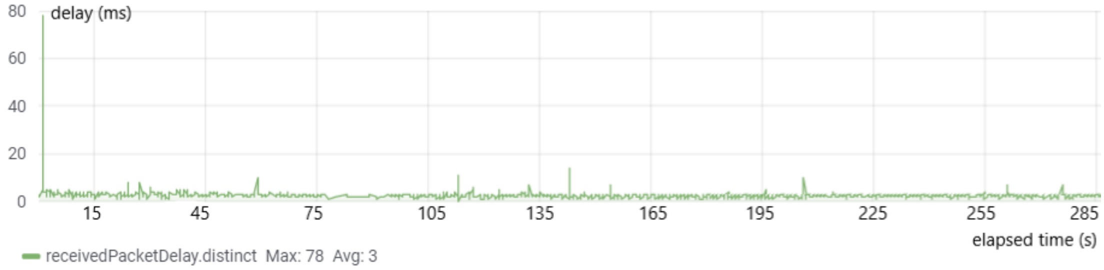

Figure 4.19: Packet delay of Secure Client for scenario 6.

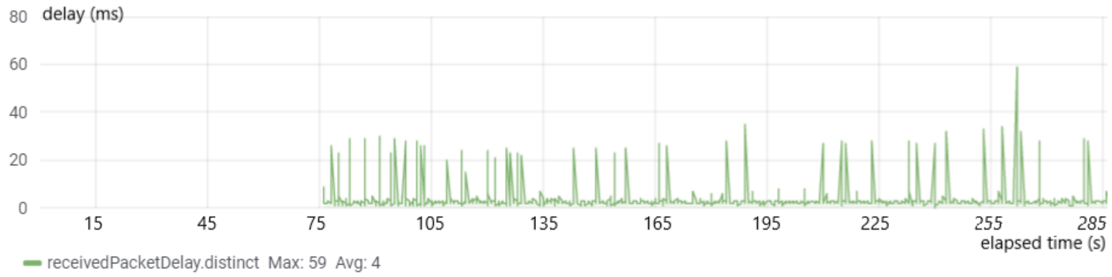

Figure 4.20: Packet delay of DoS Attacker Client for scenario 6.

4.7 Scenario 7: Effects of High Thread Number and Low Banned Limit

In this scenario, number of thread parameter is set to 1000, and exceed limit to insert banned IP address list to 5. Packet delay of *SecureClient* on *SecureServer* on Figure 4.21, communication process delay of *SecureClient* on Figure 4.22 and communication process delay of *DosDtlsAttacker* on Figure 4.23 are shown.

Number of thread parameter is ten times more than previous scenario. The difference is perceivable in the scenario's graphs. In Figure 4.21, we can analyze the DoS attacker effects directly to communication between *SecureClient* and *SecureServer*. During under attack, its maximum delay value increases to 6.08s.

Figure 4.22 and 4.23 show that exceed limit to banned list parameter absorbs differences on number of thread parameter because there is no any significant changes in the figures. Communication quality is protected successfully.

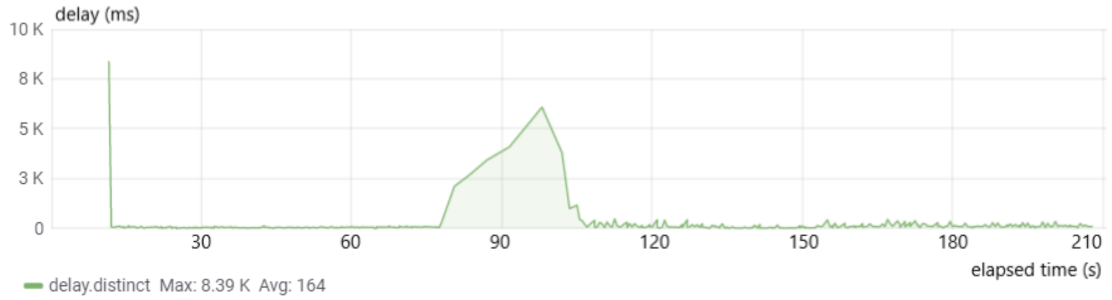

Figure 4.21: Delay on Secure Server for scenario 7.

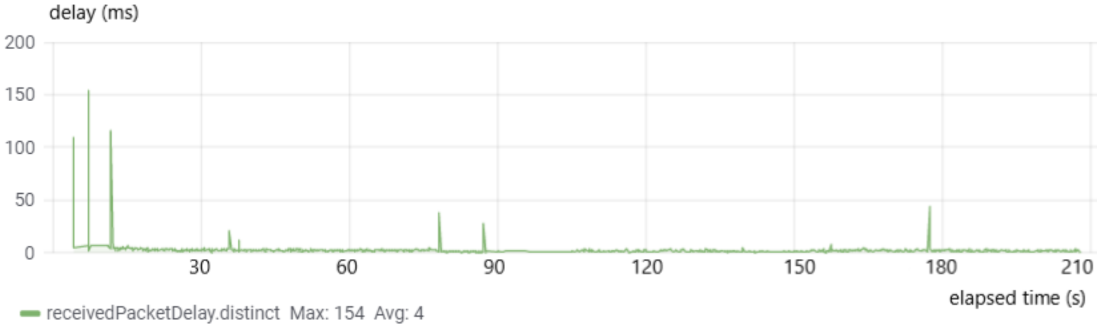

Figure 4.22: Packet delay of Secure Client for scenario 7.

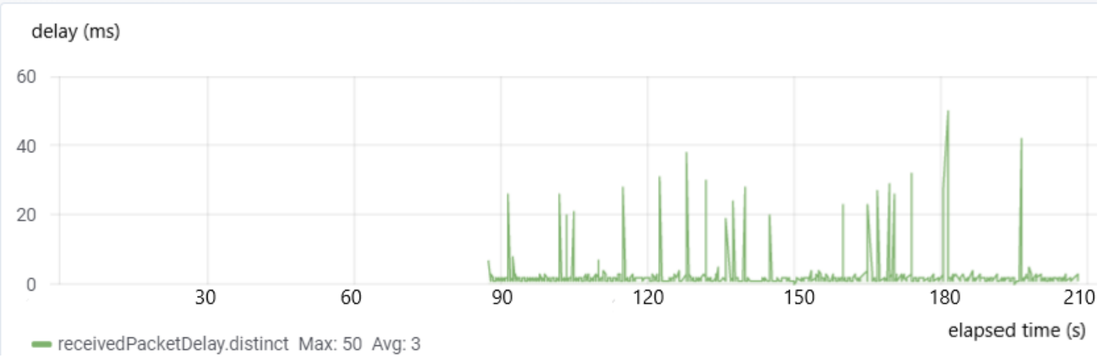

Figure 4.23: Packet delay of DoS Attacker Client for scenario 7.

4.8 Discussion

Table 4.1 shows that results of all scenarios. We test our security extension on seven scenarios. Running time of devices and order of running start are constant for all scenarios because these parameters are not affect results in no sense. However, number of threads and exceed limit of banned list is directly affect results. When we change value of these parameters, we can get different results. In first scenario, we use 100 for parameter of number of threads and 50 exceed limit of banned list. These values are enough to affect secure communication between client and server which is completed handshake process. In other scenarios, we increase or decrease these parameters. In second scenario, we increases number of thread to 1000, and the change increases time of insertion banned list to two times than previous scenario. Because the server tries handshake process with every DoS attackers, and the processes is caused to increase delay. In sixth scenario, we

Scenario Name	**Number of Threads**	**Limit on Banned IPs List**	**Max delay during attack**	**Attack Elapsed Time**
Scenario 1	100	50	3.60 sec	36.5 sec
Scenario 2	1000	50	14.42 sec	62 sec
Scenario 3	100	25	2.81 sec	18 sec
Scenario 4	500	25	10.48 sec	41.6 sec
Scenario 5	1000	25	10.71 sec	47.4 sec
Scenario 6	100	5	3.65 sec	26 sec
Scenario 7	1000	5	6.08 sec	31.2 sec

Table 4.1: Results of All Scenarios.

decreases parameter values and then both the delay values are decrease on network. In seventh scenario, we set number of thread to 1000 and exceed limit to banned list to 5. Actually, these settings are meaningful to observe achievement of our security extension on the scenario. Delay value and attack time before avoid these attackers are decreased. Thus, we get more successful result with low value of exceed limit of banned list.

In scenario 3, 4 and 5, we want to observe effects of number of threads values. In previous scenarios, we set the parameter with minimum and maximum values for the our setup. However, we set exceed limit to banned list to 25 in the 3 scenarios. The mid value of exceed limit to the list contributes variety to our study. According to results of these scenarios, values of number of threads are directly affected to performance on the network. Between 500 and 1000 values of number of threads, the network performances get very similar results each other. The reason for this is performance of Raspberry Pi devices. Our DoS attacker code is run on Raspberry Pi device and it can not start all threads in limited time. If we have 10 Raspberry Pi devices to run DoS attacker codes, we can get higher max delay values in Scenario 5.

Chapter 5

Conclusion

We propose an enhanced security extension against DoS attacks for DTLS running together with CoAP protocol to provide protection in IoT ecosystem. IoT is extremely growing area in computer science, and CoAP with DTLS is commonly run on IoT devices. Therefore, our work tries to bring a new point of view about security for web transfer protocols.

In this study, we successfully develop a security extension on DTLS to provide protection for CoAP messaging. We implement all improvements on Californium, which is Java based library including implementation of CoAP and related protocols. We compare Californium library with other CoAP libraries. It has some certain advantages. The library provides high performance, is well-implemented for CoAP and its project structure is much more well-established to be improved than other CoAP implementations.

In the experiments, we create different scenarios to test the performance of the method. In every scenarios, we test most meaningful parameter values such as number of attack threads and limit on banned IP Addresses to see the effects of new security extension. These values are differently set according to network's requirements. The parameter values of exceed limit of banned list is set to 5, 25, and 50. It is seen that the smaller values of limit achieves an earlier detection of DoS attacks. The number of threads are also changed between 100, 500, and

1000. We examine the effect of load created on the server as a result of traffic that is created by attack threads.

In future works, the extension will be added and tested with other frequently used unreliable datagram protocols such as UDP.

References

[1] M. T. Lazarescu, "Design of a wsn platform for long-term environmental monitoring for iot applications," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 3, no. 1, pp. 45–54, March 2013.

[2] S. Amendola, R. Lodato, S. Manzari, C. Occhiuzzi, and G. Marrocco, "Rfid technology for iot-based personal healthcare in smart spaces," *IEEE Internet of Things Journal*, vol. 1, no. 2, pp. 144–152, April 2014.

[3] E. Theodoridis, G. Mylonas, and I. Chatzigiannakis, "Developing an iot smart city framework," *IISA 2013*, pp. 1–6, 2013.

[4] "Web browser as universal client for iot," http://www.slideshare.net/hollobit/web-browser-as-universal-client-for-iot, 2011, accessed on 14 May, 2019.

[5] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," June 2014. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7252.txt

[6] "Mq telemetry transport," 2013. [Online]. Available: http://mqtt.org/

[7] S. Vinoski, "Advanced message queuing protocol," *IEEE Internet Computing*, no. 6, pp. 87–89, 2006.

[8] O. M. G. (OMG), "Data distribution service for real-time systems." [Online]. Available: http://www.omg.org/spec/DDSII2

[9] H. Subramoni, G. Marsh, S. Narravula, P. Lai, and D. K. Panda, "Design and evaluation of benchmarks for financial applications using advanced message queuing protocol (amqp) over infiniband," in *2008 Workshop on High Performance Computational Finance.* IEEE, 2008, pp. 1–8.

[10] K. Krinkin, A. Filatov, A. Filatov, O. Kurishev, and A. Lyanguzov, "Data distribution services performance evaluation framework," in *2018 22nd Conference of Open Innovations Association (FRUCT)*, May 2018, pp. 94–100.

[11] R. Khare, "Extending the representational state transfer (rest) architectural style for decentralized systems." UC Irvine, Information Computer Science, 2003.

[12] E. Rescorla, "The transport layer security (tls) protocol version 1.3." Internet Engineering Task Force (IETF), August 2018.

[13] E. Rescorla and N. Modadugu, "Datagram transport layer security version 1.2." Internet Engineering Task Force (IETF), January 2012.

[14] M. Kovatsch, "Californium," 2013. [Online]. Available: https://github.com/eclipse/californium

[15] I. E. Bagci, S. Raza, U. Roedig, and T. Voigt, "Data distribution services performance evaluation framework," *Security Comm. Networks*, 2015.

[16] D. Sitenkov, "Master thesis: Access control in the internet of things," 2014.

[17] S. Raza, H. Shafagh, K. Hewage, R. Hummen, and T. Voigt, "Lithe: Lightweight secure coap for the internet of things," *IEEE Sensors Journal*, vol. 13, no. 10, pp. 3711–3720, 2013.

[18] R. A. Rahman and B. Shah, "Security analysis of iot protocols: A focus in coap," in *2016 3rd MEC International Conference on Big Data and Smart City.* IEEE, 2016, pp. 1–7.

[19] T. Kothmayr, C. Schmitt, W. Hub, M. Brünig, and G. Carle, "Dtls based security and two-way authentication for the internet of things," *Ad Hoc Networks*, pp. 2710–2724, 2013.

[20] Y. Maleh, A. Ezzati, and M. Belaissaoui, "An enhanced dtls protocol for internet of things applications," in *IEEE*, 2016, pp. 168–173.

[21] H. S. T. Levä a, O. Mazhelis, "Comparing the cost-efficiency of coap and http in web of things applications," *Decision Support Systems*, pp. 23–38, July 2014.

[22] D. Thangavel, X. Ma, A. Valera, H. Tan, and C. K. Tan, "Performance evaluation of mqtt and coap via a common middleware," in *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*. IEEE, 2014, pp. 1–6.

[23] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, 2017, pp. 1–7.

[24] N. M. Khoi, S. Saguna, K. Mitra, and C. ´Åhlund, "Irehmo: An efficient iot-based remote health monitoring system for smart regions," in *2015 17th International Conference on E-health Networking, Application Services (HealthCom)*, 2015, pp. 563–568.

[25] M. Tiloca, K. Nikitin, and S. Raza, "Axiom: Dtls-based secure iot group communication." SICS Swedish ICT AB, 2017.

[26] M. Kovatsch, "Copper (cu)." [Online]. Available: https://github.com/mkovatsc/Copper

[27] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: Scalable cloud services for the internet of things with coap," in *2014 International Conference on the Internet of Things (IOT)*, 2014, pp. 1–6.

[28] M. Iglesias-Urkia, A. Orive, and A. Urbieta, “Analysis of coap implementations for industrial internet of things: A survey,” in *The 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017)*, IK4-Ikerlan Technology Research Centre, Information and Communication Technologies Area. Po J.M.Arizmendiarrieta, 2. 20500 Arrasate-Mondrag´on, Spain, 2017, pp. 188–195.

[29] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, “Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile,” May 2008. [Online]. Available: https://tools.ietf.org/rfc/rfc5280.txt