A Multi-Vendor Model for Authenticating Electric Vehicles in Smart Grid Systems using RSA and ECC

By

Ehsanul Khan

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements for the degree of
Master of Applied Science in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
University of Ottawa

Abstract

Electric Vehicles (EV) have shown promising prospects of reducing emission of carbon dioxide and air pollution. There is great demand in the market for these vehicles from the consumer end. However, in order to allow deployment of these vehicles in the streets, the charging infrastructure needs to exist and charging stations need to be set up in an efficient manner. The big challenge will be to develop a secure communication system from the Electric Vehicles (EV) to the Service Provider via the Charging Points.

Once the Electric Vehicles are plugged in to the Charging Points, they need to be authorized and authenticated by their charging service provider prior to charging. The authentication of Electric Vehicles within the smart grid system is a hot topic in today's world and is also the core focus of this thesis.

In this paper, a multi-vendor model is proposed to provide authorization and authentication service to Electric Vehicles by the Charging Service Providers through Charging Points. EVA Charge Boards specified by the ISO 15118 standard [ISO13] were used to emulate the actions of the Electric Vehicle (EV) and the Electric Vehicle Supply Equipment (EVSE). The multi-vendor model consists of two vendors, Vendor A and B, responsible for the registration, certificate issuing, certificate handling and certificate validation of the Electric Vehicles. Vendor A and B operated using RSA and ECC cryptosystems respectively to perform the cryptographic operations within the authentication processes. The cryptographic operations within both the cryptosystems used in the model were then tested in order to compare their performances between them and also with State of The Art schemes.

Acknowledgements

My sincere gratitude goes to my supervisor Professor Hussein Mouftah, who both encouraged and funded my research. I would also like to thank Binod Vaidya deeply for his invaluable advice, encouragement, and guidance throughout my research activities. This thesis could not have been successful without their help.

Furthermore, I would like to acknowledge the University of Ottawa for providing me with the opportunity to undertake my Master's degree and provide me with a Teaching Assistantship during my study.

Finally, I would like to thank my parents for financing me and encouraging me to pursue higher studies at the University of Ottawa.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Background

Cryptography is the study of techniques to facilitate secure information exchange between two persons/entities amidst presence of adversaries waiting to grab valuable confidential information from the middle. The challenge is to make it more difficult for the adversary to collect the data being exchanged between the entities in its original form. Encryption is the process of converting plaintext to ciphertext in order to hide the original information that is to be passed from one entity to another. Decryption is the reverse of encryption, where in the ciphertext is reverted back to plaintext in its original form. The default way of exchanging messages is by the encryption and decryption process, where the sender encrypts the data and the receiver decrypts the data using keys. Broadly, there are two types of algorithms for exchanging data: Symmetric key algorithms and Asymmetric key algorithms. Symmetric key algorithms use only one key to encrypt and decrypt data. In this case, the same key has to be possessed by both the entities prior to exchanging messages. Asymmetric key algorithms (also known as Public Key Cryptography) require both entities to possess keys in pairs: a private key and a public key. Thus, each entity is to have their own private and public key, where the private key always stays with the owner and is never to be shared with anyone and the public key can be shared with the world. The sender can use the receiver's public key to encrypt the message to be sent to the receiver. Upon receiving the encrypted message, the receiver can use its private key to decrypt and retrieve the original message. Examples of symmetric key algorithms include Advanced Encryption Standard (AES) algorithm, Twofish, Serpent, Blowfish etc. ELGAMAL is an example of the asymmetric key algorithm.

Digital Signature is used in the field of cryptography to verify the authenticity of digital messages. A Digital Certificate is an electronic document used to identify the ownership of a public key. These certificates typically contain information such as the issuer of the certificate, subject of the certificate, expiry date of the certificate, public key, digital signature etc. Digital Certificates are usually signed by a Certificate Authority, an entity responsible for issuing certificates. Key establishment protocols are used to provide ways for two or more entities to establish a shared common secret key based on contribution from all the entities involved. Diffie Hellman is a well-known key establishment protocol.

1.2 Motivation

Electric Vehicles are at the center of discussion among State of The Art technologies. They are among the most promising area from the automobile front that aim to reduce carbon dioxide emission and air pollution [KAL09]. With the high expectations and expected rise in the production of electric vehicles in the long run, planning of the development of electric vehicle charging infrastructures is of utmost importance. There is much research into connecting these electric vehicles to the smart grid system. The electric vehicles are expected to impact smart grid architecture in a profound way and have raised numerous concerns regarding the capacity to interact and connect with customers for vehicle charging. Mass deployment of charging station facilities are required for the near future, in order to allow the advent of electric vehicles in the market. Charging facilities and infrastructures are expected to be deployed in parking lots, garages and offices, allowing the electric vehicles to be charged anywhere at any time. The big issue at hand being also a major research area, and also the issue residing at the center of this thesis, is the challenge of providing electric vehicle authentication in the smart grid environment [BAU11, KHU10, KUN11, LAK11, NIC14, PAV13, ZHA12, VAI13]. Numerous authentication techniques and key management protocols are being proposed to develop the communication interface between the Electric vehicles (EV) and the Electric Vehicle Supply Equipments/Charging Stations (EVSE).

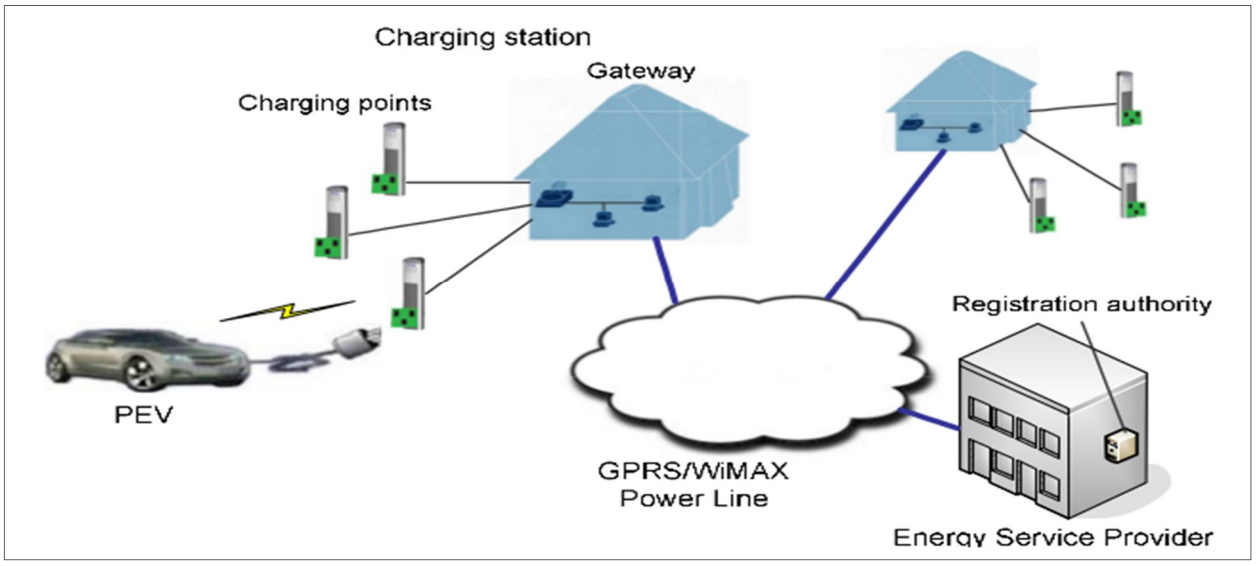


Figure 1-1: Electric Vehicle Charging in Smart Grid Environment [VAI11]

Figure 1-1 shows the broad picture of the electric vehicle charging in a smart grid environment. The electric vehicles are charged from the charging points which are connected to gateways. The charging points and gateways are able to communicate with the Energy Service Provider via GPRS/WiMax communication systems. Our focus on the thesis is narrowed towards the left end of the broad picture, involving the Electric Vehicle and the Charging Points.

The model that is going to be proposed in the thesis is not only abstract, but has also been implemented and tested. The thesis breaks down the complex model into small portions for ease of understanding for the readers, and explains the model thoroughly. Chapters 3 and 4 have gone into the depths of the model to explain the model architecture and its implementation elaborately. The model has been explained from a broad view of the overall system architecture to specific important code implementations within the system.

Since security is the prime aspect of this model, it has been made sure that the system is well-secured. Some of the important security features within the model include encryption and decryption, signing and verifying digital signatures, verifying a certificate chain, construction of Public Key Infrastructure (PKI) for efficient certificate handling, use of key exchange protocols like One Pass Diffie Hellman (OPDH) for exchanging keys, generation of random Initialization Vector (IV) for encryption-decryption purposes etc. Many of the security features implemented within the model have been based on the ISO 15118 [ISO13] standard. As per the standard, X509 digital certificates should be used. Elliptic Curve Cryptography (ECC) should be the default cryptosystem with One-Pass Diffie Hellman (OPDH) used as the key exchange protocol as per the ISO 15118 standard. The combination of ECC cryptosystem with OPDH is termed as ECDH. X509 digital certificates and the ECDH algorithm have been used in our model as will be shown later.

1.3 Objective

There are two main objectives to this thesis. The first objective is to create a robust multi-vendor model to allow efficient authentication and authorization of Electric Vehicles in Electric Vehicle Supply Equipments. The second objective is to compare the system authentication delay between RSA and ECC, compare the speed at which the cryptographic operations are performed within the Authorization

Server (Intel Architecture) and the EV board (ARM architecture), and also to compare the RSA encryption and decryption times with State of The Art schemes.

The model assumes the electric vehicles being charged in the system to belong to either of the two vendors: Vendor A or B. The vendors involved in this model are the energy service providers for the Electric Vehicles being authenticated, which are referred to as the Secondary Actors in the ISO 15118 standard [ISO13]. The vendors are responsible for registering the Electric Vehicle in their system and installing the Original Equipment Manufacturer (OEM) certificate inside the car upon the owner purchasing the vehicle. In the scenario of the electric vehicle being plugged in to the charging points for charging, the vendors are responsible for generating the vehicle's Contract Certificate prior to first time charging and/or validating the vehicle's Contract Certificate to give the go-ahead for the charging process to occur. In our case, we have used two EVACharge boards as per the ISO 15118 standard in order to emulate the Electric Vehicle and the Electric Vehicle Supply Equipment. We have used a high-configuration Intel processor to simulate the server/vendor end. The electric vehicles associated with the first vendor, Vendor A, use the widely accepted RSA algorithm for implementing cryptographic operations such as generating keys, signature signing and verification, etc. The electric vehicles associated with the second vendor, Vendor B, use the popular ECC cryptosystem combined with One-Pass Diffie Hellman as the key establishment protocol, or together known as ECDH. The cryptosystem and the key establishment protocol involved in the latter case is as per the ISO 15118 standard. However, it is not to be assumed that the ISO 15118 standard has been followed to its entirety throughout the proposed model. The motive behind creating a multi-vendor model is geared towards accommodating charging of electric vehicles belonging to different vendors, such that an electric vehicle belonging to Vendor A can authenticate itself based on the RSA cryptosystem and an electric vehicle belonging to Vendor B can authenticate itself based on ECDH with both being able to receive service from the same Electric Vehicle Supply Equipment.

The multi-vendor model is designed to protect the system from two major threats: substitution attacks and certificate fraud. Substitution attacks are carried out with a third party attacker intending to replicate or disguise itself as another entity. Certificate fraud can take place in circumstances when an entity attempts to authenticate itself using a false certificate. The model is able to prevent substitution attacks by the use of the “random challenge signing and verification” method which checks whether the owner of the valid certificate (OEM/Contract Certificate) is who it claims to be. This method checks for the authenticity of the certificate owner. Threats like certificate fraud are assessed in two steps:

verifying the trustworthiness of the certificate and checking the certificate record in the database server. The trustworthiness of the certificate is verified by checking if the certificate has been signed and issued by a common trusted Certificate Authority (CA). Further, it is ensured that the certificate being verified exists in the database system and the certificate subject (VIN or EMAID) matches with the information stored in the database. This authorizes the entity intending to be passed clear for charging.

With the model being built and tested, it was necessary to evaluate the performance of RSA and ECC cryptographic operations involved within the authentication processes. The performance metric used for comparison was the execution time of the cryptographic operations. The operations whose execution times were taken for analysis were for private and public keypair generation, encryption, decryption, signature generation and signature verification. In order to test these specified operations with both cryptosystems and by varying keysizes and plaintext sizes, independent test modules were built to derive the execution times for the operations. The encryption and decryption execution times for RSA cryptosystem were then compared with results from State of The Art schemes. The execution times were found to be smaller in all the cases as will be shown in the performance evaluation chapter of the thesis.

1.4 Thesis Contributions

The significant contributions in the thesis are highlighted by the following special features:

1) Designing the multi-vendor model: We have designed the model such that it is able to accommodate vehicles belonging to more than one vendor. In our case, the two vendors handled are Vendor A and Vendor B. This particular feature aims to prevent a scenario where an Electric Vehicle belonging to a particular vendor is not able to charge in a particular station. The model can be extended to accommodate additional vendors if needed.

2) Implementation of numerous security features from the ISO 15118 standard: Numerous security features that have been highlighted within the ISO 15118 standard have been implemented within the multi-vendor model. Some of the features include the use of One Pass Diffie Hellman as key agreement scheme, the signing and verification of random challenge, signing and verification of X509 digital certificates, the use of random Initialization Vector (IV) within AES symmetric encryption, verifying

EVSE certificates for certificate authenticity and integrity, construction of X509 digital certificate trust model, etc.

1.5 Thesis Outline

Chapter 2 of the thesis mentions about the state of the art schemes and provides the results from the papers. In Chapter 3 of our thesis, we define our System Architecture where the system components within the model are defined and their functions explained. Chapter 4 deals with the System Implementation of the proposed model. The four main use case scenarios are outlined with sequence diagrams and their steps are explained in detail. An additional fifth use case is also presented to show how the system is robust and is able to recover in case of a failure in the signature verification process. The last major chapter, Chapter 5 covers the performance evaluation section of the thesis. The first portion of this chapter presents the system timing and the overall authentication delay in our system. Our results are compared with State of The Art schemes and then plotted as column graphs or line graphs for ease of comparison and analysis in the last portion of Chapter 5.

Chapter 2 - Survey of Authentication and Key Exchange Protocols

Chapter 2 is divided into two sections. The first section discusses some recent State of the Art models that have been proposed for establishing robust key management protocols and authentication techniques for increased security in systems. The second section discusses some of the relevant and recent State of the Art schemes whose results have been provided and compared with for evaluating the performance of our cryptosystem operations used within our model.

2.1 Discussion of a few State of The Art models

2.1.1 One-Time Password Authentication in Wireless LAN (OTP)

People have become increasingly dependent on Wireless Local Area Networks (WLANs) to remain connected to the Internet for various services. This increases vulnerability to man-in-the-middle attacks and security concerns due to an open medium for wireless networks. The article in discussion [VAI06] proposes authentication based on the One-Time-Password (OTP) system [HAL96]. The objective of this protocol is to do away with the vulnerability in the authentication and security concerns within IEEE 802.11 [IEE2012] Wireless LAN's [GAS02]. Note that the One-Time-Password indicates frequent changes in password and not indicative of a single static password used all the time. A counter is kept for the duration of the password to be used in the system.

The One-Time Password Authentication protocol is intended to tackle eavesdropping on information such as login and passwords, or in other words, tackle replay attacks. It serves to prevent stealing authentication data to prevent the thief from accessing the system some other time using the stolen information. There are four main stages to the proposed protocol as shown below:

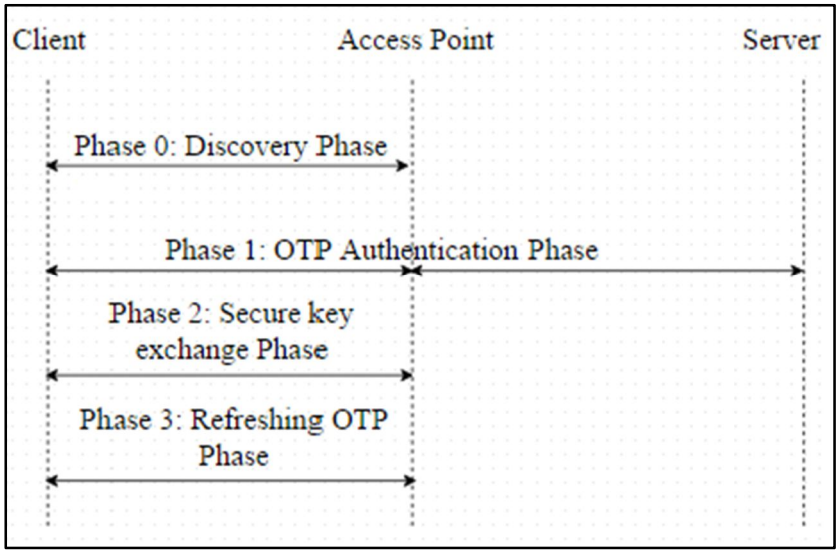


Figure 2-1: Stages in OTP Authentication protocol [VAI06]

The stages are the following:

1) Phase 0 - Discovery Phase: At this initial stage, the client and the Access Point (AP) recognize one another and determine one another's capabilities.

2) Phase 1 – OTP Authentication Phase: This stage will begin after the discovery phase. At this stage, the client authenticates and validates itself to the Access Point (AP). The authentication details are explained in [VAI06].

3) Phase 2 – Secure Key Exchange Phase: This stage begins right after phase 1. This stage involves the creation of a shared secret key between the client and Access Point (AP).

4) Phase 3 – Refreshing OTP Phases: The One-Time Password needs to change upon the counter decreasing to zero.

The One-Time Password (OTP) method can also be used with the Extensible Authentication Protocol (EAP) [CHE05], known as EAP-OTP. There are two scenarios that are considered for analysis: the proposed authentication protocol and the EAP with OTP.

The One-Time Password Authentication protocol and EAP-OTP were tested for evaluating their performance based on the number of wireless clients being authenticated. The transmission speed used

between the client and AP was 11 Mbps. The following were the performance metrics analyzed:

1) Response time: this is the time taken for traffic to propagate from one point to another and back. The response time is taken into consideration for establishing connection, at the secret key negotiation phase as well as any time taken for data exchange.

2) Authentication Delay: This is the time required to for the OTP Authentication Phase (phase 1) to be executed completely.

The first graph shows the mean response time for specified number of wireless clients and the second graph shows the authentication delay recorded in the OTP authentication phase based on the number of wireless clients. The graphs are as shown:

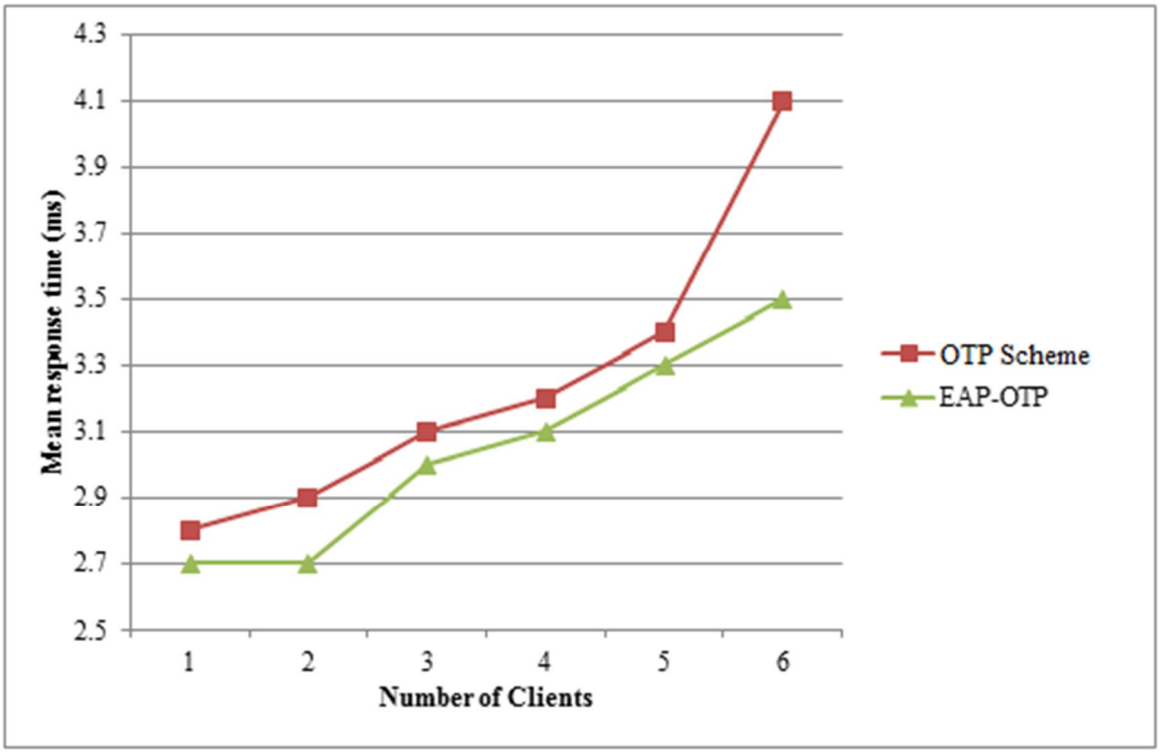


Figure 2-2: Mean response time against number of wireless clients [VAI06]

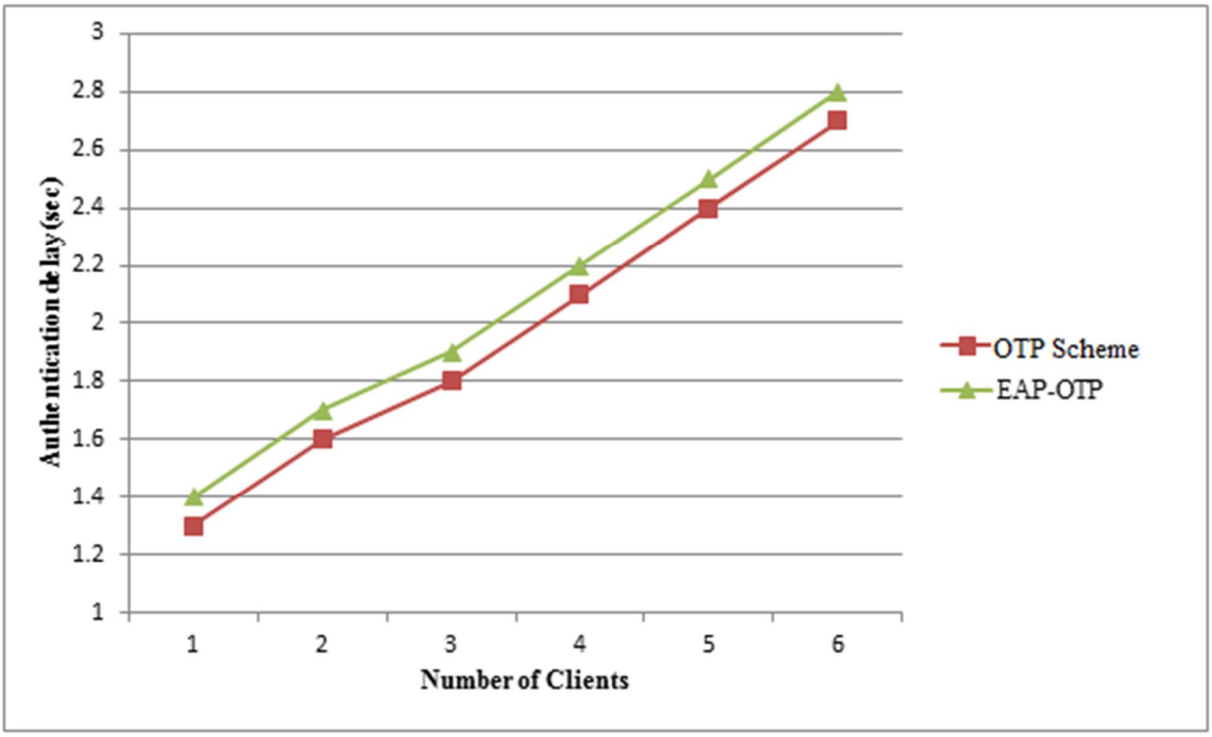


Figure 2-3: Authentication Delay against number of wireless clients [VAI06]

With respect to the first graph, the mean response time has been found to be lower for the EAP-OTP scenario while for the second graph, the authentication delay in the OTP authentication scheme has been found to be lower.

2.1.2 Key Management Protocol for Internet of Things using Implicit Certificates

A Key Management Protocol is proposed for use with mobile devices and systems connected to Internet of Things (IoT) [PAN15]. The key features of this protocol include efficient shared key negotiation, authenticating nodes, and preventing replay attacks. ECDH is used for key management. Implicit certificates, unlike explicit certificates, can be implicitly verified such that there is a chance of the private key associated with the certificate to be known by none. Implicit certificates have been used in this case, which is suitable for resource-constrained devices [RAB10] such as the IoT devices. After experimental evaluation, the proposed key management protocol with the use of implicit certificates have shown an improvement in airtime consumption with respect to the state of the art schemes.
The key management protocol can be represented in the following way:

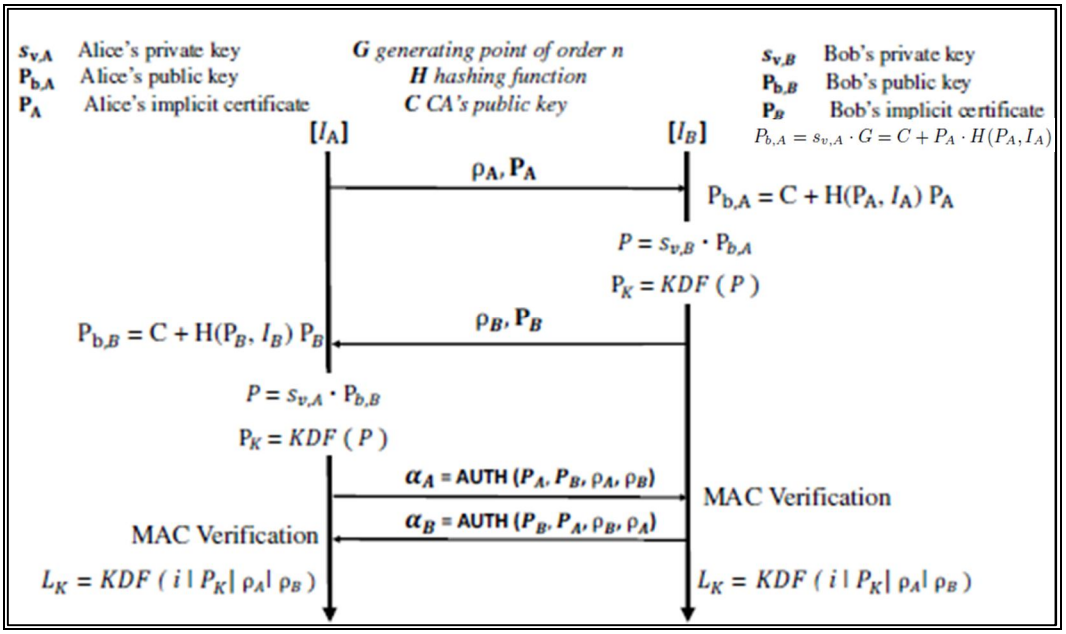


Figure 2-4: Key Management Protocol [SCI15]

The summary of the proposed key management protocol is as follows:

1) Assuming the nodes involved are Node A and Node B, Node A first shares its implicit certificate and its nonce [BAD09] with Node B.

2) Node B then extracts the public key from the certificate and uses its own private key to generate the secret key.

3) Node B sends its own implicit certificate and nonce to Node A.

4) Node A extracts the public key from Node B's implicit certificate and is able to generate the same secret key.

5) Node A and B both use a key derivation function (KDF) on the secret key to get a Pre Link Key.

6) Authentication fields α_A and α_B are calculated by each node respectively and shared with one another to ensure both have the same Pre Link Key.

7) The authentication fields obtained are then verified. The KDF is used to derive the link key for each i'th cipher block, where in the parameters for the KDF are i, Pre Link Key and the two nonces.

2.1.3 Cyber-Physical Authentication in Smart Grid

This paper proposes authenticating Electric Vehicles based on both cyber and physical domains in order to prevent substitution attacks. The idea is to ensure authentication in each domain (cyber and physical) takes place independently and then ensure that the entity being authenticated in both domains are one and the same. This is to provide increased security for systems in smart grid environments.

As seen in the protocol presented in the paper below, a TLS [ORD10] handshake is done between the EV and server over the cyber domain in order to establish a secure channel. The medium for communicating in the cyber domain is wireless, whereas the charging cable is the medium used for the physical domain. The unique technique is that the server not only undergoes a challenge and response exchange with the EV in cyber domain, but also with the EV's physical domain.

After the server sends both the challenges in encrypted form, both the domains retrieve the original challenge provided by decryption process. In the physical domain, the challenge is embedded in PWM signal [LU11] and sent to the EV via wireless channel (to the cyber domain). The EV in its cyber domain creates the response using the two challenges and sends to the server. The server verifies the response and if alright, sends an “Authentication OK” signal to the charging cable to allow for charging to start. The steps in the protocol are as shown:

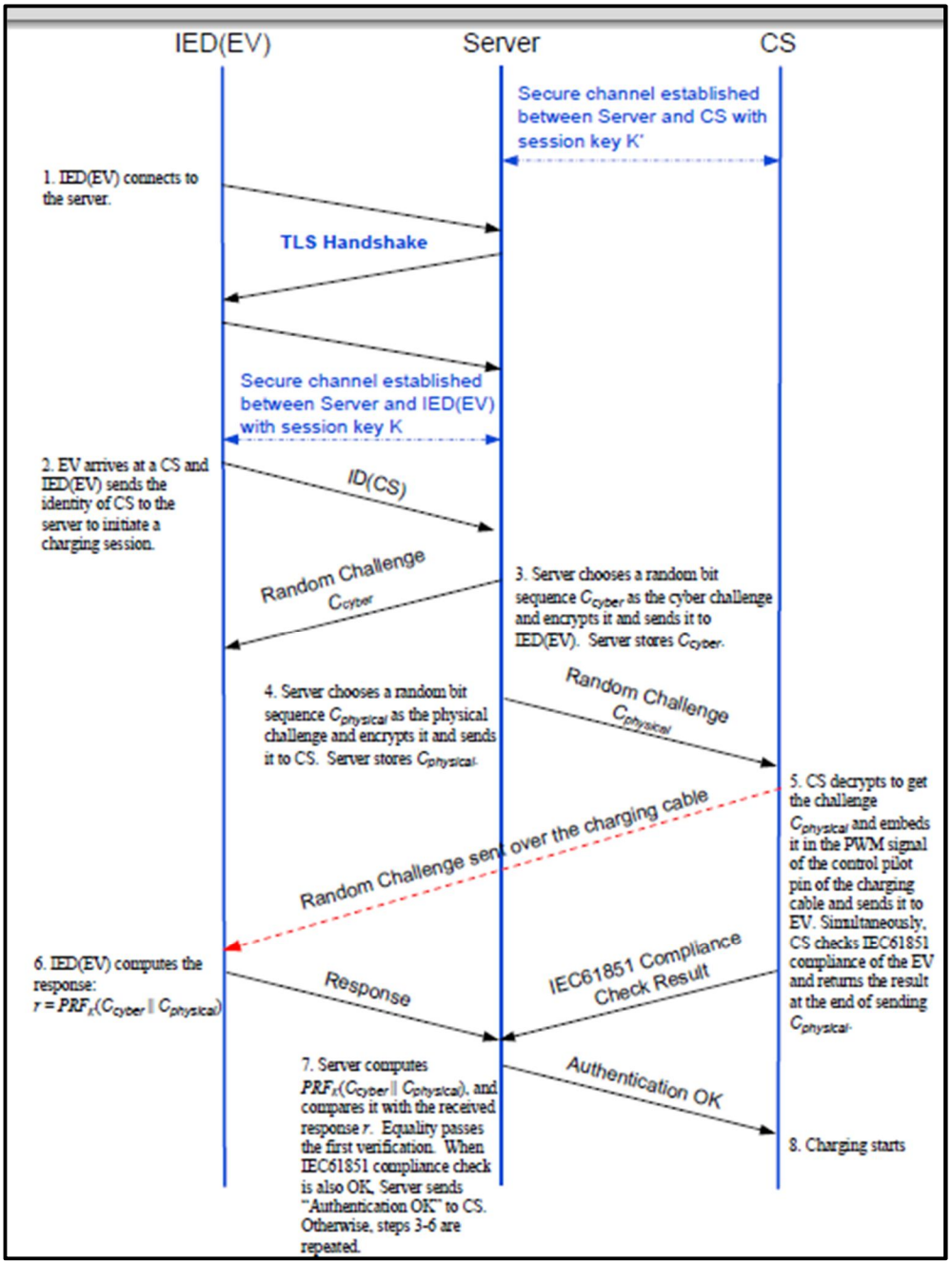


Figure 2-5: Two-domain EV authentication [CHA14]

This ends the first section of Chapter 2 where some of the relevant State of the Art technologies have been discussed. The next section discusses and provides results from relevant State of the Art schemes which have been used for comparing and evaluating the performance of our RSA and ECC cryptosystem operations. Note that we have not modified the RSA and ECC cryptosystems in any way, but rather have implemented them in our model.

2.2 State of the art schemes for performance comparison with our results

2.2.1 RSA, ECC, MQQ comparison for embedded systems

Embedded systems [SIF10] have become a daily part of our lives. Important confidential data need to be stored in such devices for our needs and as such, require good protection. Hence, these devices require robust security mechanisms. Gustavo da Silva and Edward Moreno have assessed the performance of three cryptographic algorithms [QUI13] in order to find the best one suited to operating in ARM processors [AUE95]. Due to the limitations in resources within such processors, it is important that the cryptosystems can be made to work on them with minimal resource usage. The cryptosystems that have been the subject of study are RSA [ZHO11], ECC [NIM12] and MQQ [ELH08]. The CPU processing time, memory and processing usage of each of the three cryptosystems have been analyzed and compared.

The relevance of this article with respect to my thesis are in three areas: the analysis is based on an ARM architecture, the comparison of RSA and ECC, and the analysis of CPU processing time for the cryptosystems. The boards that have been used to emulate Electric Vehicle [CHA93] and Electric Vehicle Supply Equipment [BRO13] in our analysis are also ARM architectures. It will be helpful to compare the CPU processing times of RSA[TOY10] encryption and decryption operations in our analysis compared to the one in the article being discussed. Simillar procudure has been used for finding RSA encryption and decryption times with regards to our implementation and theirs. However, the article has used ELGAMAL encryption scheme within the ECC cryptosystem whereas we used AES symmetric encryption and decryption as dictated by the ECDH algorithm. Thus, only our RSA encryption and decryption performance can be compared. SimpleScalar tool [AUS02] have been used for their simulation purposes.

ELGAMAL elliptic curve encryption and decryption has been followed in this article. The procedure for encrypting and decrypting messages between two entities A and B using ECC was as follows:

1) B calculates a private key “d” and generates a public key “Q” by “Q=d*P” where P is the generator point on the curve used.
2) To encrypt message “M” and send to B, A chooses a random positive integer “k” which ranges [1,n-1] and calculates “$C_a = k*P$”, and also calculates “$C_b = M + k*Q$” using B’s public key. A

sends C_m to B where $C_m = \{ C_a, C_b \}$.

3) To decrypt and recover the message "M", B does "$M = C_b - d*C_a$".

Based on this article, RSA based encryption takes place by deriving ciphertext "C" using "$C = M^E \bmod N$" where "M" is the message, E is the public exponent and "N" is the modulus (product of two prime numbers). Using private exponent "D", cihpertext is decrypted to recover "M" using "$M = C^D \bmod N$".

The RSA and ECC cryptosystems in [QUI13] were formulated using Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL) [MIR]. The MIRACL library is equipped with tools that provide algorithms to establish security in real world applications. The library, according to this article, provides top-notch performance in terms of processing and based on [REL], [PIG11] and [BIS04], supersedes the other known libraries out there such as Crypto++ [CRYA], LibTomCrypt [CRYB], OpenSSL [VIE02] and Lydia+GMP [GNU]. The simulator ran on StrongArm SA-110 platform [FUR96], a 32-bit RISC processor. The features of the ARM architecture simulated are as shown:

Table 2-1: Features of ARM architecture simulated [QUI13]

Fetch queue (instructions)	8
Branch prediction	Not taken
Fetch and decode width	1
Issue width	2
ITLB	32-entry, fully associative
DTLB	32-entry, fully associative
Functional units	1 int ALU/ 1 int MUL/DIV
Instruction L1 cache	16KB, 32 way
Data L1 cache	16KB, 32 way
L1 cache hit latency	1 cycle
L1 cache block size	16B
L2 cache	None
Memory latency (cycles)	64.1
Memory bus width (bytes)	4

The code, after being implemented using MIRACL, was compiled using standard GCC compiler and then cross-compiled to using ARM-LINUX-GCC compiler [SAD13] in order to make the code compatible for ARM architecture. For the purposes of this experiment, the key generation operation was not taken into consideration when calculating the processing time. Figure 2 below shows the processing time for RSA, ECC and MQQ with varying sizes for each. Keysizes of 1024 and 2048 were generated for RSA, while the sizes used for ECC were 192, 224, 258, 384 and 521.

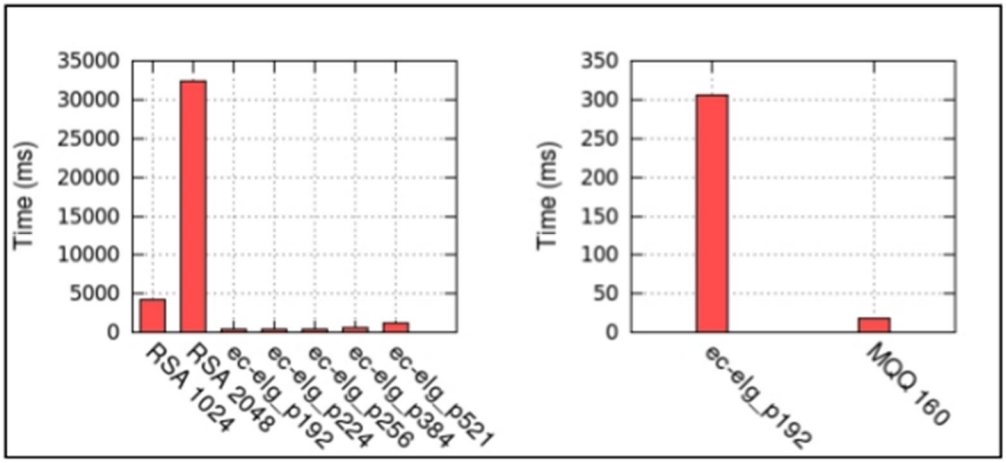


Figure 2-6: Processing Times for RSA, ECC and MQQ [QUI13]

From the figure above, it can be seen that the processing times for MQQ are relatively lower compared to ECC and significantly lower compared to RSA. It took only around 25 milliseconds to process using MQQ algorithm. RSA of keysize 1024 and ECC of keysize 192 had the lowest processing within their own comparison with ECC performing much better than RSA. The MQQ processed data 16 times faster than ec-elg_p192 (ECC of keysize 192) and 230 times faster than RSA 1024 [QUI13]. The processing time calculated is the sum of the processing time for encryption and decryption. Figure 2-7 below represents the same data as just discussed, except that now, the processing times have been broken down to show the portion used in encryption and decryption separately.

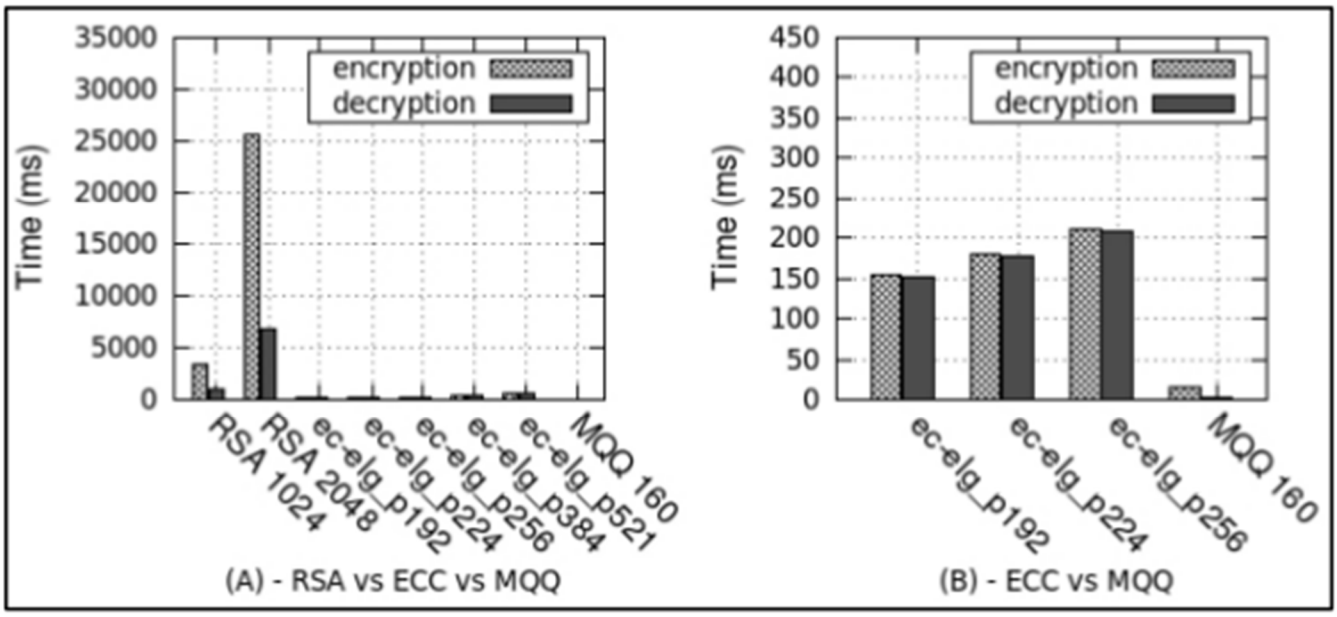


Figure 2-7: Encryption and decryption processing times (RSA, ECC, MQQ) [QUI13]

It is clear from the figure above that RSA requires the maximum processing time, within which encryption takes a significant portion of it. The decryption took only 23% of the processing time for MQQ while it took 48% for ECC 192 [QUI13].

2.2.2 Performance Analysis of Encryption in RSA, ECC and Goldwasser-Micali Cryptosystems

A recent international journal [ANJ14] has undergone a comparative study of three cryptosystems: RSA, ECC and Goldwasser-Micali. The cryptosystems were compared based on an assessment of their encryption time, decryption time, throughput and ciphertext size with varying plaintext sizes. For the sake of comparison with our results, we are concerned with their encryption and decryption time analysis with respect to varying plaintext sizes for the RSA cryptosystem.

Once again, simillar procudure has been used for finding RSA encryption and decryption times with regards to our implementation and theirs. Similar to the previous article compared, there are differences in the encryption and decryption algorithm for ECC cryptosystem. Thus, only our RSA encryption and decryption performance can be compared.

The tests were independently conducted for RSA, ECC and GM cryptosystems. The three graphs shown below presents the plaintext vs ciphertext size used, encryption and decryption time for varying plaintext size and ciphertext size. The first of the three graphs presents the respective ciphertext sizes for the plaintext sizes of 5, 10, 15, 20 and 25 KB. For the encryption time graph, the x axis represents the plaintext size. The numbers 1, 2, 3, 4 and 5 represent plaintext sizes of 5, 10, 15, 20 and 25 KB respectively. For the decryption time graph, the x axis represents the ciphertext sizes where 1, 2, 3, 4 and 5 represents ciphertext sizes of 12, 24, 24, 32 and 44 KB respectively.

1) Plaintext Size vs Ciphertext Size

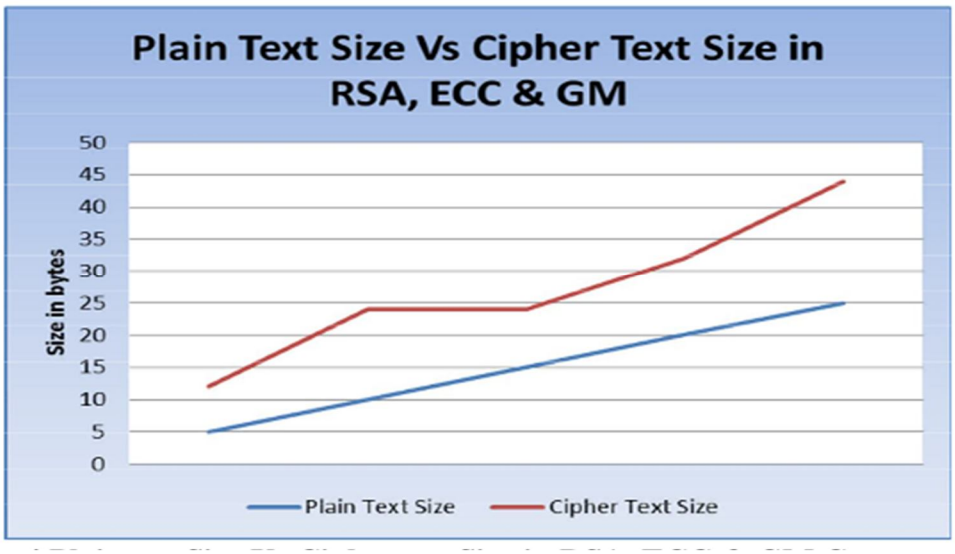


Figure 2-8: Plain Text Size vs Cipher Text Size in RSA, ECC and GM cryptosystems [ANJ14]

2) Encryption Time in RSA, ECC and GM

Verifying

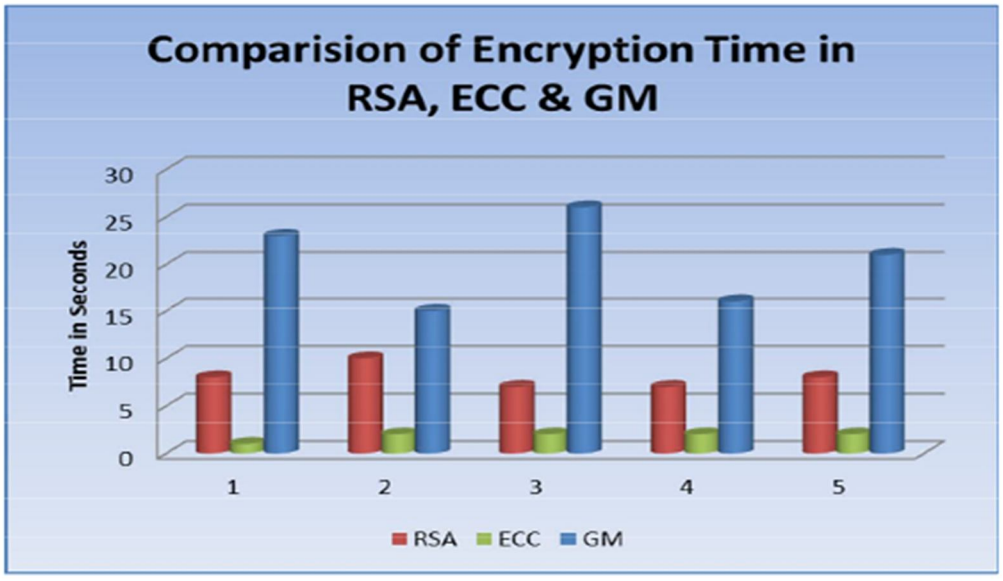


Figure 2-9: Encryption Time in RSA, ECC and GM cryptosystems [ANJ14]

3) Decryption Time in RSA, ECC and GM

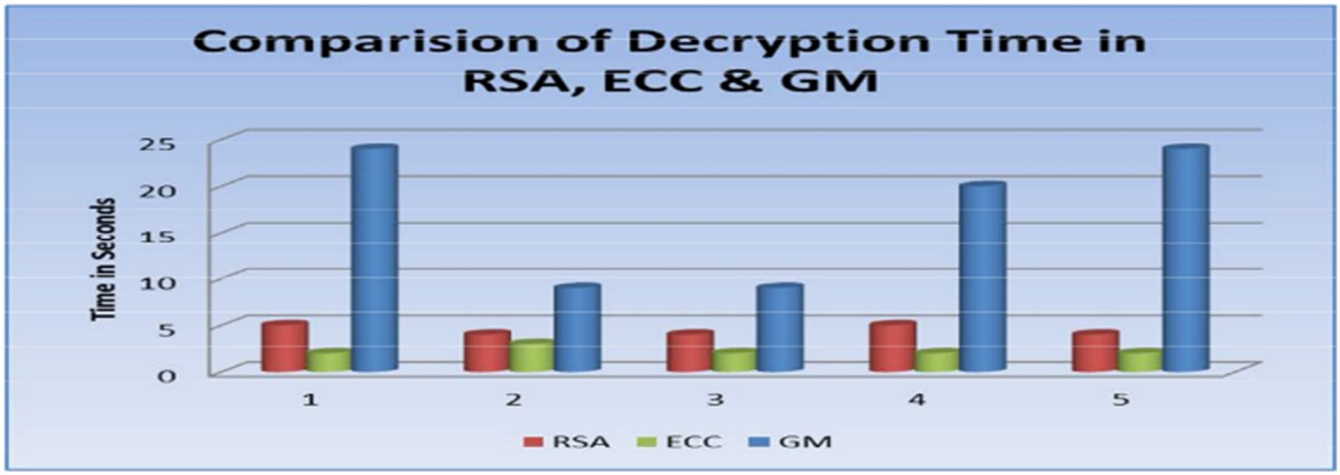


Figure 2-10: Decryption Time in RSA, ECC and GM cryptosystems [ANJ14]

Among the 3 graphs above, the graphs representing the encryption time and decryption time for the three cryptosystems are of importance to us for comparing with our encryption and decryption execution time results. Thus, as will be shown later in Chapter 5 of the thesis, a column graph was used to present our results and the results presented in paper [ANJ14], where both the encryption and decryption times were compared side by side.

ELGAMAL elliptic curve encryption and decryption is followed in this article. The procedure for encrypting and decrypting messages between two entities A and B using ECC was as follows:

4) B calculates a private key "d" and generates a public key "Q" by "Q=d*P" where P is the generator point on the curve used.
5) To encrypt message "M" and send to B, A chooses a random positive integer "k" which ranges [1,n-1] and calculates "C_a = k*P", and also calculates "C_b = M + k*Q" using B's public key. A sends C_m to B where C_m = { C_a, C_b }.
6) To decrypt and recover the message "M", B does "M = C_b – d*C_a".

Based on this article, RSA based encryption takes place by deriving ciphertext "C" using "$C = M^E \bmod N$" where "M" is the message, E is the public exponent and "N" is the modulus (product of two prime numbers). Using private exponent "D", cihpertext is decrypted to recover "M" using "$M = C^D \bmod N$".

Chapter 3 - The Multi-Vendor Model Architecture

3.1 System components, Software and Configurations

3.1.1 EVA Charge Boards (EV, EVSE)

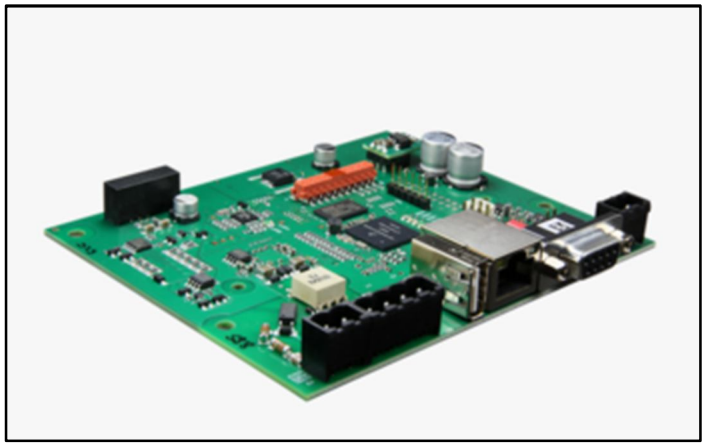


Figure 3-1: EVACharge SE Board [ELE]

The EVACharge SE boards are specified by the ISO 15118 [ISO13] standard as the boards to be employed inside Electric Vehicles and Electric Vehicle Supply Equipments (Charging Stations). These are the same boards that we have used for emulating the performance of Electric Vehicle (EV) and Electric Vehicle Supply Equipment (EVSE). The boards run on a Linux operating system although they can be accessed through computers using both Linux and Windows operating systems. The following are the board configurations:

1) Microcontroller: FreeScale i.MX287
2) Power Supply: 12V DC
3) Storage Flash: 2 Gbyte cMMC or micro SD
4) Storage RAM: 128/256 Mbyte DDR2
5) Operating System: Linux kernel 3.10
6) Temperature Range: -40C to +85C
7) Size: 120mm x 100mm x 30mm

The computer that has been used to access both the EV and EVSE boards run on Ubuntu 14.1. The configuration is as follows:

1) Memory: 8 GB

2) Processor: Intel Core I-7 CPU @ 3.4GHz x 8

3) OS Type: 64 bit

4) Disk: 365 GB

OpenSSH server [SSH] needs to be installed on the computer to be able to establish SSH connections with the board over port 22. Upon the OpenSSH server being installed, RSA, DSA and ECDSA keys were set up to ensure secure communication. OpenSSH server was already installed in the board upon purchase. *Minicom* [MIN] had to be installed in order to allow serial communication between the computer and the boards. Figure 3-2 below shows the EV and EVSE boards connected to one another along with other cables connected:

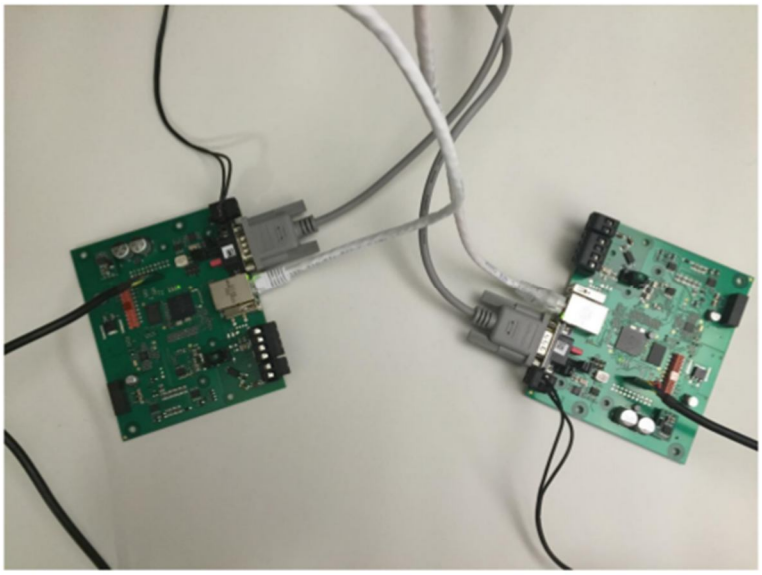


Figure 3-2: EV and EVSE board setup [SHA15]

The following are the extensions/cables that need to be connected for the boards to run:

1) Ethernet cables: Establishes connection between board and computer

2) “TTL232R-3V3” cable: Connected between board and computer for debugging purposes

3) 5V power adapter: Supplies power to boards

4) RS232 serial cable: Connected between the boards to allow serial communication between them

The EVSE board also has the Apache/1.3.4 (UNIX) server installed which was needed for communication and data transfer between the EVSE and Secondary Actor over HTTP. The server has not been utilized from the EV end since the only communication that EV has is with EVSE through serial communication via the RS232 cable. C programming language has been used in order to program

inside the boards. However, the code first had to be written in the computer before being cross-compiled to make it executable on the boards. Eclipse IDE for C/++ Developers [MEN08] has been installed in the computer in order to provide the platform for C coding. The Eclipse version used was Luna Service Release 2 (4.4.2).

There are numerous editors available for use. However, Eclipse is open-source software; it is popular and is widely used by developers. The decision to use eclipse was also motivated by the ease at which errors are detected and traced back once built. The OpenSSL 1.0.2 library [SSL] also had to be installed and integrated with the C program in order to allow use of cryptographic functions within it. C and OpenSSL were also used at the authorization server end to deal with the cryptographic mechanisms. The program had to be compiled using the standard gcc compiler in order to make it runnable on the authorization server (discussed later). However, arm-Linux-gnueabi-gcc [ARM] had to be installed in order to cross compile C code to make it runnable on the ARM architecture inside the boards.

3.1.2 Secondary Actor (Vendor A/B)

Our system has two ends: the EV-EVSE end which deals with the Car-Charging Station functions, and the other end where the main Server is deployed to provide services to the Electric Vehicles. The entity at this end also provides services such as registering the EV, providing the contract certificate for EVSE and EV, authorizing EV, allowing EV to charge, etc. This entity is being referred to here as the Secondary Actor or Vendor. Our system allows authentication and charging of EV's from two independent vendors, Vendor A and Vendor B. Vendor A and Vendor B closely follow the communication interface and standard as specified by ISO 15118 [ISO13], except that Vendor A deviates somewhat from the standard by replacing ECC and Diffie-Hellman (DH) operations with RSA. In other words, the primary difference between the two vendors is that Vendor A uses the RSA cryptographic algorithm whereas Vendor B uses ECC combined with DH (ECDH).

The laptop has the following configurations and operating system:

1) Operating system: Ubuntu 15.04
2) Memory: 3.7 GiB
3) Processor: Intel Core i3 CPU @ 2.4GHz x 4
4) OS type: 64-bit
5) Disk: 9.4 GB

The EVSE controller uses HTTP protocol to transfer data to the Secondary Actor end by communicating with the Application Server. The Application Server runs on Java, hence Eclipse Java EE IDE was used as the platform to code. The version used was Mars.1 Release (4.5.1).

Apache Tomcat Server v7.0 was installed and deployed at the Application Server. The server was integrated with Eclipse Java EE IDE. Servlets were created using Java in Eclipse Java EE IDE in order to handle HTTP post methods to receive HTTP posts from EVSE server or to hit the server at the EVSE side with post method. Four servlets were created to handle four scenarios: two for the Certificate Installation process for Vendor A and Vendor B, and two for the Certificate Validation process for Vendor A and Vendor B. MySQL database has been installed on the system, which has been used by the Authorization Server in order to either confirm Electric Vehicle registration for new contract certificate creation or confirm authenticity and validity of contract certificates to allow vehicles for charging. The version of MySQL database used is 5.6.27-0ubuntu0.15.04.1. This version, as is self-implied from the information, is compatible with the Ubuntu 15.04 operating system which we are using.

The database structure has been simplified for our use. A database is created, inside which a single table is created with the name “Cars” in order to store information about the Electric Vehicles registered in the system. Independent database systems are to be maintained for Vendors A and B, with registration information available in the database system for each vehicle that is tied to their specific vendors. That is to say, if an EV is tied to Vendor A, it means its registration information will be only available in the database system maintained under Vendor A and the EV is to be authorized by the protocol regulated by Vendor A.

Figure 3-3 below shows the types of data the table “Cars” holds for each system.

```
Database changed
mysql> DESCRIBE Cars;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| Id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| VIN   | varchar(30) | YES  |     | NULL    |                |
| EMAID | varchar(30) | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)

mysql>
```


Figure 3-3: Table Properties for “Cars”

As shown in the above figure, the first field used is the EV Id which is also the primary key. The next field is the Vehicle Identification Number (VIN) [MUL12] of the EV which is also the Subject field in the OEM Provisioning Certificate (to be discussed later) belonging to each EV. It is assumed that this VIN number is stored in the system as part of registration procedure for the EV upon purchase by the car owner. The third and last field is the EMAID or the contract Id. The contract Id is the subject field of the Contract Certificate belonging to each EV. The contract Id is checked by the system and matched with the subject field present in each contract certificate to check if the EV has a valid contract certificate or not.

3.2 System Internal Structures and Architectures

3.2.1 MySQL Database Structure

The following Figure 3-4 shows a sample of the database for one of the vendors. From this figure, it can be understood that there are three vehicles registered in the system at the time of the screen shot taken.

```
mysql> SELECT * FROM Cars;
+----+-------------------+----------------------+
| Id | VIN               | EMAID                |
+----+-------------------+----------------------+
|  1 | G43L83PQRNXIZQCKO | 3165f621a38e7e3856ed |
|  2 | R56K98SERDQGTNTYO | 4193a365ae93b4e5cf61 |
|  3 | S56K98SERDQGTNTYO | 9c9fffc8d2f2e734b964 |
+----+-------------------+----------------------+
3 rows in set (0.00 sec)
```

Figure 3-4: Database Sample

From the table above, three EV's are registered in the system. For example: EV with Id 2 has VIN “R56K98SERDQGTNTYO” which should be present in the subject field of the OEM Certificate belonging to that particular EV. The EMAID information present for that EV means that a Contract Certificate has been created by the system for that EV with contract Id “4193a365ae93b4e5cf61” which should also be present in the subject field of the Contract Certificate belonging to that EV.

3.2.2 PKI and Two-Tier Trust Model Hierarchy

Trusting entities within the system is of prime importance during the authentication stage prior to Electric Vehicle charging. A mechanism must exist in order for entities to know that the other entities with whom confidential information is to be shared can be trusted. The Public Key Infrastructure [FON11] solves that problem with policies to ensure the presence of "guardian entities" that can be trusted. The guardians here are called Certificate Authorities (CA) that are entrusted with the duty of issuing Certificates to entities involved in the system.

For our purposes, we have established a two-tier trust model hierarchy with the Root Certificate Authority (Root CA) at the top of the trust chain and its Subordinate Certificate Authority (Sub CA) right below it. Figure 3-5 shows the general structure of the two-tier hierarchy used for our systems (Vendor A and B):

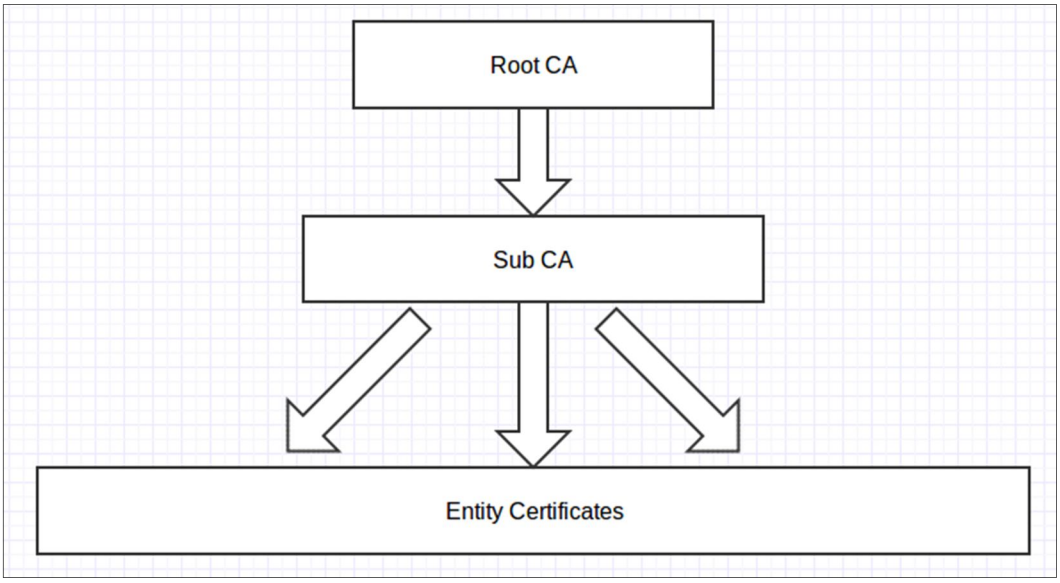


Figure 3-5: General Structure for the Two-Tier Trust Model

As can be seen from the figure above, the Root CA lies at the top of the chain and is trusted fully by all entities within the system. The Root CA self-signs itself, meaning that it generates and signs its own certificate using its own private key. The private and public keypair is generated for the Subordinate Certificate Authority (Sub CA). A Certificate Signing Request (CSR) is used to create the certificate for Sub CA. The Root CA uses its private key to sign the Sub CA certificate. After doing so, the Root CA hands off the responsibility of issuing and managing entity certificates to the Sub CA, which lies in the second tier of the trust model. The Sub CA then issues the entity certificates that include OEM

Provisioning Certificate as well as Contract Certificate for both the EV and the EVSE. These certificates are signed using the Sub CA's private key.

This trust model plays an important role in verifying the certificates belonging to entities. For instance, the EV checks the authenticity of the EVSE Certificate by verifying the EVSE certificate chain. The certificate chain in this case goes back to the Sub CA who is the common trusted anchor for both EV and EVSE. Figure 3-6 shows the specific Two-Tier Trust Model used by Vendor A. The Two-Tier Trust Model used by Vendor B is similar to Vendor A.

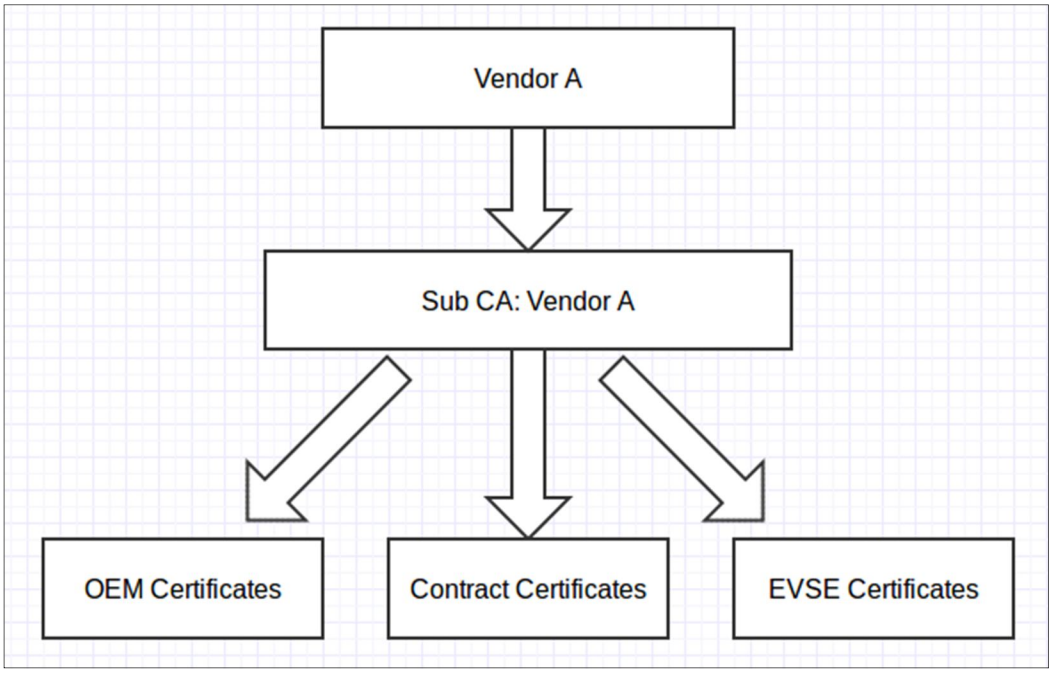


Figure 3-6: Two-Tier Trust Model for Vendor A

As can be understood from the two figures above, the Root CA for both the systems are Vendor A and Vendor B (stated as ZeeCharge on certificate). The Sub CA's do the same work as has been mentioned before, which involves issuing the OEM Provisioning Certificate, the Contract Certificate and the EVSE Certificate.

3.2.3 X509 Certificate Structures

X509 digital certificates, as defined in the ISO 15118 standard, have been used as part of our system for identifying the owner of the certificate, for certificate installation and validation, for signing and verification purposes etc. These certificates serve numerous purposes and are vital in our system. This

section looks at the X509 certificate structure, the fields involved and their appearances.

All the certificates issued using our system have the same fields. There are numerous fields involved and only a few important ones will be mentioned. The subject and the issuer fields contain the unique identity of the certificate holder/subject and the name of the entity responsible for issuing the certificate respectively. Then the version and the serial number of the certificates are specified. The expiry date states the time up to which the certificates are deemed valid. Next, the type of cryptographic algorithm associated with the public key in the certificate is stated. This is one of the fields that is distinct for Vendor A and Vendor B since for Vendor A, RSA is used and for Vendor B, ECC is used. Then the public key is embedded in the certificate.

The last two important fields are the signature algorithm and the signature itself included in the certificate. The signature algorithm is another field where in the information is distinct for both the vendors. For Vendor A, RSA with sha256 is used for signing, which in the certificate is stated as “1.2.840.113549.1.1.11”. For Vendor B, ECDSA with sha256 algorithm is used for signing, which in the certificate is stated as “SHA256 with ECDSA”.

The information within some fields will vary depending on whether it is the Root Certificate, Sub CA Certificate or an Entity Certificate. The information is distinguishable based on comparison between the three types. However, when compared with the Vendor A and Vendor B equivalent certificate for the same type, the differences are not that apparent except for the public key and the signature section. Below are images of each of the X509 certificates provided that are used in our system. The images for the certificates for both vendors of the same type are placed side by side for ease of comparison.

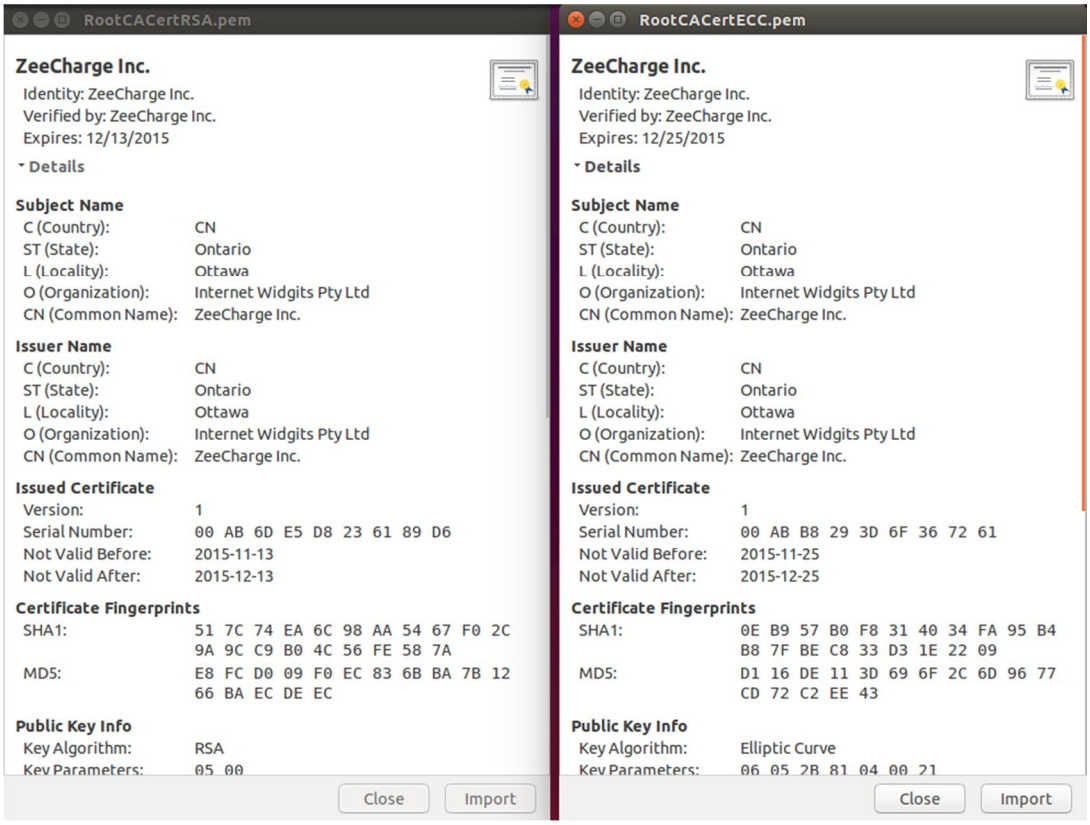


Figure 3-7: Root Certificates for Vendor A and Vendor B

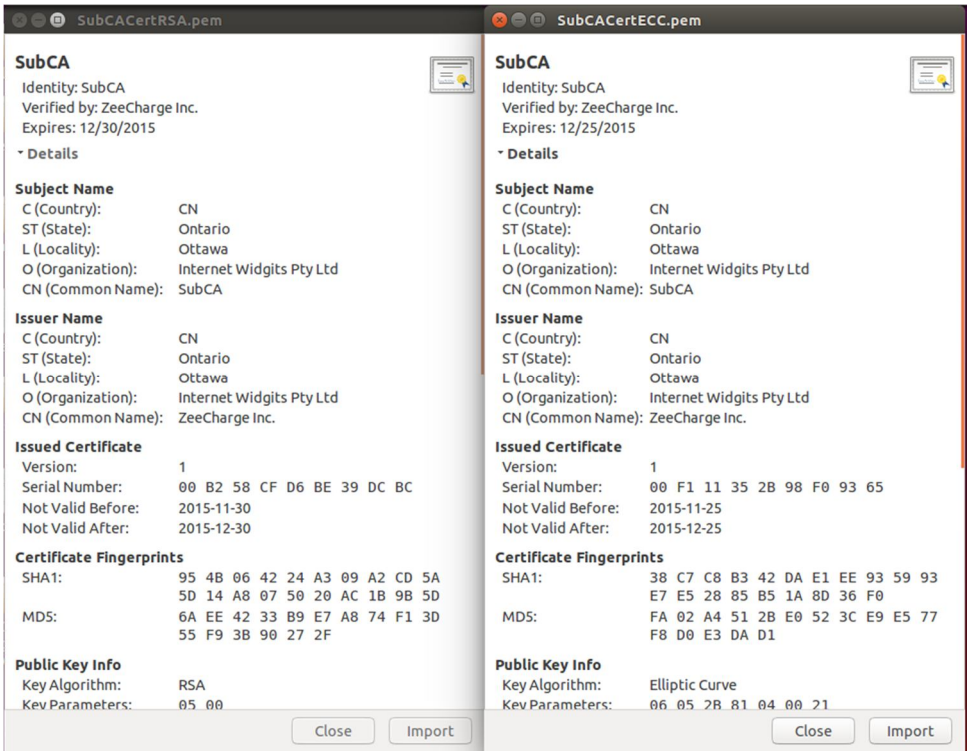


Figure 3-8: Sub CA Certificates for Vendor A and Vendor B

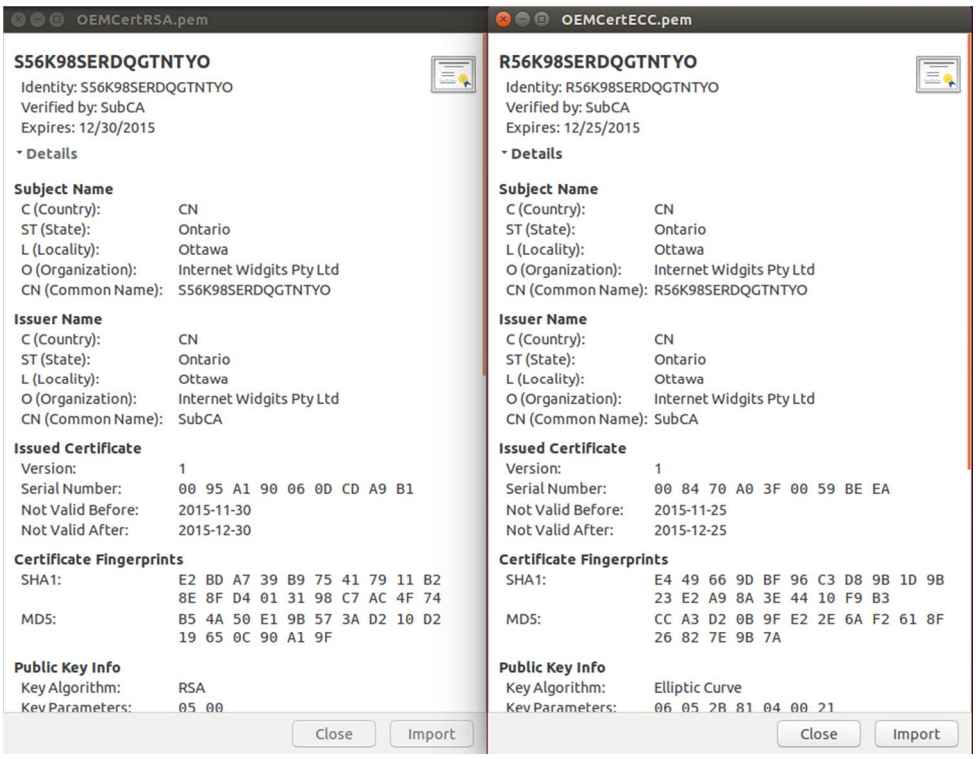


Figure 3-9: OEM Provisioning Certificates for Vendor A and Vendor B

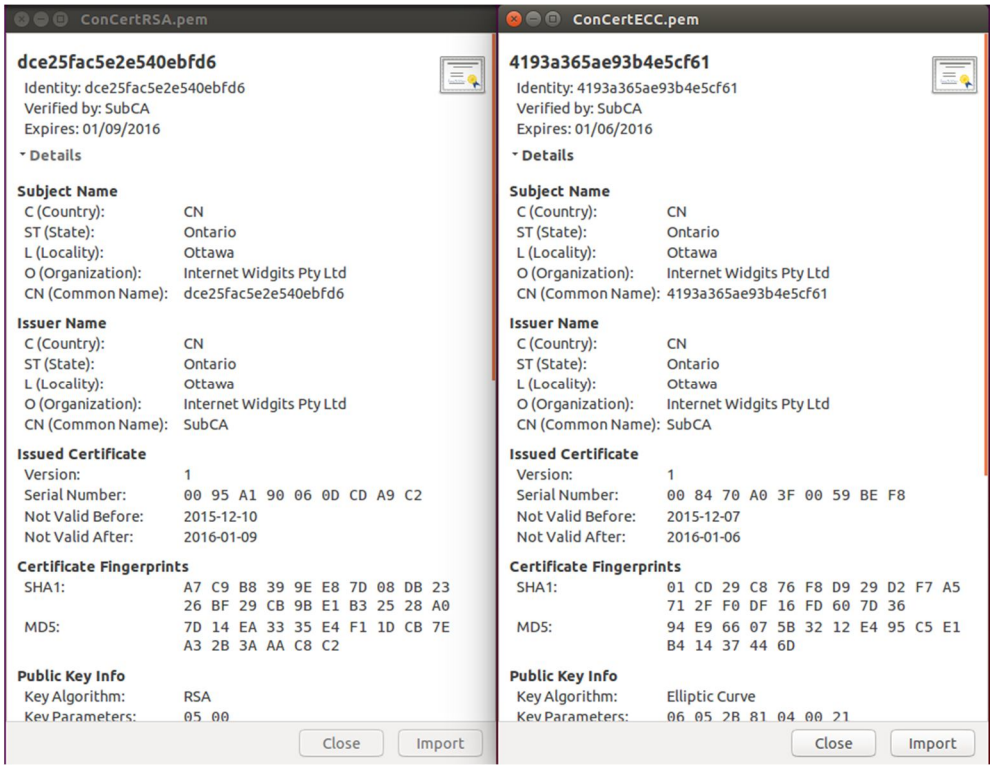


Figure 3-10: EV Contract Certificates for Vendor A and Vendor B

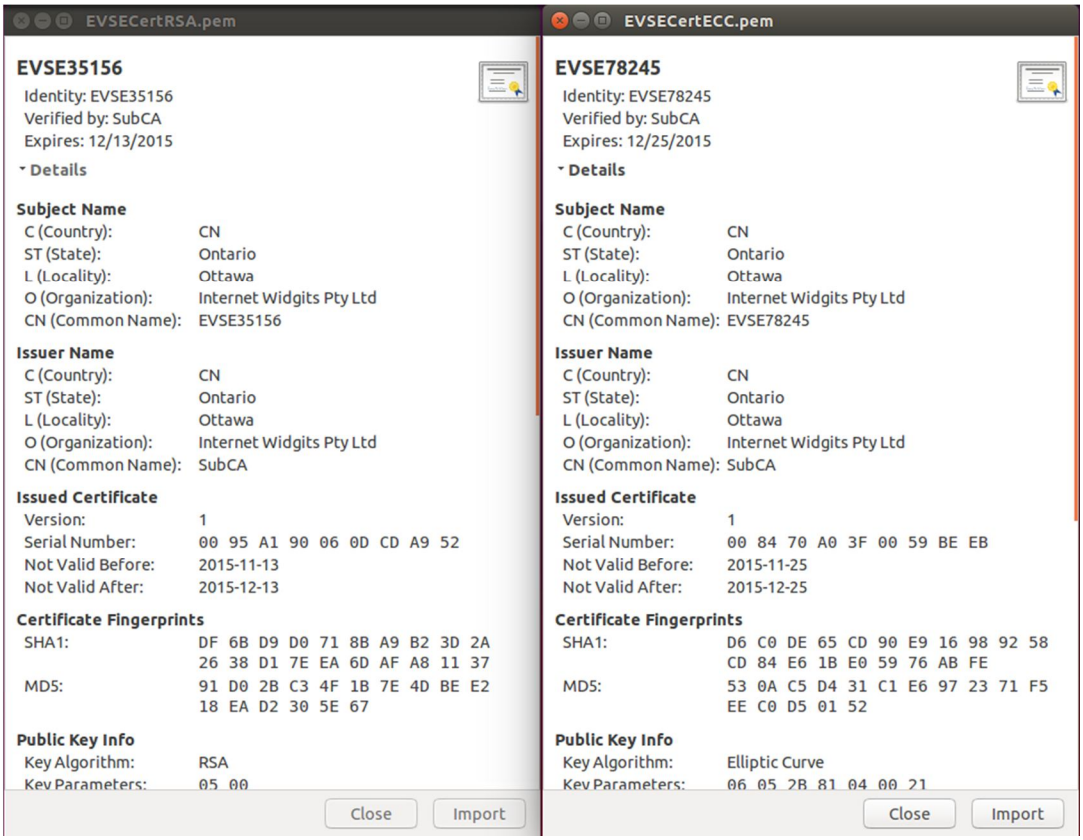


Figure 3-11: EVSE Contract Certificates for Vendor A and Vendor B

The self-signed Root Certificates have the Identity, Verified, Subject and Issuer field as themselves. For the Sub CA certificates, the Subject and Identity provided is the Sub CA itself, but the certificate is verified and issued by the Root CA. The identity and the subject used in OEM Provisioning Certificates are the EV VIN numbers. As for the identity and the subject fields within the EV and EVSE Contract Certificates, EMAID or contract Id is used. The OEM Certificate and Contract certificate is differentiated by their identity, with the VIN number and EMAID used in that field respectively. The VIN number and EMAID can be distinguuished based on their different formats. The OEM Provisioning Certificates are installed in the EV's upon production and are used at the initial stage of securely installing the Contract Certificate. Contract Certificates are used to authenticate EV's for charging. EVSE Certificates are used by EV to verify the authenticity of the EVSE.

We have built up towards our main system architecture by highlighting the Trust Model, X509 certificate and Database structures first. Now that we have stated them, the next section will take an overall look at the general system architecture from a broad-view and then examine the internal structures within the components in the general architecture. That is to say that the EV, EVSE and Secondary Actor (Vendors) components are shown in the general picture; the internal structure within the EV-EVSE and the Secondary Actor will be discussed and concluded with for this chapter. The use

of Root CA, Sub CA, OEM and EV Contract Certificates within the hierarchy model are based on the ISO 15118 standard. However, contrary to the ISO 15118 standard, the same Root CA used for providing EV Contract Certificates has been used to certify EVSE certificates. This reduces the overall hierarchy size and is applicable for use in a small scale model.

3.2.4 Overall System Architecture

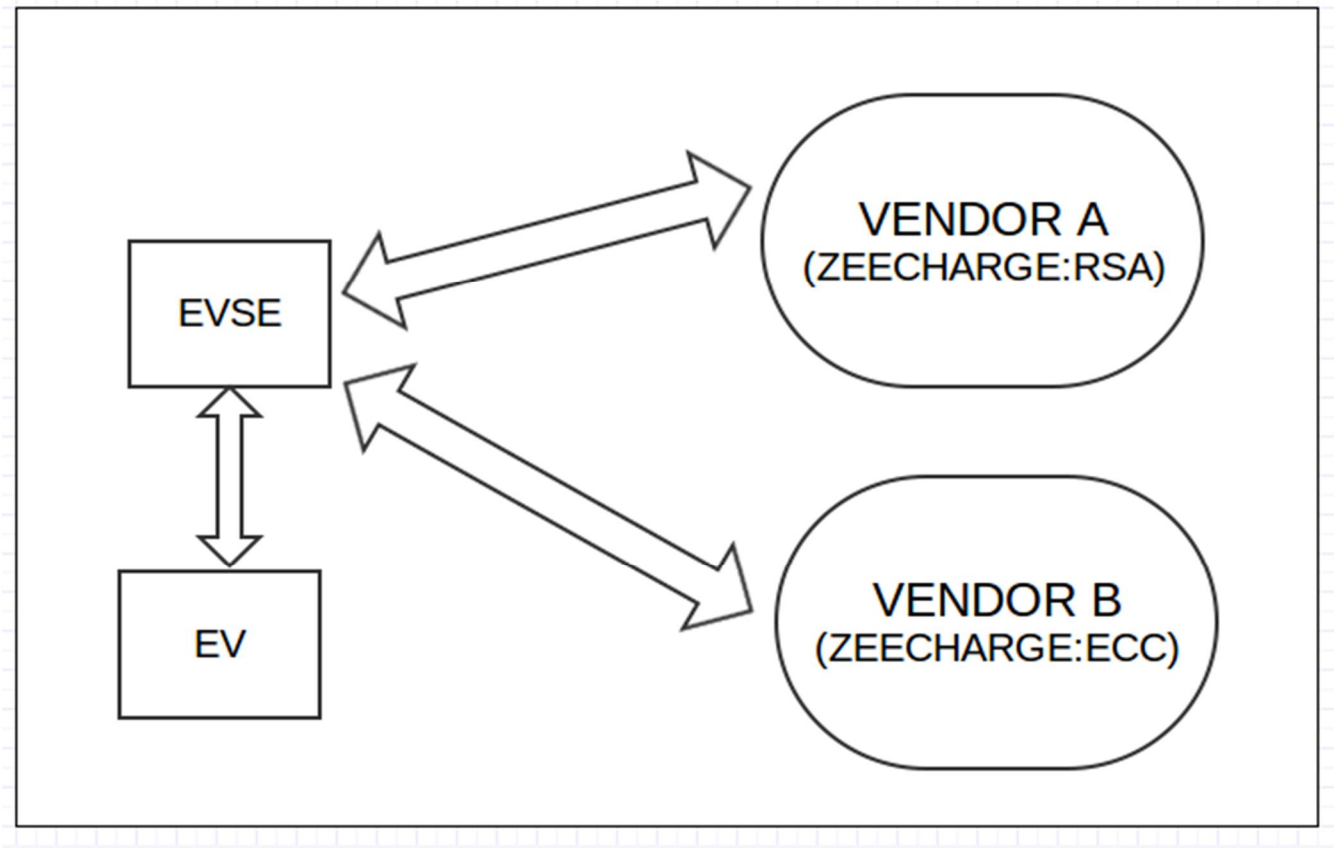


Figure 3-12: Overall System Architecture

Figure 3-12 shows the overall framework of our system from a broad-view. The authentication stage is initiated from the left hand side of the architecture, with the handshaking between the EV and EVSE to establish connection between them. Then, the EVSE acts as the client and the EV as the server, wherein the client (EVSE) always dictates the actions to take place among the two controllers. The operations gradually move to the right hand side of the architecture, and hits the service of the Application Server of either of the Secondary Actors depending on which vendor the EV is tied to.

Once service is requested from Application Server, the Secondary Actor takes care of either the Contract Certificate generation or validation and then returns the appropriate response to the EVSE.

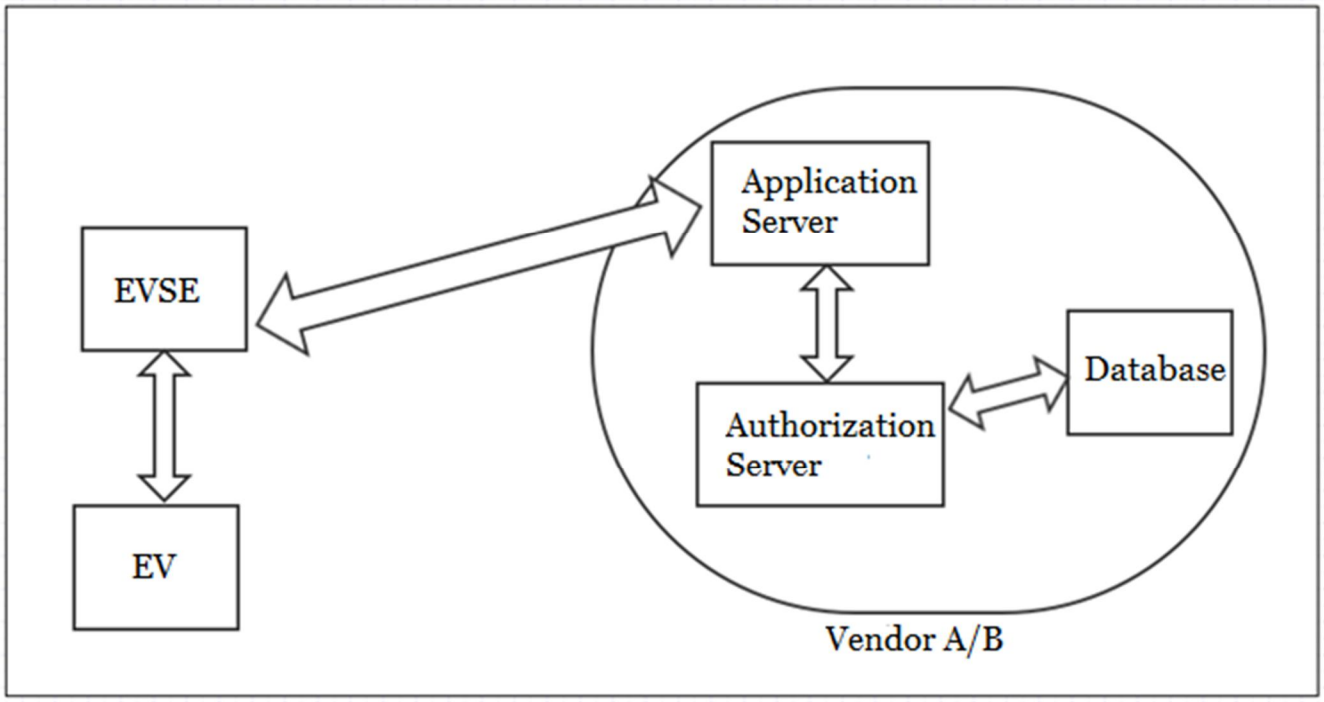


Figure 3-13: Internal architecture within Secondary Actor

Figure 3-13 provides an in-depth view of the architecture within the Secondary Actors (Vendor A/B). Each of the Secondary Actors has three main components within it: the Application Server, Authorization Server and the Database Server. The Application Server acts as the medium of communication between the user end (EV-EVSE side) and the management end (Secondary Actor). The Application Server is responsible for receiving requests from EVSE and directs the Authorization Server to carry out the necessary services as per the request from EVSE. The main function of the Authorization Server is to check for the authenticity and integrity of the Contract Certificate by verifying its Contract Certificate chain. The other functions of Authorization Server include services such as querying Database Server to check if EV is registered or if Contract Certificate exists, creating Contract Certificate for EV, encrypting confidential data based on either RSA asymmetric encryption (for Vendor A) or AES symmetric encryption (for Vendor B), etc. The Database Server can only be accessed by the Authorization Server directly and Authorization Server can only be accessed directly by the Application Server as shown in the figure above. The Authorization Server performs necessary tasks and returns the necessary response to the Application Server which is then forwarded to the EVSE, and eventually from EVSE to EV.

The steps mentioned above have been highlighted via a flow diagram for the same process as shown in Figure 3-14.

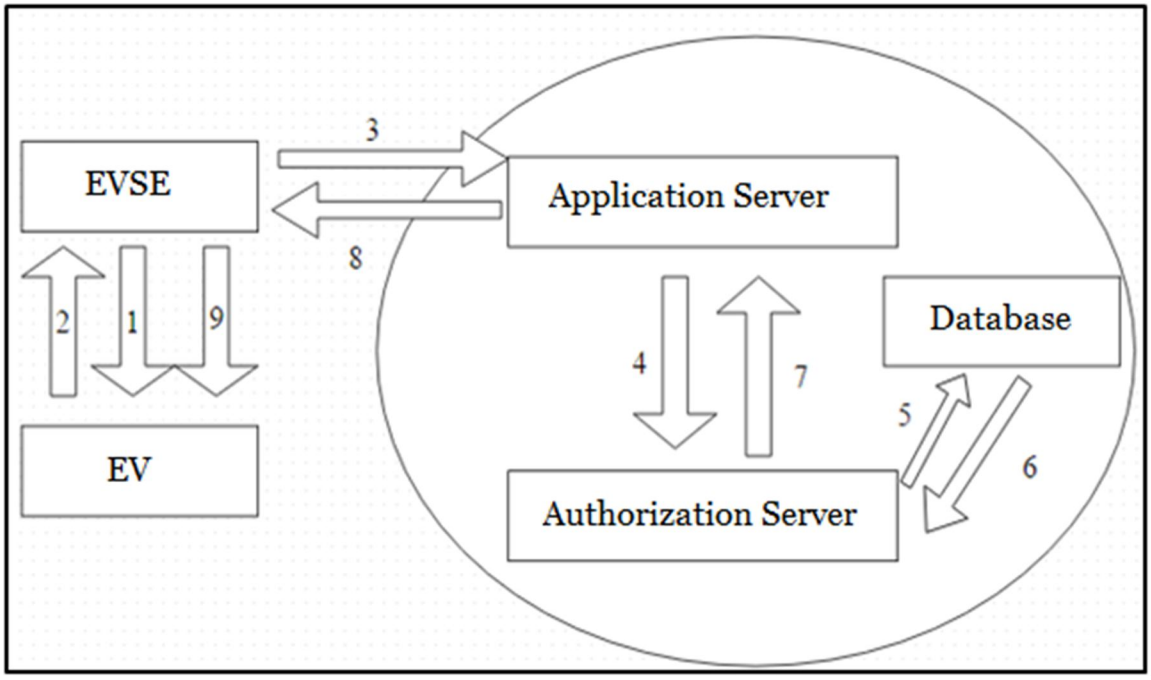


Figure 3-14: System Architecture Flow Diagram

As shown in the figure above, step 1 is communication initiated by the client (EVSE) with the server (EV). Step 2 is the response of the server to the client's request. Step 3 is the request forwarding from EVSE to Application Server (Vendor A/B), step 4 is directing request over to Authorization Server, step 5 is querying the Database by the Authorization Server, step 6 is the Database returning the response to query, step 7 is the Authorization Server returning the response to the original request from Application Server, step 8 is the forwarding of the response from the Application Server to the EVSE and step 9 ends with forwarding the response back from EVSE to EV. Figure 3-15 below shows the internal architecture within EVSE controller:

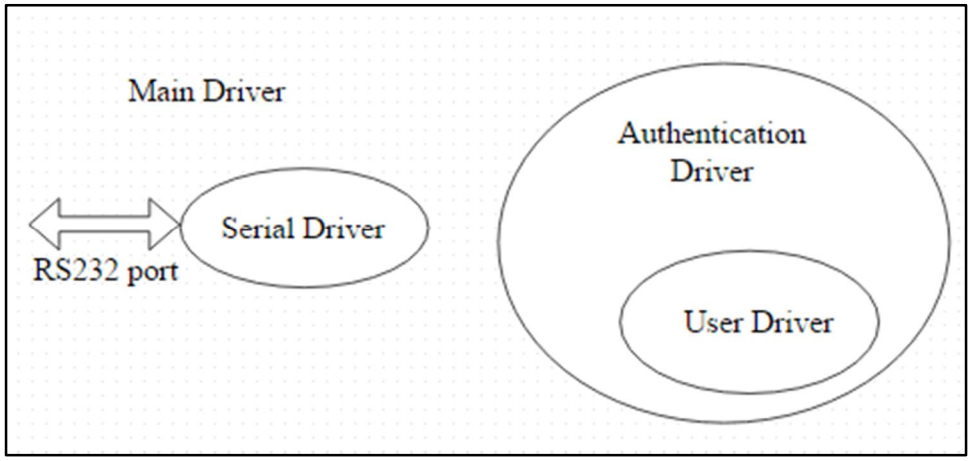


Figure 3-15: Internal architecture within EVSE controller

Four main drivers are involved in the authentication phase of the system. Each of the drivers has its own mailbox Id associated with the drivers. The mailbox Id provides the address for each of the drivers, in case a message needs to be sent to any of them. The main driver is the primary driver which contains the other drivers within it. The Serial Driver is used to serially communicate any messages over to the EV board. The Authentication Driver is used to direct any authentication-related messages/requests from the user. In the context of the authentication scenario, the User Driver functions inside the Authentication Driver. Each of the drivers has its own threads running. Figure 3-16 shows the threads associated with the drivers, their relations and dependencies, and the flow process between the threads upon any user service (client) required.

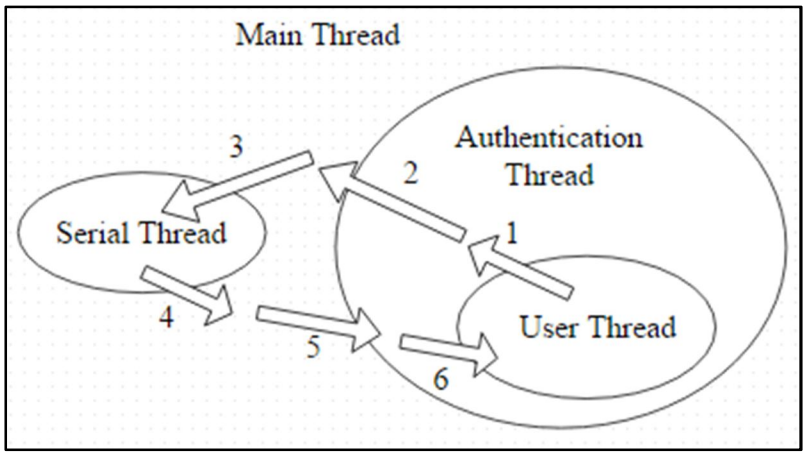


Figure 3-16: Thread diagram for EVSE architecture

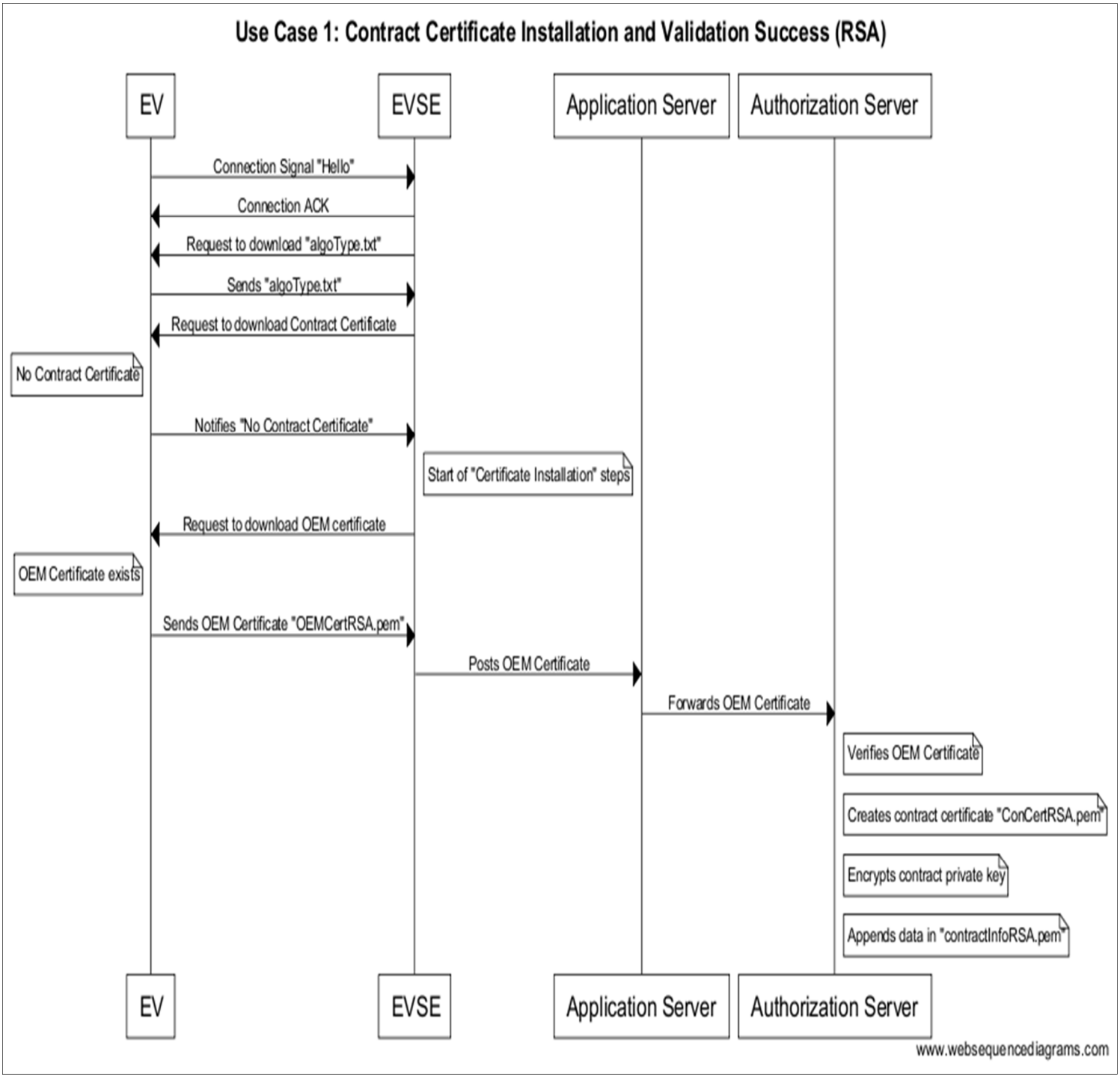
As can be seen from the figure, each driver has its own threads running at the back end independently. However, the main thread starts initially upon EVSE start up. Then all the other threads start. Upon the client/user requesting an authentication related service, the message flows from one thread to another.

Let us consider a scenario where the user/client requests for a file to be uploaded to the server (EV). This request contains the name of the file to be uploaded. This request is inserted inside a sendbox structure (C/C++ structure type) with the destination address set to the mailbox Id of the Authentication Driver. So the upload request together with the filename gets forwarded to the Authentication Driver. The Authentication thread handles the request accordingly and forwards the request to the Main Driver. The main thread running in the Main Driver further forwards the request to the Serial Driver. The serial thread operating within the Serial Driver sends the requested file over to EV serially via the RS232 cable and then sends a response back the same way to where it initially arrived. So the response goes back to the main driver which forwards to the authentication driver which finally forwards to the user driver. At this stage, the process is complete and the request is taken care of. The internal architecture within EV is similar to that of EVSE. Hence, any further explanation of the EV architecture is unnecessary and redundant.

Chapter 4 - Implementation of the Multi-Vendor Model

4.1 Use Case Sequence Diagrams

4.1.1 Use Case 1: Certificate Installation and Validation Success (RSA)



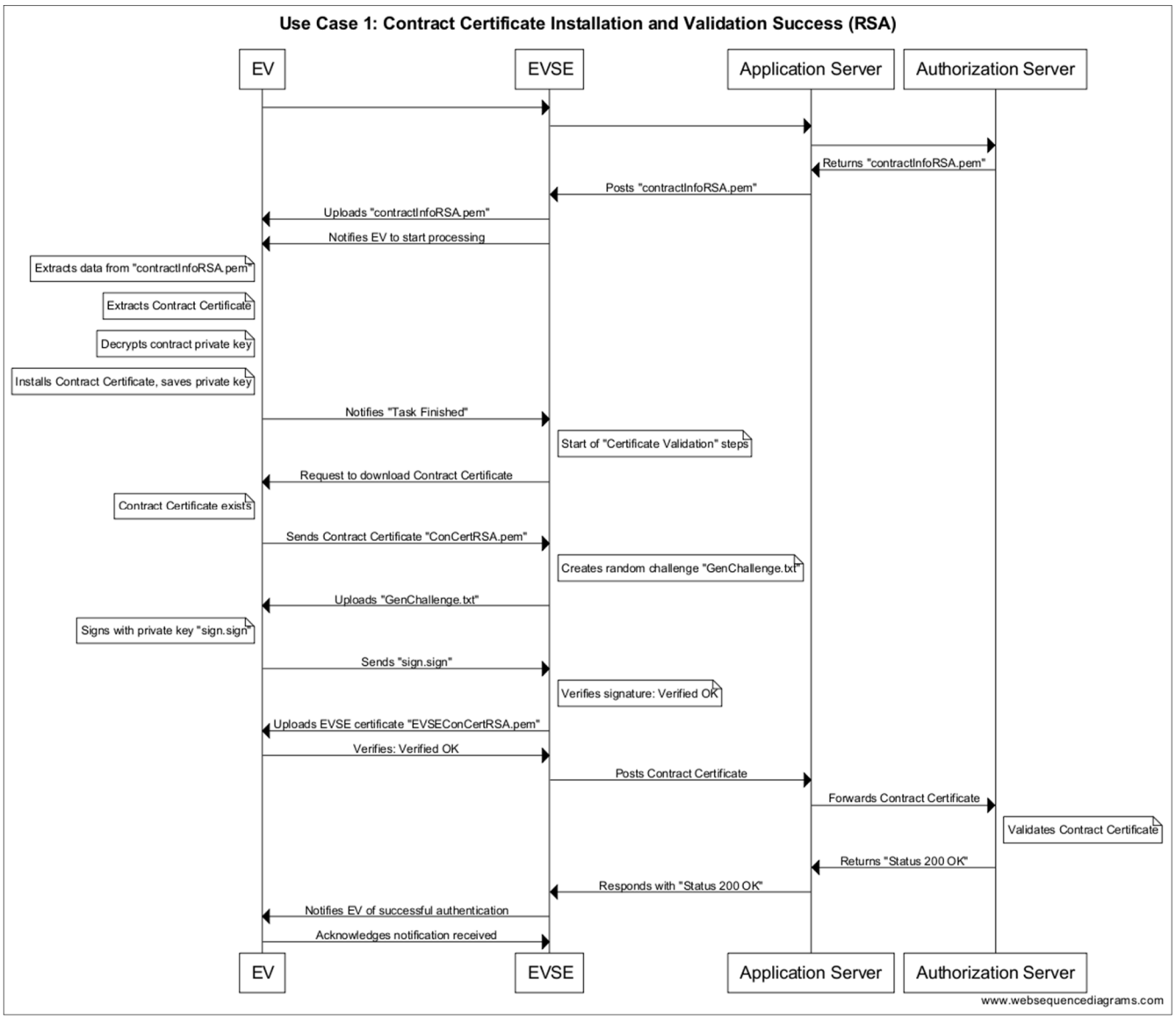


Figure 4-1: Use Case 1 (Certificate Installation and Validation with RSA)

Figure 4-1 shows the first use case scenario with respect to Vendor A. This is the case where an EV is registered in the system but has yet to charge its vehicle. So, for the first time it plugs in to a charging station (EVSE), it uses its OEM certificate to confirm its registration and hence receives a contract certificate. Having received the contract certificate, it then installs the contract certificate and saves the private key along with it.

Prior to running down the steps for use case 1, for this scenario to happen successfully, the EV must

have its OEM certificate, OEM private key and “algoType.txt” file installed in its system. The OEM certificate and the private key are needed for confirming EV registration in the system and decrypting the contract private key using the OEM private key respectively. The “algoType.txt” file has to exist in every EV indicating whether it belongs to Vendor A or Vendor B. Using the information from “algoType.txt” file, the EVSE can thus know which algorithm to follow (RSA or ECC) and adjust accordingly. The steps for use case 1 are as follows:

1) There is handshaking involved between EVSE and EV to establish communication once EV is plugged in to EVSE.

2) EVSE asks for the “algoType.txt” file. EV sends the file. EVSE reads the contents inside the file and thereby knows which vendor the EV belongs to. The contents within the “algoType.txt” file can include the vendor name in case there are multiple vendors using the same cryptographic algorithm.

3) At this point, EVSE does not know whether the EV has contract certificate installed or not. However, EVSE always asks for the contract certificate first since it’s the most common scenario to occur. If contract certificate does not exist, it then knows that the EV needs to authenticate using its OEM certificate. However, normal case of certificate validation using contract certificate will occur.

4) EV informs EVSE that contract certificate has not been installed yet. So this time, EVSE asks for the OEM certificate. EV sends its OEM certificate “OEMCertRSA.pem” file accordingly.

5) EVSE posts the OEM certificate to the Application Server at the vendor end.

6) The Application Server forwards the OEM certificate to the Authorization Server and asks for service from the Authorization Server.

7) The Authorization Server does the following tasks:

a) Verifies whether OEM certificate can be trusted

b) Queries database server to check if EV is registered in the system (Vendor A)

c) If verified and registration confirmed, a contract certificate with private key is created

d) Using RSA algorithm, private key is encrypted (details provided later)

e) New contract certificate, encrypted private key and other important data are all appended in a

file called “contractInfoRSA.txt” using “$” delimiter

8) The Authorization Server provides Application Server with the file “contractInfoRSA.txt”, also indicating successful authorization and contract certificate installation.
9) The Application Server responds to EVSE request with the “contractInfoRSA.txt” file.

10) The EVSE uploads the “contractInfoRSA.txt” file to EV and tells EV to start its task of extracting and saving important data.

11) The EV then receives the file and does the following tasks:

a) Extracts all data using “$” delimiter

b) Saves contract certificate as “ConCertRSA.pem” file

c) Decrypts contract private key using RSA algorithm (details explained later)

d) Saves private key as “prConRSA.pem” in the system

12) The EV responds to EVSE indicating that its task is done. Thus, at this point, the certificate installation scenario is done and certificate validation scenario begins.

13) EVSE asks for the contract certificate again. But this time, EV has a contract certificate installed. So EV sends the contract certificate to EVSE.

14) At this point, having received EV's contract certificate, EVSE has to verify that the EV being interacted with is actually the owner of the contract certificate. It does this by performing the “signing and verification” method. EVSE creates a random challenge and appends in a file “GenChallenge.txt”. The file is uploaded to EV.

15) EV extracts the random challenge, uses its contract private key to sign the random challenge and creates a file “sign.sign”. It then sends the file to EVSE.

16) EVSE verifies the signature and matches with the original random challenge to see if there is a match. This verifies that the EV is the owner of the contract certificate and has its private key in possession. The EVSE then uploads its own certificate (EVSE certificate) to EV. The EV checks for the authenticity and trustworthiness of the EVSE certificate by verifying the EVSE certificate chain.

17) EVSE then posts the contract certificate to the Application Server.

18) The Application Server then forwards the contract certificate to the Authorization Server and asks for required service from it.

19) The Authorization Server does the following tasks:

a) Verifies the Contract Certificate chain to check for the authenticity of the certificate and whether it can trust the certificate.

b) Queries database server to check if contract certificate exists and is validated

20) Once verified and authorized, it returns a status 200 message on file to Application Server.

21) The Application Server responds to EVSE request with a status 200 indicating that contract certificate has been authorized and is ready for charging.

22) The EVSE notifies the EV that authentication has been successful. The EV acknowledges that notification has been received. This ends the whole authentication process. From this point one, the EV can start charging. However, the charging section is out of the scope of our thesis.

4.1.2 Use Case 2: Certificate Validation Success (RSA)

Use Case 2 authentication scenario also involves dealing with Vendor A (using RSA algorithm). However, this time, the certificate installation section is skipped and contract certificate validation is directly done. The result returned is status 200 indicating authentication is successful.

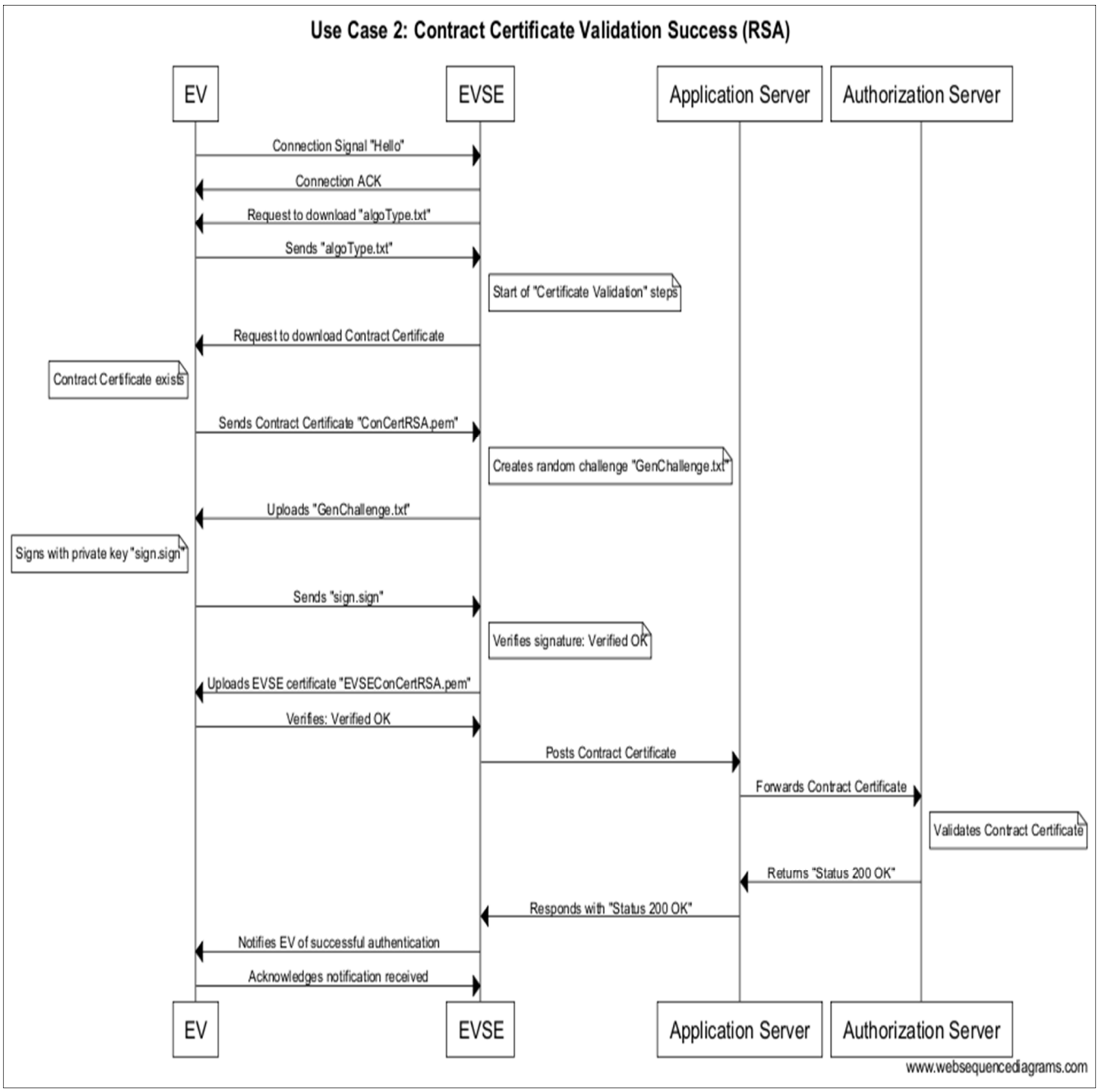


Figure 4-2: Use Case 2 (Certificate Validation with RSA)

Figure 4-2 above shows the scenario where contract certificate validation directly takes place with Vendor A involved in the process. Contrary to Use Case 1, after asking for “algoType.txt” file and confirming association of EV with Vendor A, the EVSE again asks for contract certificate from the EV and this time the EV is able to send its contract certificate since it has already been installed previously. Use Case 2 is the common occurring case for EV authentication for Vendor A.

For Use Case 2, it is assumed that EV has previously charged from one of the EVSEs available and has undergone the certificate installation process where it used its OEM certificate in order to install its contract certificate and private key in its system. The steps in this use case are similar to half of the later half portion of Use Case 1, except that contract certificate is checked in the beginning and is received by EVSE. The steps for Use Case 2 are as follows:

1) There is handshaking involved between EVSE and EV to establish communication once EV is plugged in to EVSE.

2) EVSE asks for the “algoType.txt” file. EV sends the file. EVSE reads the contents inside the file and thereby knows that EV is associated with Vendor A.

3) EVSE asks for contract certificate. Contrary to Use Case 1, EV has the contract certificate this time and thus is able to provide EVSE with its contract certificate.

4) Once again, having received EV's contract certificate, EVSE has to verify that the EV being interacted with is actually the owner of the contract certificate. It does this by undergoing the “signing and verification” process. EVSE forms a random challenge and inserts the challenge in a file named “GenChallenge.txt”. The file is uploaded to EV.

5) EV extracts the random challenge, uses its contract private key to sign the random challenge and creates a file “sign.sign”. It then sends the file to EVSE.

6) EVSE verifies the signature and matches with the original random challenge to see if there is a match. The EVSE then uploads its own certificate (EVSE certificate) to EV. The EV checks for the authenticity and trustworthiness of the EVSE certificate by verifying the EVSE certificate chain. The EV notifies the EVSE that the authentication process can carry on.

7) EVSE then posts the contract certificate to the Application Server.

8) The Application Server then forwards the contract certificate to the Authorization Server and asks for required service from it.

9) The Authorization Server does the following tasks:

a) Checks the authenticity and trustworthiness of the contract certificate by verifying the EVSE certificate chain.

b) Queries database server to check if contract certificate exists and is validation

10) Once verified, it returns a status 200 message on file to Application Server.

11) The Application Server responds to EVSE request with a status 200 indicating that EV is ready for charging.

12) The EVSE notifies the EV that authentication has been successful. The EV acknowledges that notification has been received. This ends the contract certificate validation process for Vendor A. This is the same authentication process that takes place every time an EV plugs in to an EVSE.

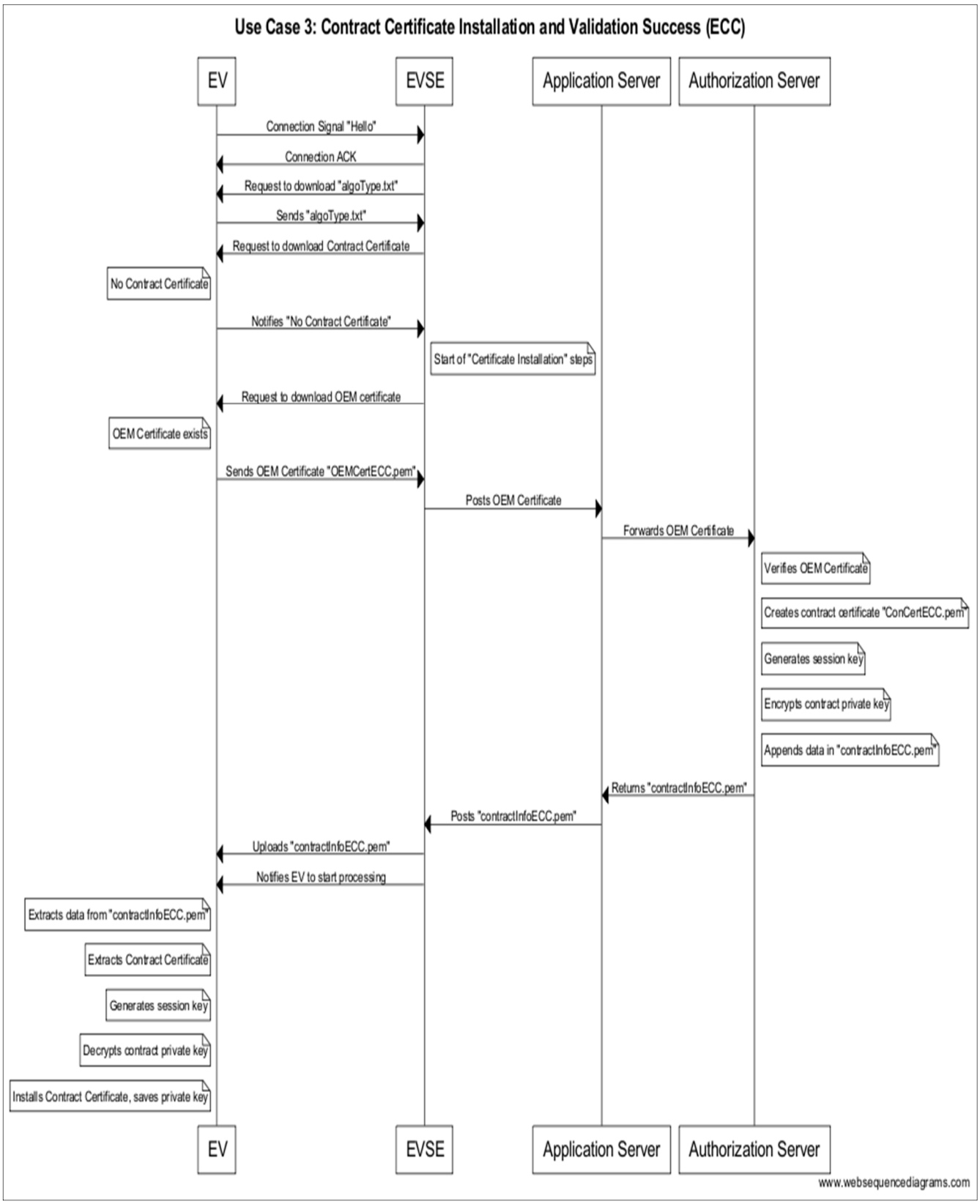
4.1.3 Use Case 3: Certificate Installation and Validation Success (ECC)

Use Case 3 deals with the Certificate Installation and Validation process, except that the concerned vendor dealt with is Vendor B. And with Vendor B, as we know, the ECDH (ECC with Diffie-Hellman) algorithm is used for key generation, signing, verification, encryption and decryption purposes.

The main difference in the way the two vendors (Vendor A and B) operate is in the way the important data are encrypted and decrypted. The authentication delay will also vary depending on which vendors the EV is associated with and the authentication process is done with. It can be said very clearly and we will be seeing in the performance evaluation section later, that the ECDH algorithm is able to get its encryption and decryption tasks accomplished faster than RSA.

Also, it must be kept in mind that the digital certificates used for the authentication and authorization process for both vendors (Vendor A and Vendor B) are different. We have tried to create a clear way of distinguishing the certificates used for Vendor A and those used for Vendor B by putting their algorithm names (RSA or ECC) at the end of the filename. For example, notice that the contract private key generated using the RSA algorithm has been named “prConRSA.pem” while that generated by the ECC algorithm has been named “prConECC.pem”. Figure 4-3 below shows the sequence diagram for Use Case 3 where contract certificate installation and validation both takes place one after another, and

Vendor B (ECDH algorithm) is the vendor.



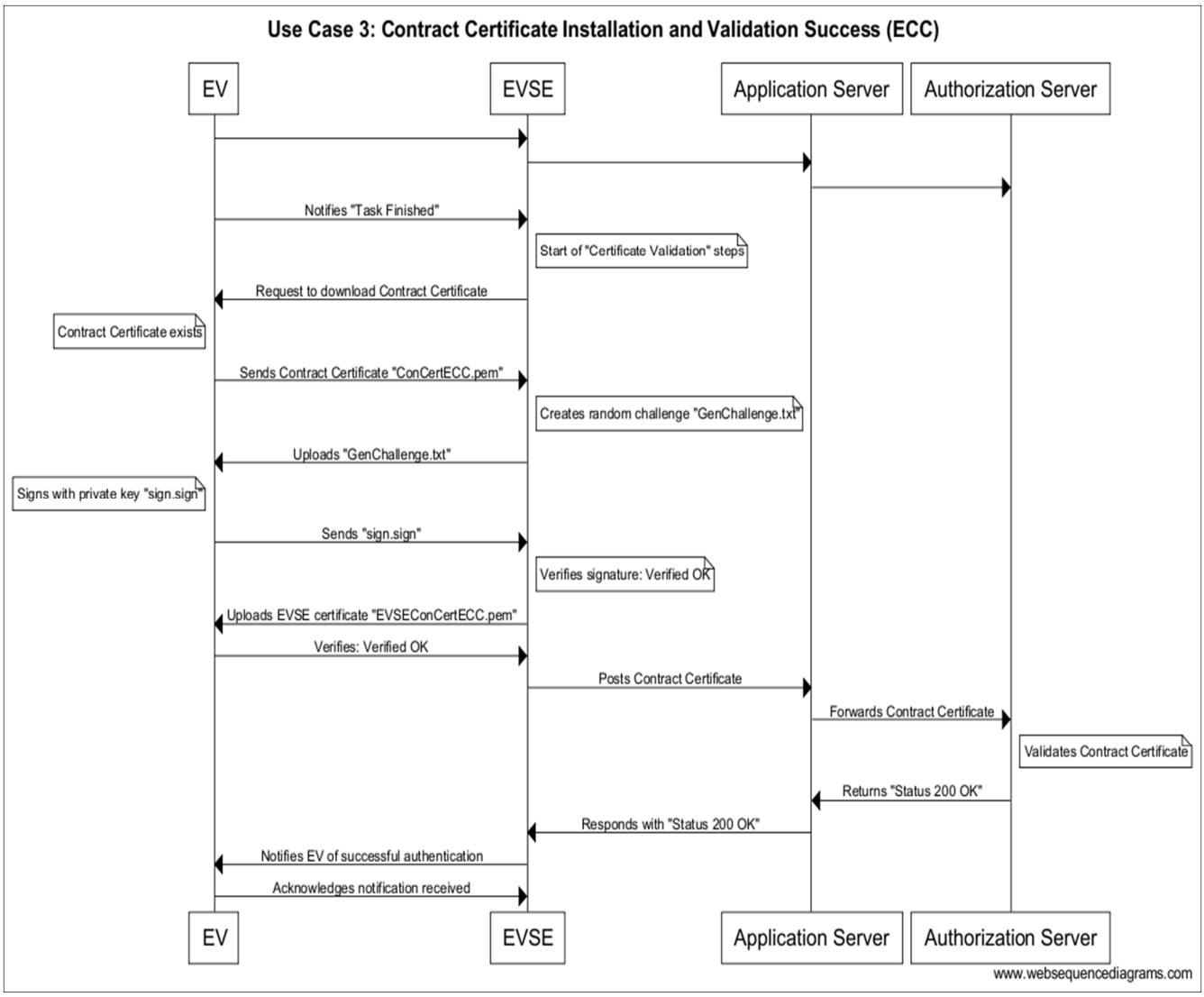


Figure 4-3: Use Case 3 (Certificate Installation and Validation with ECDH)

The steps for use case 3 are as follows:

1) EVSE and EV detect one another's presence and perform handshaking.

2) EVSE requests for the file “algoType.txt”. The file “algoType.txt” for the EV should contain “ECC” as the algorithm type since it belongs to Vendor B. EV uploads the file over to EVSE.

3) EVSE requests for EV's contract certificate.

4) EV informs EVSE that contract certificate has not been installed yet. Having received this response,

EVSE asks for the OEM certificate. EV sends its OEM certificate “OEMCertECC.pem” file accordingly. Notice how the filename for OEM Certificate is different compared to the one used for Vendor A.

5) EVSE posts the OEM certificate to the Application Server at the vendor end via HTTP protocol.

6) The Application Server sends the OEM certificate to the Authorization Server and requests for OEM certificate processing service from the Authorization Server.

7) The Authorization Server does the following tasks:

a) Verifies whether OEM certificate can be trusted

b) Queries database server to check if EV is registered in the system (Vendor B)

c) If verified and registration confirmed, a contract certificate with private key is generated

d) Using ECDH algorithm, private key is encrypted (details provided later)

e) New contract certificate, encrypted private key and other important data are all appended in a file called “contractInfoECC.txt” using “$” delimiter

8) The Authorization Server provides Application Server with the file “contractInfoECC.txt”, also indicating contract certificate has been created successfully.

9) The Application Server responds to EVSE request with the “contractInfoECC.txt” file.

10) The EVSE uploads the “contractInfoECC.txt” file to EV and tells EV to start its task of extracting and installing the certificate.

11) The EV then receives the file and does the following tasks:

a) Extracts all data using “$” delimiter

b) Saves contract certificate as “ConCertECC.pem” file

c) Decrypts contract private key using ECDH algorithm (details explained later)

d) Saves private key as “prConECC.pem” in the system

12) The EV responds to EVSE indicating that its task is done. Thus, at this point, the certificate installation scenario is done and the certificate validation scenario begins.

13) EVSE requests EV to send the contract certificate again. But this time, EV has a contract certificate installed. So EV sends the contract certificate “ConCertECC.pem” to EVSE.

14) EVSE generates a random challenge and stores it in the “GenChallenge.txt” file. The file is uploaded to EV.

15) EV extracts the random challenge, uses its contract private key to sign the random challenge and creates a file “sign.sign”. It then sends the file to EVSE. Notice how the “GenChallenge.txt” and “sign.sign” have the same filenames for both Vendor A and Vendor B. This has been kept the same since changing the filenames is not really necessary in this case and does not require being distinguished.

16) EVSE verifies the signature and matches with the original random challenge to see if there is a match. This verifies that the EV is the owner of the contract certificate and has its private key in its possession. The EVSE then uploads its own certificate (EVSE certificate) to EV. The EV checks for the authenticity of the EVSE certificate by verifying the EVSE certificate chain and also whether it can trust the certificate. The EVSE is then notified and can carry on with its other operations.

17) EVSE then posts the contract certificate “ConCertECC.pem” to the Application Server.

18) The Application Server then forwards the contract certificate to the Authorization Server and asks for the necessary services.

19) The services from Authorization Server include:

a) Verifies whether the Contract Certificate can be trusted.

b) Querying database server to check if contract certificate exists and is validated

20) Once verified and authorized, it returns OK with status 200.

21) The Application Server responds to EVSE request with a status 200 indicating that the contract certificate has been authorized and is ready for charging.

22) The EVSE notifies the EV that authentication has been successful. The EV acknowledges that

notification has been received. This ends the whole authentication process for EV having Vendor B as its vendor.

4.1.4 Use Case 4: Certificate Validation Success (ECDH)

Use Case 4 deals with the last and final successful authentication scenario, which also involves dealing with Vendor B (using ECDH algorithm). The certificate installation section is skipped and contract certificate validation is directly done. The result returned is acknowledgement status 200 indicating that authentication is successful.

Use Cases 2 and 4 are very similar in their operations. Both the cases involve authenticating and authorizing EV for charging during contract certificate directly, without having to install and deploy the contract certificate in the first place. The two operations are similar since no encryption and decryption takes place in these use cases. The steps from signing and verifying random challenge to verifying EVSE certificates, to authenticating contract certificates and returning 200 from the server side, are identical.

Figure 4-4 below provides the last and final successful authentication scenario within the broad context of our multi-vendor system. In this case however, the digital signature algorithm used is ECDSA with sha256 as compared to the digital signature algorithm used in Use Case 2, which was RSA with sha256.

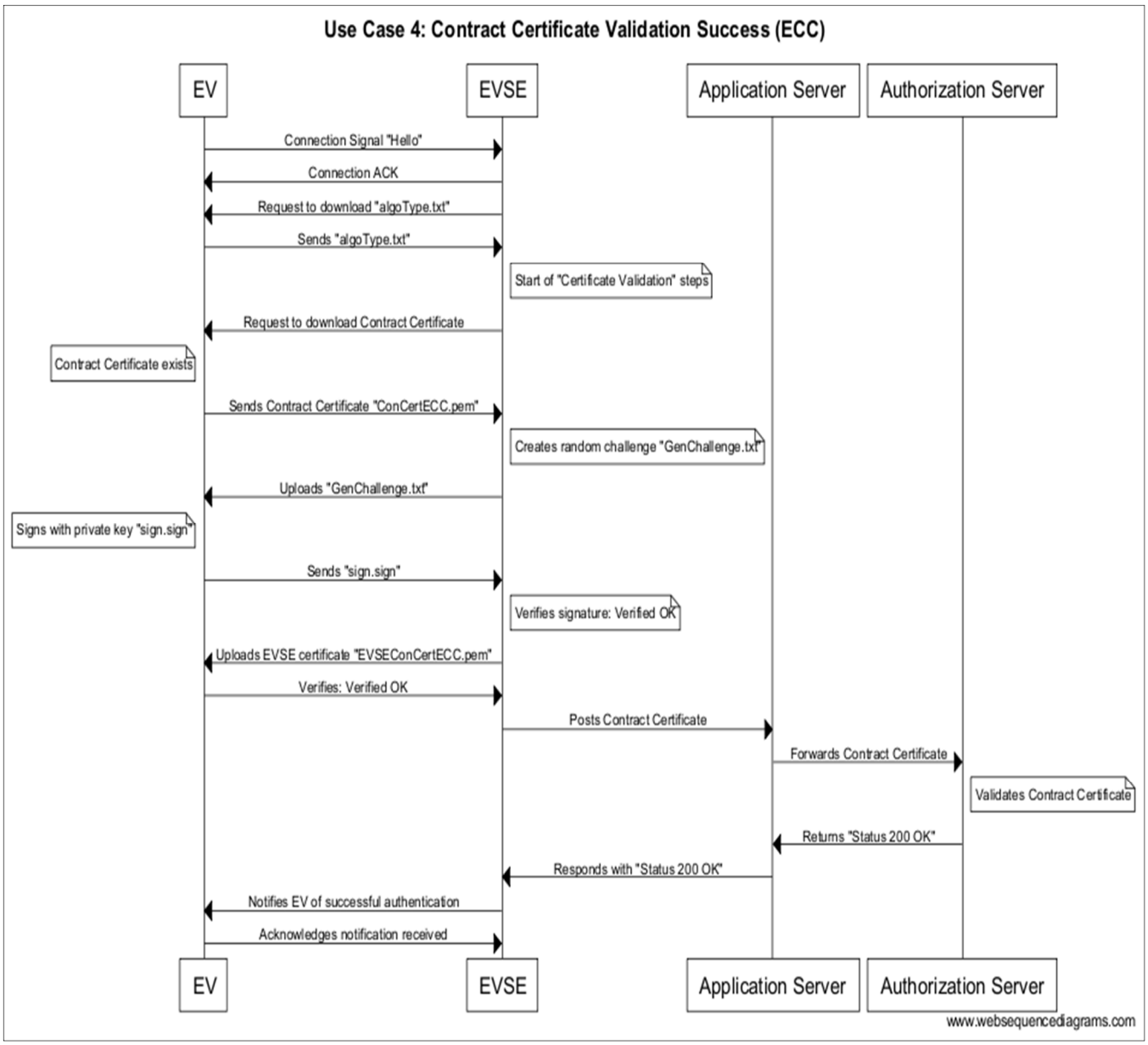


Figure 4-4: Use Case 4 (Certificate Validation with ECDH)

The steps for Use Case 4 are as follows:

1) As always, there is handshaking involved between EVSE and EV to establish communication once EV is plugged in to EVSE.

2) EVSE asks for the “algoType.txt” file. EV sends the file. EVSE reads the contents inside the file and thereby knows that EV is associated with Vendor B.

3) EVSE asks for contract certificate. Contrary to Use Case 3, EV has the contract certificate this time and thus is able to provide EVSE with its contract certificate. The contract certificate is named as “ConCertECC.pem”.

4) Once again, having received EV's contract certificate, EVSE has to verify that the EV being interacted with is actually the owner of the contract certificate. It does this by undergoing the “signing and verification” process. EVSE forms a random challenge and inserts the challenge in the “GenChallenge.txt” file. The file is uploaded to EV.

5) EV extracts the random challenge, uses its contract private key to sign the random challenge and creates a file “sign.sign”. The digital signature algorithm used is ECDSA with sha256. The file is then sent to EVSE.

6) EVSE verifies the signature and matches with the original random challenge to see if there is a match. The EVSE then uploads its own certificate (EVSE certificate) to EV. The EV checks for the authenticity of the EVSE certificate by verifying the EVSE certificate chain and also whether the certificate can be trusted. The EV then notifies the EVSE that the authentication process can carry on.

7) EVSE then posts the contract certificate to the Application Server.

8) The Application Server then forwards the contract certificate to the Authorization Server and asks for required service from it.

9) The Authorization Server does the following tasks:

a) Verifies whether the Contract Certificate can be trusted.

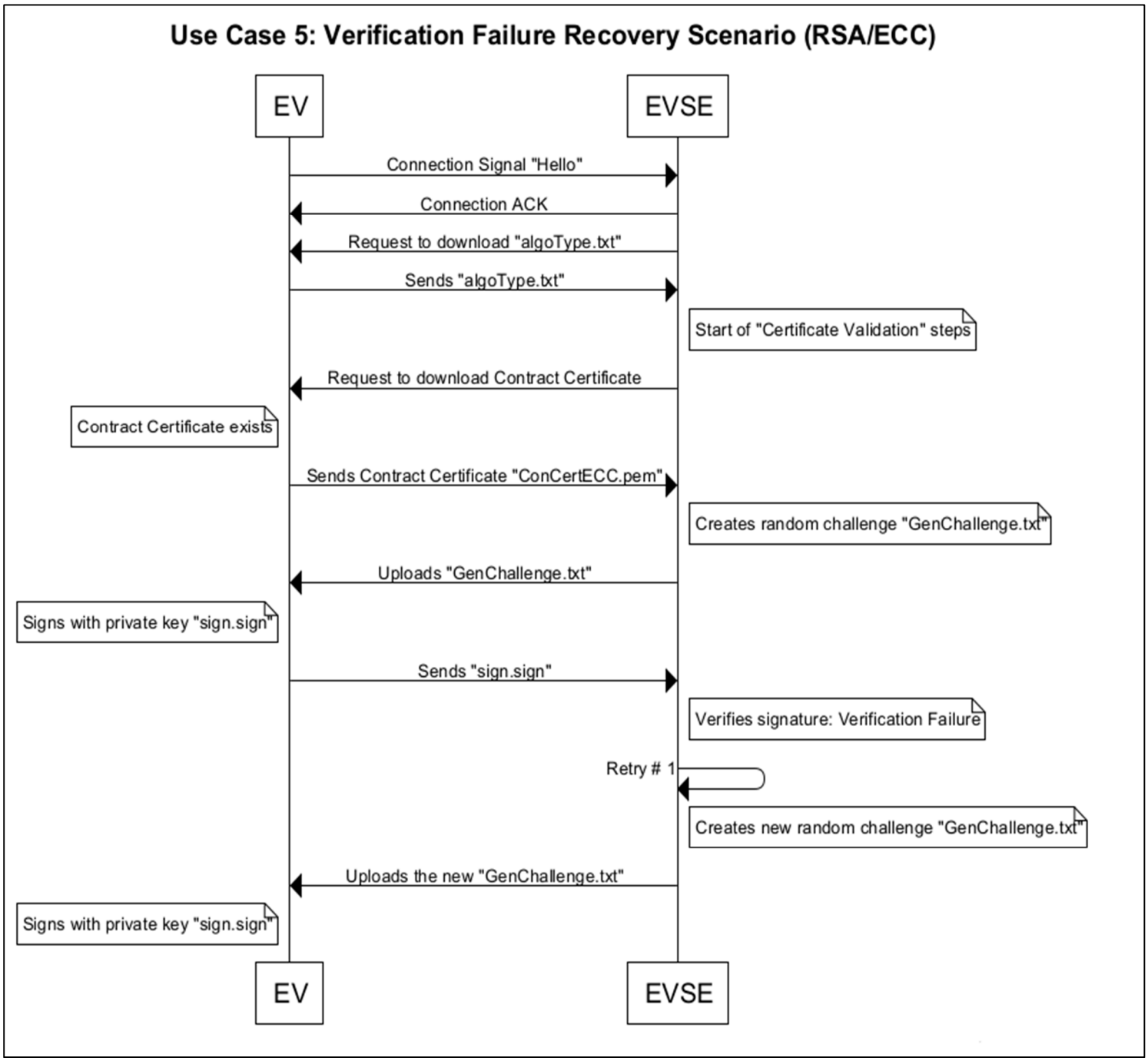
b) Queries database server to check if contract certificate exists and is validation

10) Once verified and authorized, it returns a status 200 message on file to Application Server.

11) The Application Server responds to EVSE request with a status 200 indicating that EV is ready for charging.

12) The EVSE notifies the EV that authentication has been successful. The EV acknowledges that notification has been received. This ends the contract certificate validation process for Vendor B.

4.1.5 Use Case 5: Common failure recovery scenario for both Vendor A and B



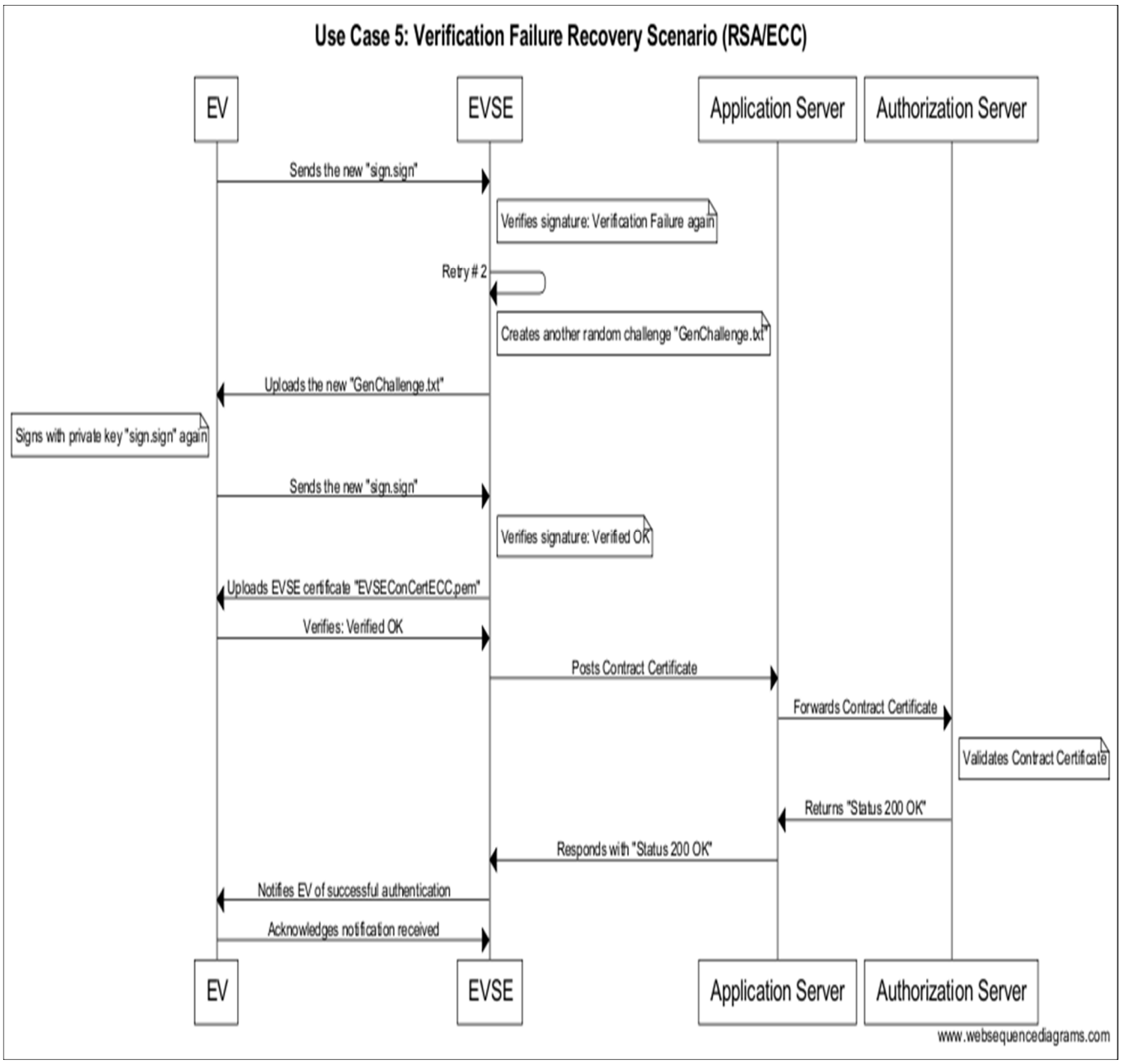


Figure 4-5: Common failure recovery scenario for both Vendor A and B

Figure 4-5 above shows the failure recovery scenario and how it is handled for all use cases. In this case, the failure involves verifying the digital signature in the “signing and verification” process. Unfortunately, this particular operation has been problematic and can be seen as a bug in the OpenSSL library. The verification process works most of the time. However, this problem has been dealt with and handled in such a way that the authentication process is bound to successfully execute and the verification process happens successfully if the entity truly is the owner of the contract certificate and private key.

The steps in this scenario are explained below:

1) The scenario shown above has been taken from either of the use cases 2 and 4. That is, it is assumed for the use case that contract certificate validation directly takes place.

2) As usual, handshaking takes place between EV and EVSE. Then the “algoType.txt” file is uploaded from EV to EVSE, notifying EVSE as to which vendor the EV is associated with. In this case, it could be Vendor A or Vendor B, and the failure recovery scenario would still be the same.

3) EV provides EVSE with the contract certificate after being requested. EVSE then creates the random challenge, saves it in the “GenChallenge.txt” file and uploads it to EV for signing.

4) EV provides signature for the random challenge (could be with RSA or ECDSA with sha256), and creates the “sign.sign” file. EV then uploads the “sign.sign” file to EVSE.

5) The steps are similar to use cases 2 and 4 so far. However, this time, it is assumed that EVSE fails to verify the digital signature when comparing with the original random challenge sent to EV.

6) EVSE manages a counter on its side. This scenario is dealt with by retrying the “signing the verification” scenario again. Thus, EVSE creates another random challenge and sends a new “GenChallenge.txt” to EV. EV signs, sends over the “sign.sign” file and yet again, according to the above scenario, it is assumed that EVSE fails to verify. EVSE increments the retry counter to two, indicating it is going to retry the “signing and verification” process for the second time. However, in this case, it is seen that digital signature has been verified. And from then on, the usual steps take place. Thus, the next step moves on to verifying the EVSE certificate and completing the authentication process eventually.

The failure recovery scenario has been tested numerous times and has been found to be verified successfully within the third retry every time. The retry limit has been set to 5. This means that if the “signing and verification” has been retried 5 times and EVSE fails to verify digital signature, then it must mean that there is an ownership issue between the EV and the contract certificate. This is to say that the authentication process failed since the EV has been found NOT to be the owner of the contract certificate.

The next section is going to look at the ways the important operations have been implemented in code.

4.2 Code implementations for important operations

4.2.1 Private-public key generation

The following code example below shows how a root certificate using RSA algorithm was created:

```
system("openssl genrsa -out privateRootCaRSA.pem 1024");
system("openssl req -sha256 -new -key privateRootCaRSA.pem -out rootreq.pem");
system("openssl x509 -req -in rootreq.pem -sha256 -signkey privateRootCaRSA.pem -out RootCACertRSA.pem");
```

Figure 4-6: Code Snippet 1

The code was implemented in C. The system() command was used to access OpenSSL tools from the command line operations from C. The private key for root certificate was created of size 1024 bits and named “privateRootCaRSA.pem”. Then, a CSR (certificate signing request) was created using the private key named “rootreq.pem”. The CSR was used to create the root certificate which was named “RootCACertRSA.pem”. The certificate was signed using its own private key created in the first step using RSA as signature algorithm with sha256.

The following code demonstrates how the Root Certificate was then installed into the system and configured to be recognized as a valid Root Certificate within the systems (EV, EVSE, Authorization Server):

```
system("sudo mkdir /usr/share/ca-certificates/extra");
system("sudo cp RootCACertRSA.pem /usr/share/ca-certificates/extra/RootCACertRSA.crt");
system("sudo dpkg-reconfigure ca-certificates");
system("sudo update-ca-certificates");
```

Figure 4-7: Code Snippet 2

To summarize the code above, a new directory “extra” was created inside the specified file directory location stated. The Root Certificate was copied from its location of origin to the newly created directory. The next command allowed for the reconfiguration of all the Root Certificates within the system. After this step, the root certificate created for Vendor A was selected from a list of root certificates available and permission was given to trust this certificate as a Root Certificate in the system. The next command updated all the Root Certificates.

The next code example demonstrates the creation of Sub CA certificate from the Root Certificate using ECC algorithm. The code snippet is taken during the creation of Sub CA certificate for Vendor B which

uses ECC algorithm. The root certificate had to be created prior to creating the Sub CA certificates. However, the creation of Root Certificate for Vendor B has been skipped it follows a similar style to that for Vendor A except with ECC instead of RSA. The following was the code implemented for Sub CA certificate creation using ECC cryptographic algorithm:

```
system("openssl ecparam -out privateSubCaECC.pem -name secp224r1 -genkey");
system("openssl req -sha256 -new -key privateSubCaECC.pem -out subreqECC.pem");
system("echo            | sudo -S openssl x509 -req -in subreqECC.pem -sha256 -CA /etc/ssl/certs/RootCACertECC.pem "
       "-CAkey privateRootCaECC.pem -CAcreateserial -out SubCACertECC.pem");
```

Figure 4-8: Code Snippet 3

In the code above, a private key was first generated of size 224 bits of domain parameters corresponding to “secp224r1” and was named “privateSubCaECC.pem”. The next command created a CSR based on the private key created. The next command used our system password (censored) for user permission to generate the Sub CA certificate and was named “SubCACertECC.pem”. Notice how the private key of Root Certificate (“privateRootCaECC.pem”) has been used to sign the Sub CA certificate. ECDSA with sha256 was the digital signature algorithm used. Also the “-CA” sets the trust hierarchy for the Sub CA certificate indicating the Root Certificate to be the issuer of the Sub CA certificate. The following code example shows how an OEM certificate was created using ECC algorithm:

```
system("openssl ecparam -out privateOEMECC.pem -name secp224r1 -genkey");
system(
        "openssl req -sha256 -new -key privateOEMECC.pem -out OEMreqECC.pem");
system(
        "openssl x509 -req -in OEMreqECC.pem -sha256 -CA SubCACertECC.pem "
        "-CAkey privateSubCaECC.pem -CAcreateserial -out OEMCertECC.pem");
```

Figure 4-9: Code Snippet 4

Similar to the previous steps, a private key was created for the OEM certificate and then a CSR created. Using the CSR, an OEM certificate was generated using the private key of the Sub CA certificate that was generated in the previous operations. This time, the Certificate Authority (CA) assigned to OEM certificate is the Sub CA certificate, which is also its issuer.

Similar steps were followed as above in order to create the contract and EVSE certificates. The Sub CA was assigned as the CA for each of the certificates, and each certificate was signed using the Sub CA’s private key.

4.2.2 RSA encryption and decryption

This section discusses the way encryption and decryption operations were carried out precisely using the RSA algorithm with respect to EVs associated with Vendor A. The encryption operation was carried out on the Authorization Server end. The code example shows the steps used to encrypt the contract private key:

```
//RSA encryption of aes-256-cbc key.txt by OEM RSA pubkey
system(
        "openssl rsautl -encrypt -pubin -inkey pubkey.pem -in key.txt > encKey.txt");
//Use another aes-256-cbc key to encrypt already encrypted key.txt---------
sprintf(command,
        "openssl aes-256-cbc -nosalt -a -e -in prConRSA.pem -out private_key.txt -K %s -iv %s",
        buf2, argv[1]);
//Use another aes-256-cbc key to encrypt already encrypted key.txt-------
sprintf(command,
        "openssl aes-256-cbc -nosalt -a -e -in encKey.txt -out encDouble.txt -K %s -iv %s",
        buf1, argv[1]);
```

Figure 4-10: Code Snippet 5

There are three major steps that were followed for encryption purposes. They are as follows:

1) A 32-byte long (256 bits) randomly generated hex value was saved in file “key.txt”. In the first command, the Authorization Server extracts the public key from OEM certificate and saves it as “pubkey.pem”. RSA encryption is then performed to encrypt the random hex value in “key.txt” using the OEM public key and the encrypted hex is then saved to a file named “encKey.txt”.

2) An AES symmetric encryption is performed to encrypt the contract private key “prConRSA.pem” using the 256-bit key generated in “key.txt” and a randomly generated 128-bit Initialization Vector (IV). The encrypted contract private key is saved as “private_key.txt” file.

3) Another randomly generated 256-bit long key is generated and stored as “key2.txt”. This key is used to perform AES symmetric encryption on the already encrypted “key.txt” and the new encrypted ciphertext is saved as “encDouble.txt”. The same 128-bit long IV is used for this purpose too. This extra step is done in order to arrange the ciphertext in an organized manner so that it is readable and can be transferred over the Internet using HTTP protocol.

So, in summary, we have the three encrypted files:

1) “encKey.txt”: 256-bit long key in “key.txt” encrypted by OEM public key using RSA algorithm.

2) “private_key.txt”: encrypted contract private key symmetrically encrypted using 256-bit long key in

“key.txt”.

3) “encDouble.txt”: content in “encKey.txt” symmetrically encrypted further using new 256-bit long key in “key2.txt”.

Then the contents in “encKey.txt”, the encrypted contract private key, the two 256-bit long keys, the 128-bit long IV and the contract certificate is appended in a file named “contractInfoRSA.txt” using “$” as a delimiter. This file is sent from the Authorization Server side to the EV side.

Over on the EV side, once the “contractInfoRSA.pem” file is received, the decryption process is in three stages as shown by the code implementation below:

```
//Use aes-256-cbc key to decrypt encDouble.txt to get encKey.txt---------
sprintf(command3,
        "openssl aes-256-cbc -nosalt -a -d -in encDouble.txt -out encKey2.txt -K %s -iv %s",
        buf3, ivEVRSA);
//Extract original aes-256-cbc key
system(
        "openssl rsautl -decrypt -inkey privateOEMRSA.pem -in encKey2.txt -out KEY.txt");
//Use aes-256-cbc key to decrypt private_key.txt to get Contract Private Key---------
sprintf(command3,
        "openssl aes-256-cbc -nosalt -a -d -in private_key.txt -out privateKEY.pem -K %s -iv %s",
        buf4, ivEVRSA);
```

Figure 4-11: Code Snippet 6

The three stages in decrypting the contract private key are in reverse order of the encryption process and are as follows:

1) Symmetric decryption is performed on the contents inside “encDouble.txt” using the 256-bit long key in “key2.txt” and 128-bit long IV generated on server side to get the RSA-encrypted 256-bit long key.

2) The RSA-encrypted 256-bit long key is decrypted using RSA algorithm using the OEM private key “privateOEMRSA.pem” on EV side. The key is then saved in “KEY.txt” file.

3) The 256-bit long key in “KEY.txt” and 128-bit IV from server side is now used to perform symmetric decryption on the encrypted contract private key to retrieve the contract private key which is now saved as “privateKEY.pem” on EV side. The private key is later renamed to “prConRSA.pem”.

4.2.3 ECDH encryption and decryption

The ECDH operations work differently to encrypt and decrypt the contract private key as will be shown. The ECDH operation is used in the case of EVs associated with Vendor B. The specific Diffie Hellman scheme used for our case is One-Pass Diffie Hellman as per the ISO 15118 standard. The encryption part is carried out once again at the Authorization Server end while the decryption part is carried out at the EV end. The following code snippet shows how ECDH was used to encrypt the contract private key:

```
ephSecret = EC_DHE_deriveSecretKey(ephKey, pubKey, strlen(pubKey),
        &ephSecretLength);
sha256v2(ephSecret, buf, ephSecretLength);
//Encrypt
char command[300];
sprintf(command,
        "openssl aes-256-cbc -nosalt -a -e -in prConECC.pem -out private_key.txt -K %s -iv %s",
        buf, argv[1]);

system(command);
```

Figure 4-12: Code Snippet 7

The steps are as follows:

1) Once the OEM certificate is received at the Authorization Server, the public key is extracted from the OEM certificate and stored in the “pubkey” variable in C. An ephemeral ECC based private-public key pair is created and the ephemeral private key is stored in the “ephKey” variable. The ephemeral private key “ephKey” and the OEM public key “pubKey” is used to create a shared secret key which is stored in “ephSecret” variable.

2) A 256-bit (32 byte) hash is created from the shared secret key in “ephSecret” variable using sha256 and stored in variable “buf”.

3) The 256-bit hash in “buf” and a randomly generated 128-bit IV is then used to perform AES symmetric encryption to encrypt the contract private key “prConECC.pem” and the encrypted private key is stored in the “private_key.txt” file. The contract certificate, ephemeral public key, encrypted contract private key and the 128-bit IV is then appended onto a file named “contractInfoECC.txt” using a “$” as delimiter to separate each content and to make the extraction process easier on EV side. The file is then transferred over to EV side. When the EV receives the “contractInfoECC.txt” file, it extracts

```
//Derive the session key
sessionKey = EC_DHE_deriveSecretKey(pkey, trimmedephPubkey,
        strlen(trimmedephPubkey), &secretLength);
sha256v2((char *) sessionKey, (char *) buf2, secretLength);
char command[300];
sprintf(command,
        "openssl aes-256-cbc -nosalt -a -d -in cipher.txt -out prConECC.pem -K %s -iv %s",
        buf2, ivEVECC);

system(command);
```

Figure 4-13: Code Snippet 8

the four contents specified above based on the "$" delimiter. The following code snippet shows how the decryption operation on EV side happens:

The steps are as follows:

1) The OEM private key on EV side stored in "pkey" variable and ephemeral public key extracted from "contractInfoECC.pem" file and stored in "trimmedPubkey" variable are used to derive the same shared secret key that was used for encryption purposes on the Authorization Server end.

2) The secret shared key is stored in "sessionKey" variable. The shared key in "sessionkey" variable is once again hashed using sha256 to produce a 256-bit long hash and stored in "buf2" variable.

3) The 256-bit long hash, which should be the same hash as used on Authorization Server end, is used to perform symmetric decryption to decrypt the encrypted contract private key in "cipher.txt" file and store the contract private key as "prConECC.pem". An AES-symmetric decryption takes place using the 128-bit IV as was used on Authorization Server end.

4.2.4 Digital Signature Creation and Verification

This section deals with the last section for Chapter 4 and the last important cryptographic operations involved in our system. First, we will look at the code implementations for the "signing and verification" process used in our system with regards to random challenge, and then end with the code implementations for verifying certificate chains to trust entities.

The following commands show how the random challenge in "GenChallenge.txt" is signed on EV end:

1) “system(“rm /home/sign.sign; openssl dgst –sha256 –sign prConRSA.pem –out sign.sign GenChallenge.txt”);”

or

2) “system(“rm /home/sign.sign; openssl dgst –sha256 –sign prConECC.pem –out sign.sign GenChallenge.txt”);”

The first OpenSSL command line operation is for Vendor A related to RSA-based signing. RSA with sha256 is used to hash and then sign the random challenge stored in “GenChallenge.txt” using the contract private key “prConRSA.pem” and then stored in a file named “sign.sign”.

The second OpenSSL command line operation is for Vendor B based on ECC algorithm. ECDSA with sha256 is used as the digital signature algorithm as per the ISO 15118 standard. The random challenge in “GenChallenge.txt” is hashed, signed using contract private key “prConECC.pem” and then stored in the “sign.sign” file.

The following code snippet shows how the above signatures are verified on the Authorization Server end:

```
int value =system("openssl dgst -sha256 -verify /var/www/auth/ca/retrievedPubKey.pem "
        "-signature /var/www/auth/ca/sign.sign /var/www/auth/ca/GenChallenge.txt");
```

Figure 4-14: Code Snippet 9

The code used above is common to both scenarios, Vendor A and B. The OpenSSL command line is used to verify the signature in “sign.sign” using the contract public key “retrievedPubKey.pem” and then match the result with the content in “GenChallenge.txt”. The integer variable “value” returns 0 on a successful verification and 256 on verification failure. The final code snippet is used to show how the Authorization Server verifies the digital signature on the OEM certificate to trust it. The code snippet is taken from the Vendor A side:

```
system ("openssl verify -CAfile SubCACertRSA.pem OEMCertRSA.pem");
```

Figure 4-15: Code Snippet 10

The following OpenSSL command line operation verifies the OEM certificate chain and checks whether the Sub CA has signed it or not. The Sub CA is a trusted anchor and is responsible for the issue of OEM certificates.

Chapter 5 - Performance Evaluation of the Multi-Vendor Model

5.1 System Authentication Delay Timings and Analysis

5.1.1 Detailed system timings with RSA 1024 bits and ECC 224 bits

Table 5-1: RSA Timings (1024 bits) [for Vendor A]

RSA Timings (1024 bits)	
	Execution Time (s)
Time to download OEM Certificate	2.49
Server time to process OEM Certificate after post	0.55
Time to extract contract data	0.34
Time to upload contract data	6.81
Time to decrypt on EV side	0.79
Time to download Contract Certificate	2.48
Time to create GenChallenge	0.34
Time to sign GenChallenge	0.63
Time to upload GenChallenge	0.57
Time to download GenChallenge	0.64
Time to verify GenChallenge	0.37
Time to upload EVSE Certificate	2.55
Time to verify EVSE Certificate	0.89
Server time to process Contract Certificate	0.06
Total time to install Contract Certificate	11.19
Total time to validate Contract Certificate	7.21
Total system time	21.98

The system implementation and its details have been covered in Chapter 4 of the paper. The table above represents the timing information collected from our system for Vendor A. The timings above cover the first use case: Use Case 1 where the EV is authenticated by first undergoing the Contract Certificate Installation phase and then using the new Contract Certificate in order to authenticate itself

and allow charging to occur. All the timings are recorded from the time the EV is plugged in to the EVSE until the authentication has finished and the EV has been notified.

The Contract Certificate Installation phase took a total of 11.19 seconds in real time to be executed completely without errors. The Contract Certificate Validation phase took 7.21 seconds to be carried out. The latter phase is to take place by itself every other time the EV plugs in to EVSE from then on. Thus, we can say that if an EV associated with Vendor A (RSA) plugs in next time, it will take an average time of 7.21 seconds to get authenticated before notifying the user that he/she can start charging. However, if an EV associated with the same vendor were to plug in for the first time and undergo both certificate installation as well as validation, the average total system time taken will be 21.98 seconds.

Table 5-2: ECC Timings (224 bits) [for Vendor B]

ECC Timings (224 bits)	
	Execution Time (s)
Time to download OEM Certificate	2.02
Server time to process OEM Certificate after post	0.18
Time to extract contract data	0.34
Time to upload contract data	4.49
Time to decrypt on EV side	0.79
Time to download Contract Certificate	2.02
Time to create GenChallenge	0.37
Time to sign GenChallenge	0.63
Time to upload GenChallenge	0.57
Time to download GenChallenge	0.6
Time to verify GenChallenge	0.4
Time to upload EVSE Certificate	2.15
Time to verify EVSE Certificate	0.89
Server time to process Contract Certificate	0.06
Total time to install Contract Certificate	8.02
Total time to validate Contract Certificate	6.8
Total system time	17.94

Table 5-2 represents the system timings taken for each step for the third use case: Use Case 3. In this use case, an EV associated with Vendor B (ECC) plugs in to EVSE to charge for the first time. Thus, both Contract Certificate Installation and Contract Certificate Validation have to take place. The average time taken to install the certificate is 8.02s and average time taken to validate and authenticate certificate is 6.8 seconds. Thus, an average total system time of 17.94 seconds is taken to fully install and validate an EV associated with Vendor B. However, for every other time, it will only take around 6.8 seconds to authenticate the EV prior to charging.

Comparing the results from both systems (Vendor A and B), it can be seen that the EV associated with Vendor B manages to get the authentication process carried out quicker every time since they use ECC cryptosystem for cryptographic purposes. The certificates are installed 3 seconds quicker and certificates are validated a second quicker every time.

5.1.2 Comparison of Total Authentication Delay with RSA and ECC using different keysizes

The previous section provided detailed timings for each step in the installation and validation process of an EV for both RSA and ECC. In this section, we analyze the difference in the total authentication delay upon changing the keysizes for both RSA and ECC. Then, the authentication delays for both RSA and ECC are shown in a single graph and compared using different keysizes but the same security levels. Table 5-3 shows the authentication delays for different keysizes for RSA and ECC and Figure:

Table 5-3: Comparison of authentication delay for RSA and ECC with different keysizes

KeySizes (bits)	RSA	ECC
128/1024	21.52	17.44
224/2048	21.79	17.48
256/3072	22.43	17.49
384/7680	34.52	17.55

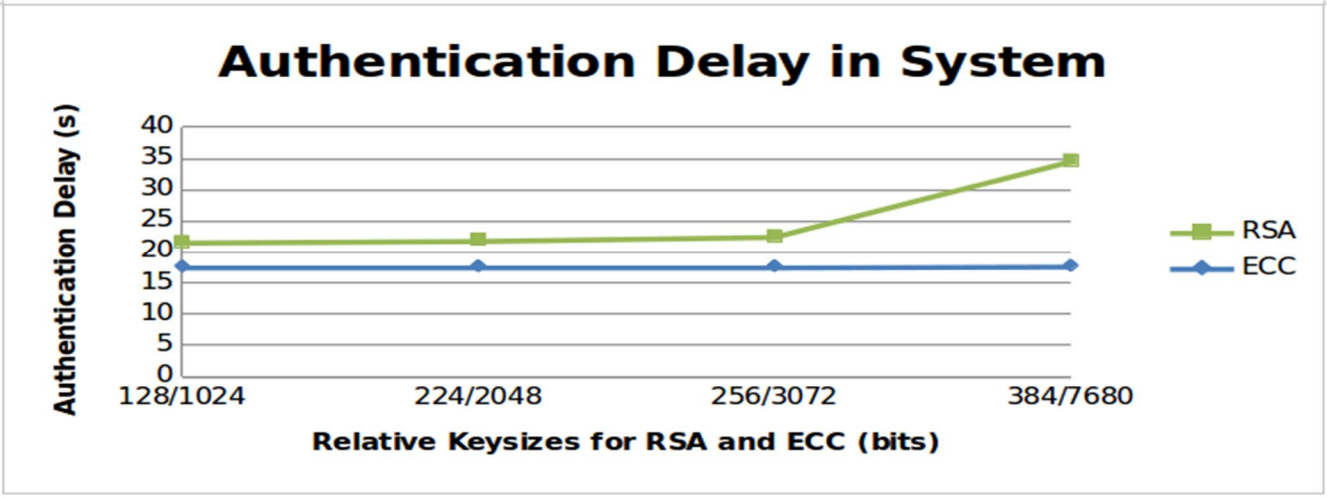


Figure 5-1: Comparison of Authentication Delay for RSA and ECC with different keysizes

The first observation from the graph that can be made is that the authentication delays with respect to ECC operations (for Vendor B) are smaller in all keysizes compared to RSA operations (for Vendor A). The second observation is that the authentication delays are relatively the same with increasing keysizes. The increase in the delay is by only a fraction of a second. The increase in authentication delay with respect to Vendor A (RSA) is also small but only up to 3072 bits. A large increase in authentication delay from 22.43 seconds to 34.52 seconds is observed upon an increase in the keysize to 7680 bits. The increase is by a staggering 12 seconds. There are two main factors behind this huge increase. They are:

1) RSA decryption time: The RSA decryption timings remained relatively small up to keysize of 3072 bits. The average decryption time rose from 0.787 seconds to 6.8 seconds (by 7 seconds) upon increasing the keysize to 7680 bits from 3072 bits.

2) RSA signature generation time: The average RSA signature generation time also rises from 0.642 seconds to 6.64 seconds (by 6 seconds) upon increasing keysize to 7680 bits from 3072 bits.

The RSA decryption time and signature generation time are expected to be similar due to the same operations performed. The two results have been shown independently due to different OpenSSL commands for the two operations and slight differences in system time.

5.2 Performance Evaluation and Comparison of RSA and ECC with State of The Art Articles

5.2.1 Experiments and analysis with RSA and ECC operations within our system for Intel and ARM architectures

Section 5.1 highlighted the authentication delay and evaluation of the system time with different keysizes for RSA and ECC. After implementing the multi-vendor system and taking timings for every important step for the whole authentication cycle with each vendor, it was necessary to evaluate the performance level of each cryptographic operations involved within the authentication process. The main cryptographic operation involved within the process were private-public key generations, encryption, decryption, signature generation and signature verification.

Separate test modules were built in C platform in Eclipse C environment. Test functions were built in order to collect the execution time taken for each of the above mentioned cryptographic operations. The code samples used for the operation behaves similar to the one used in our multi-vendor model. Test modules were run for both RSA and ECC cryptosystems by varying their keysizes or using different file sizes for evaluating encryption/decryption execution times. The tests were carried out in both the Intel and ARM architectures, since the Authorization Server (AS) runs on an Intel architecture and the EV on an ARM architecture. Below are the figures showing the execution times for private and public keypair generation, encryption, decryption, signature generation, signature verification and processing time (encryption + decryption) using RSA for both AS and EV with varying keysizes:

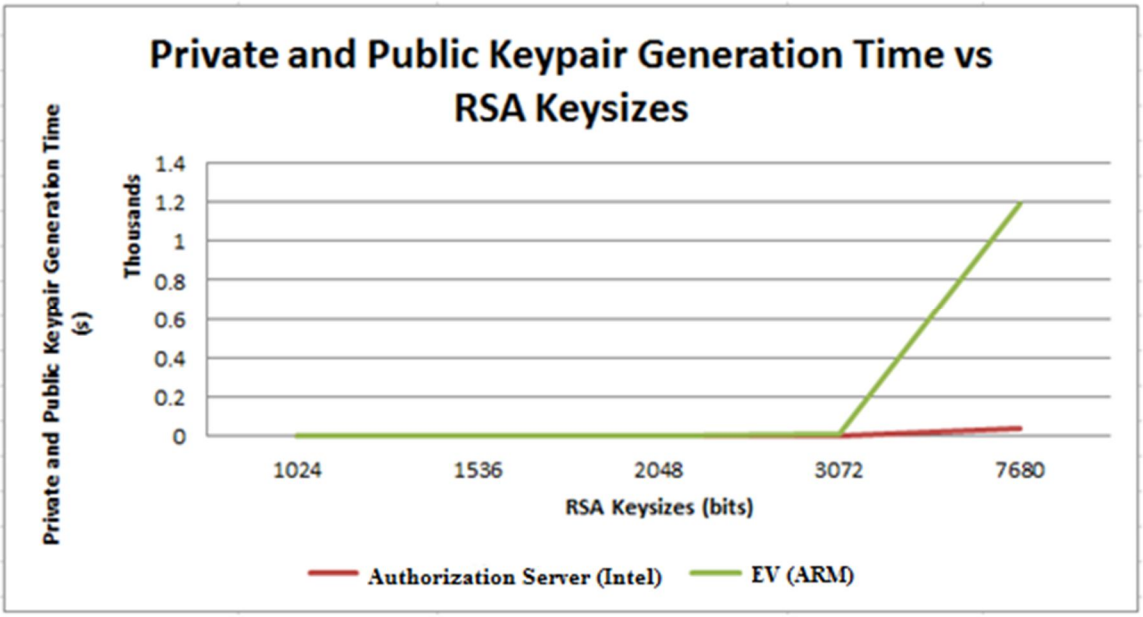


Figure 5-2: Private and Public Keypair Generation Time vs RSA Keysizes (AS and EV)

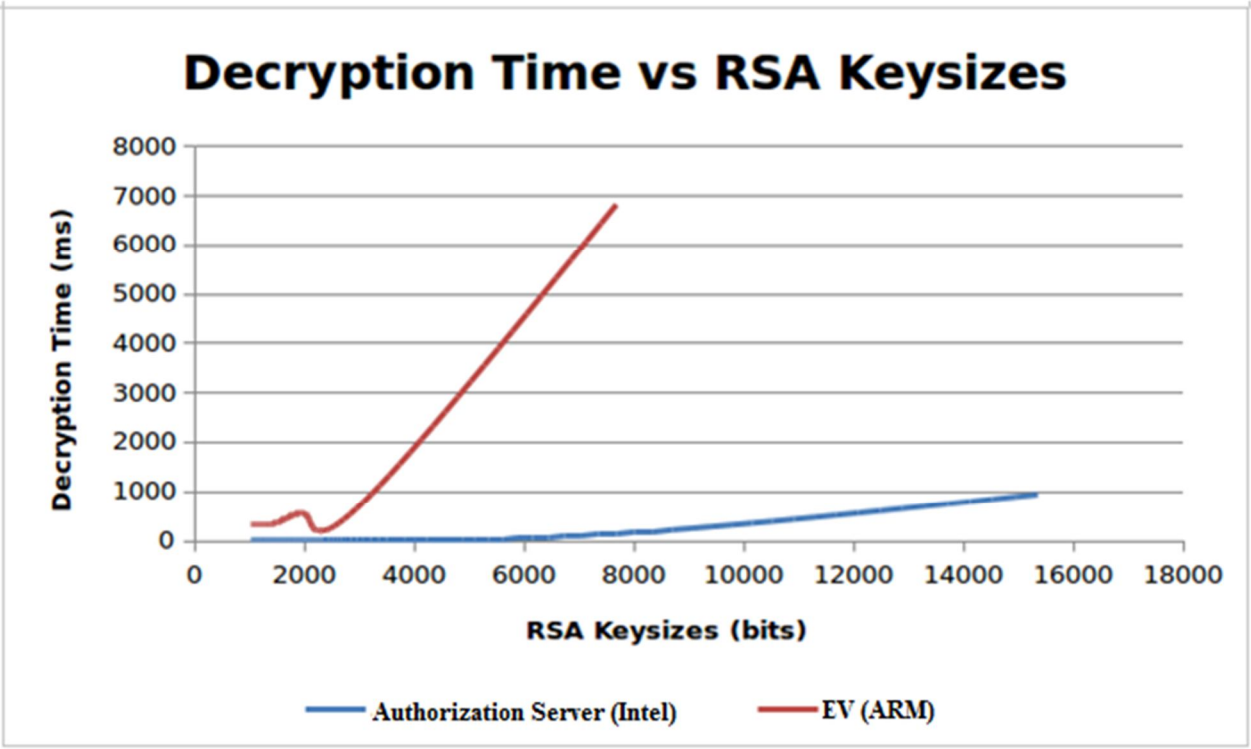


Figure 5-3: Decryption Time vs RSA Keysizes (AS and EV)

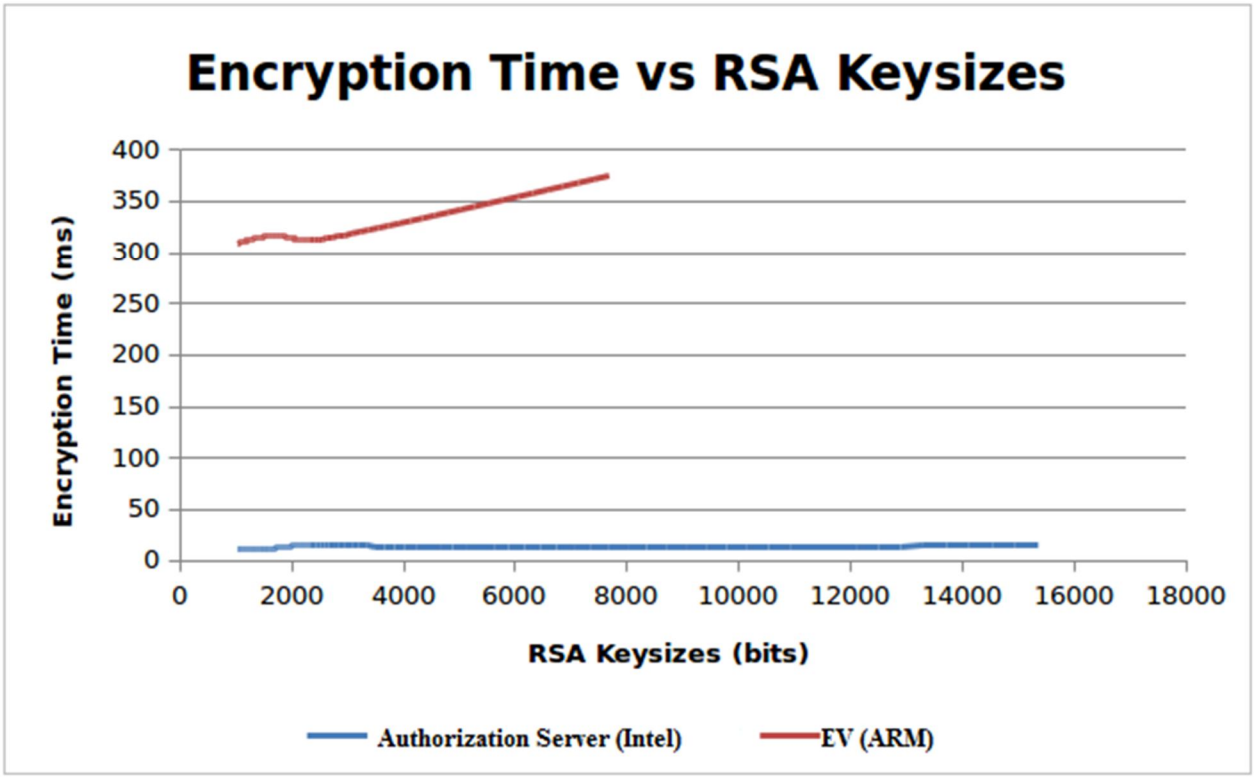


Figure 5-4: Encryption Time vs RSA Keysizes (AS and EV)

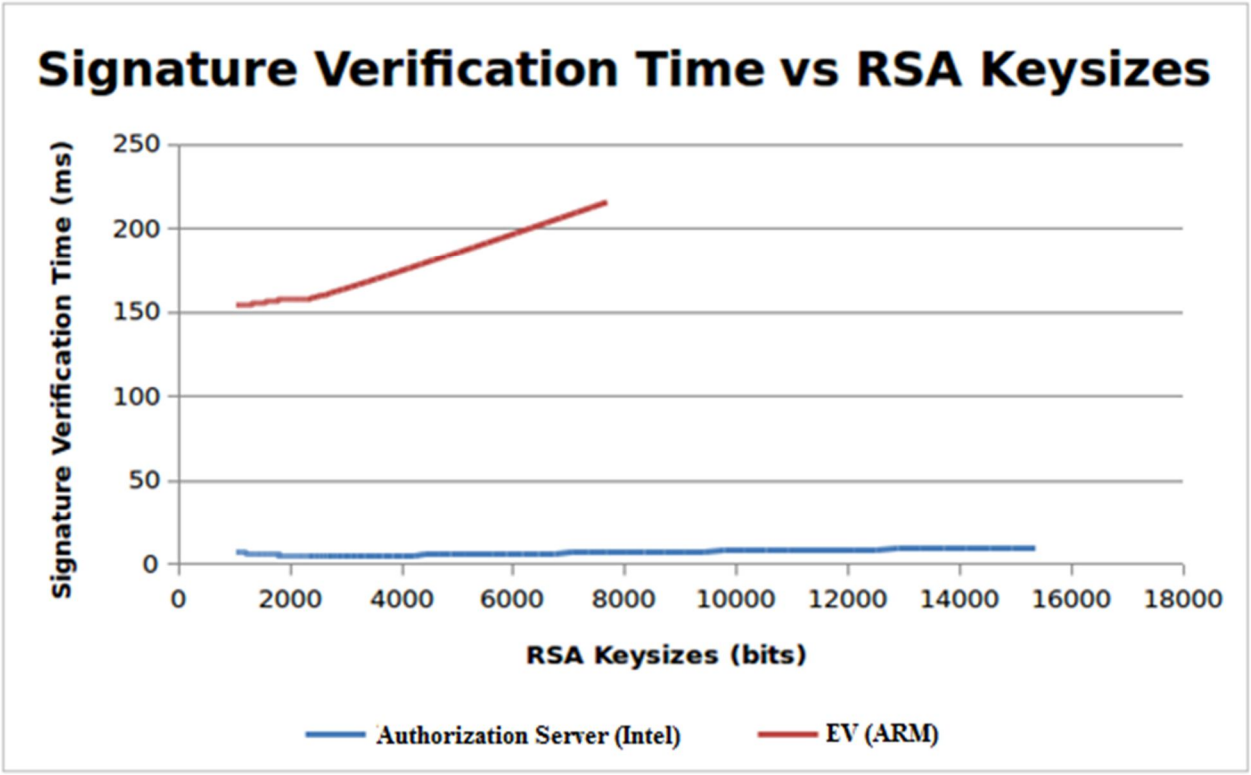


Figure 5-6: Signature Verification Time vs RSA Keysizes (AS and EV)

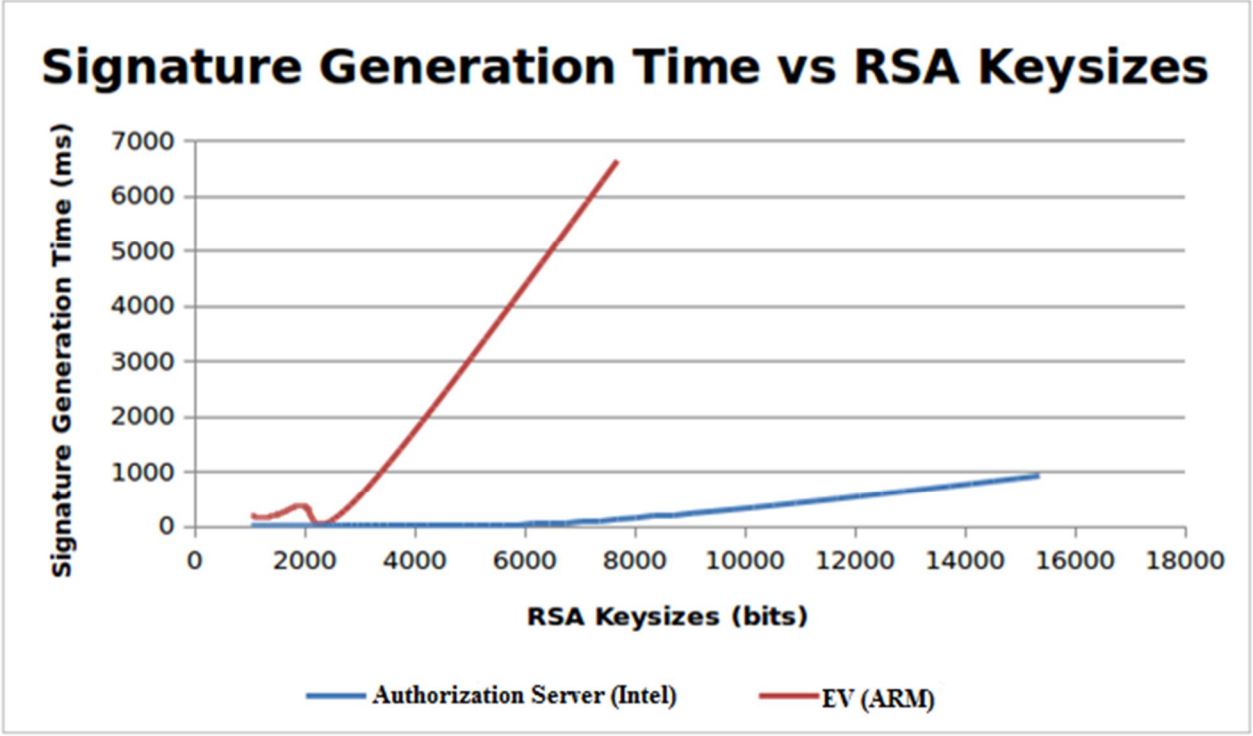


Figure 5-5: Signature Generation Time vs RSA Keysizes (AS and EV)

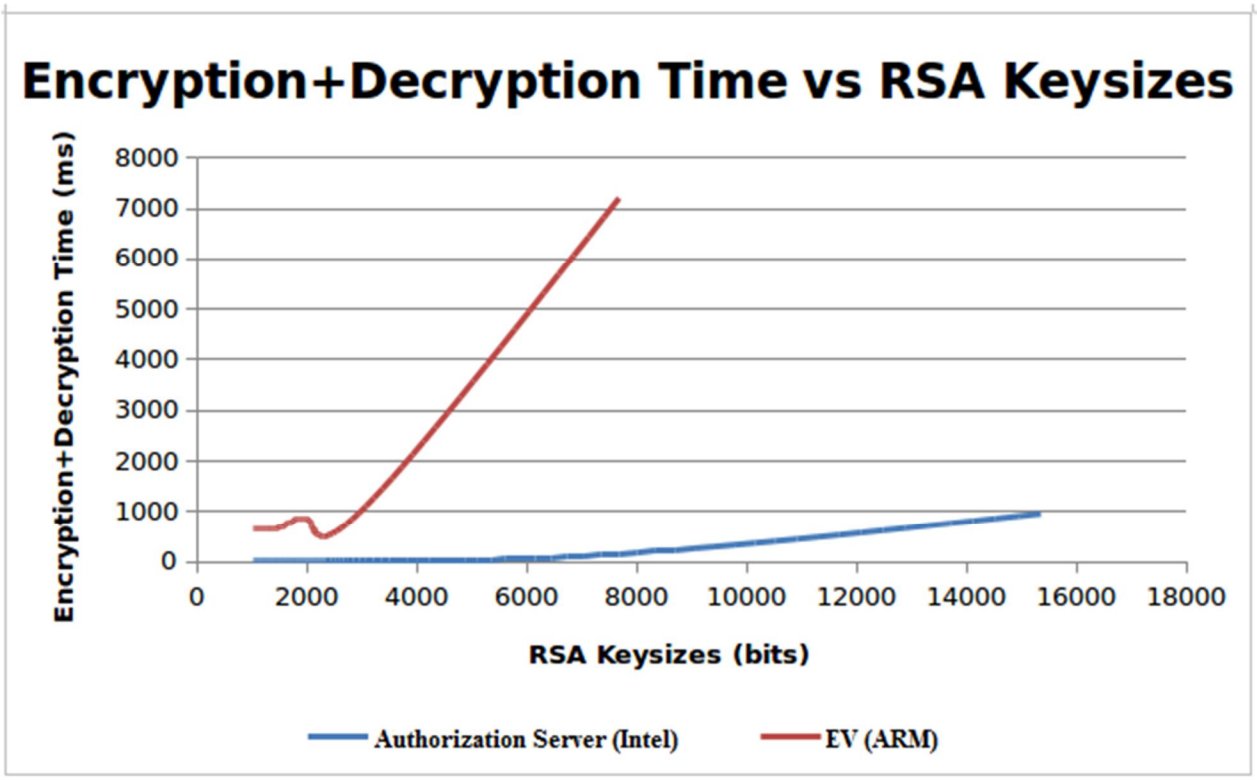


Figure 5-7: Encryption+Decryption Time vs RSA Keysizes (AS and EV)

In all of the 6 figures above, the RSA operations performed in the AS (Intel) are executed a lot faster compared to the same operations done on EV (ARM). Also, the increase in execution time for all operations is relatively large with increasing keysizes with regards to EV (ARM).

The operations performed in AS (Intel) experience a slight increase or none at all (in the case of public key generation) with increasing keysizes. But it can be concluded that the cryptographic operations are performed with greater efficiency on the AS side maintaining the security level compared to the EV side.

The next 6 figures show the execution times for the private and public keypair generation, encryption, decryption, signature generation, signature verification and processing time (encryption + decryption) using ECC cryptosystem with varying keysizes for both the Intel and ARM architecture:

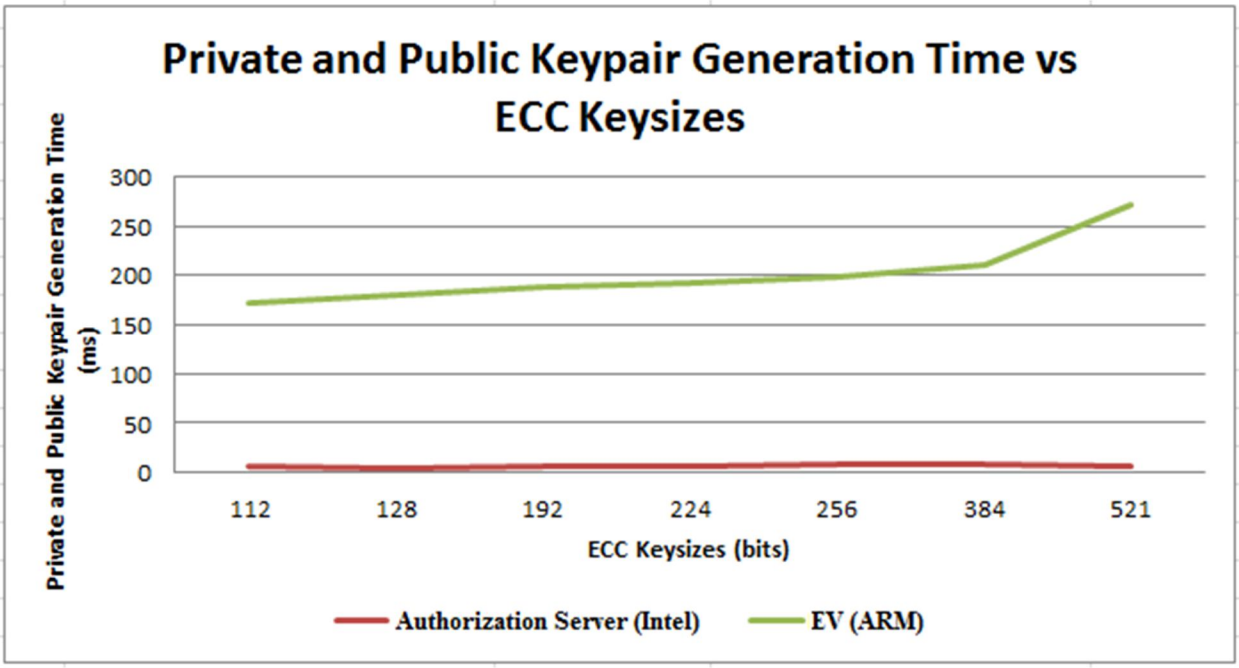


Figure 5-8: Private and Public Keypair Generation Time vs ECC Keysizes (AS and EV)

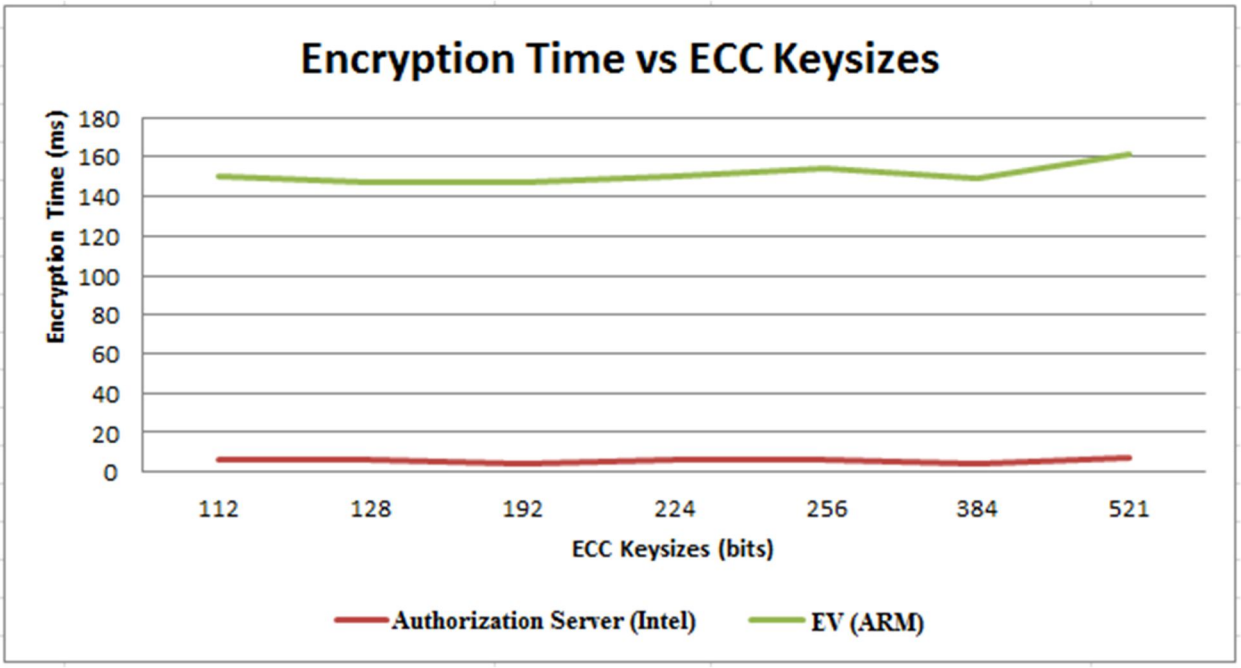


Figure 5-9: Encryption Time vs ECC Keysizes (AS and EV)

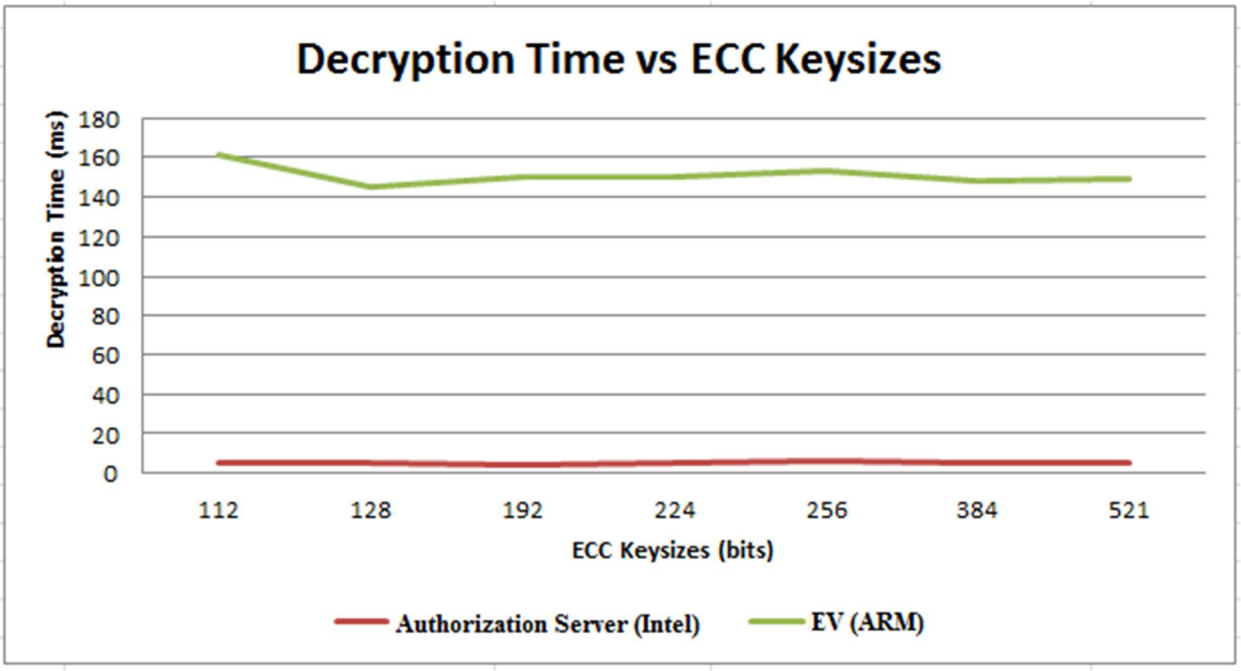


Figure 5-10:Decryption Time vs ECC Keysizes (AS and EV)

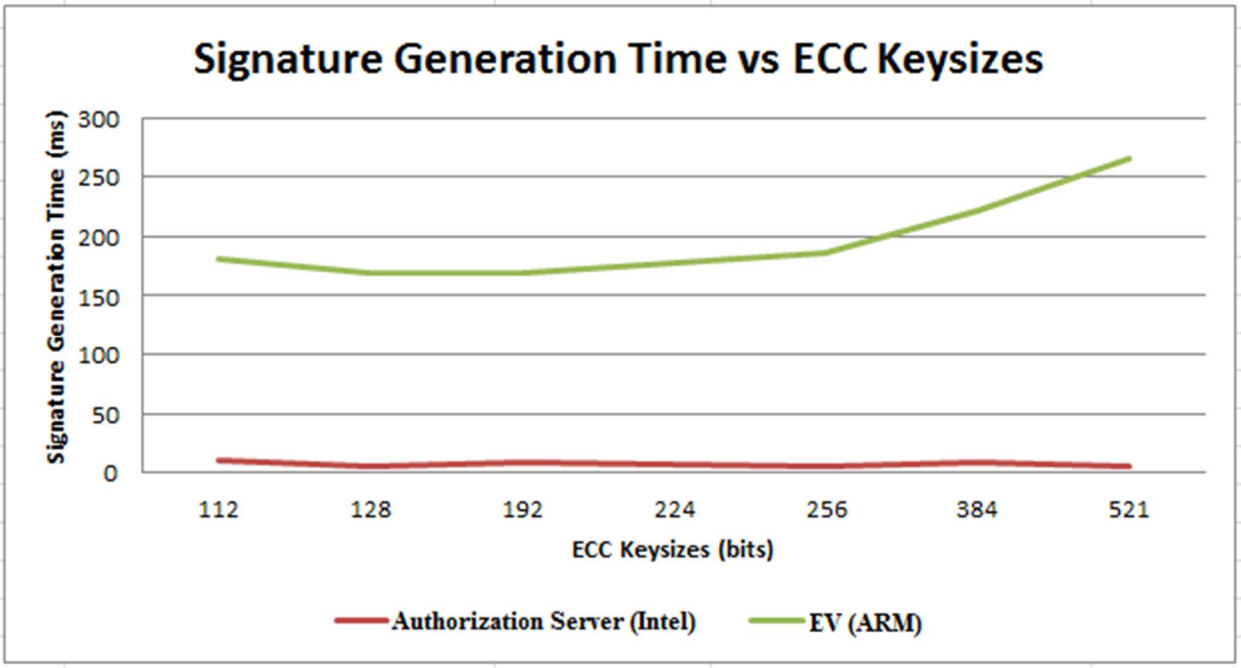


Figure 5-11: Signature Generation Time vs ECC Keysizes (AS and EV)

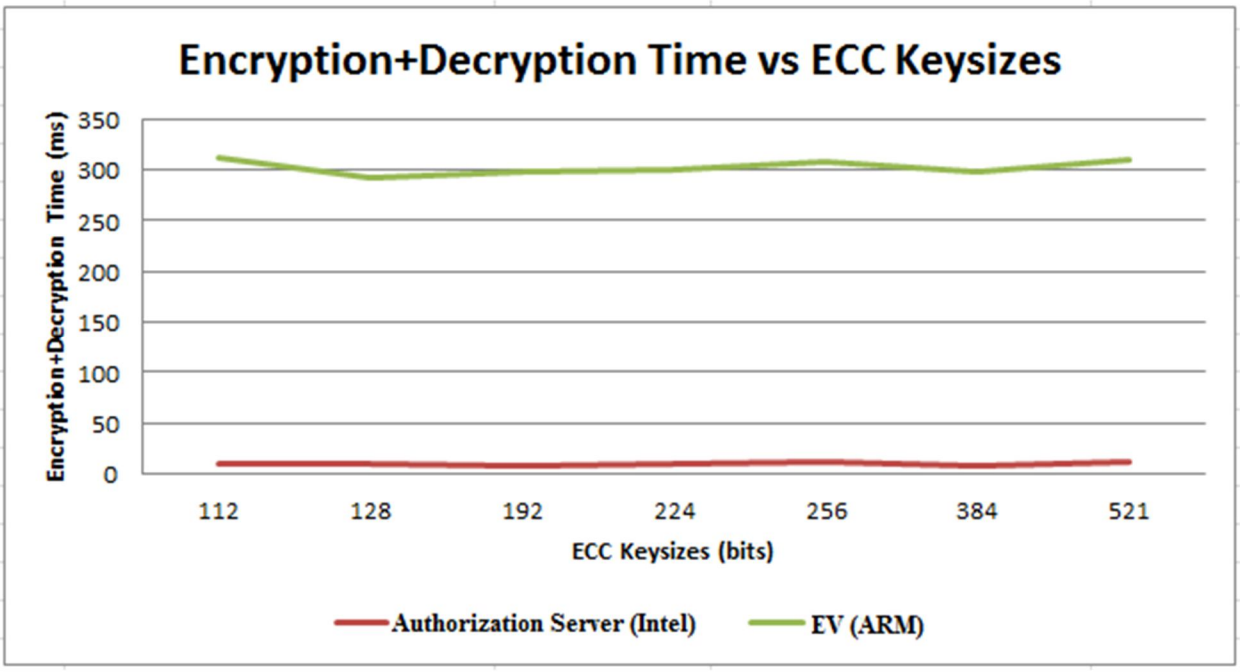


Figure 5-12: Encryption+Decryption Time vs ECC Keysizes (AS and EV)

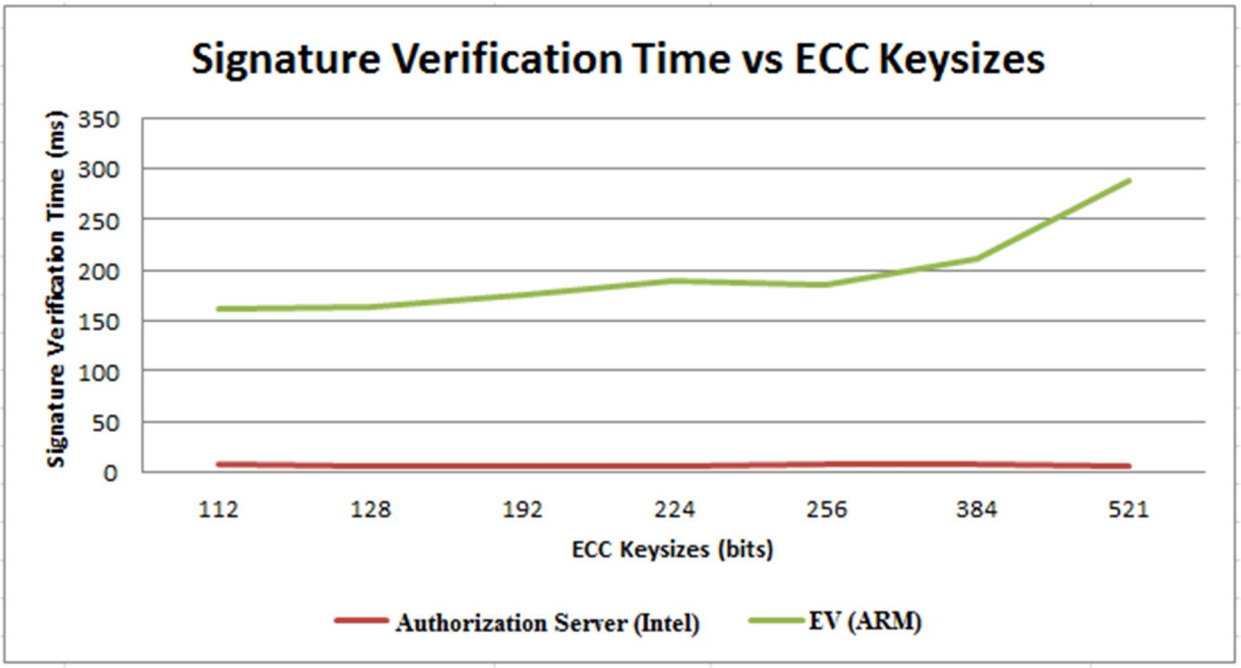


Figure 5-13: Signature Verification Time vs ECC Keysizes (AS and EV)

The execution times for all ECC operations are much quicker for both the architectures (Intel and ARM) compared to RSA. The other big difference with ECC operations in the ARM architecture with regards to RSA operations is that the execution times increase relatively slowly or not at all. The gradient is smaller compared to that in RSA with ARM architecture.

5.2.2 Comparison between execution times for RSA and ECC in Intel and ARM architectures

The next figures show the comparison between the execution times for each of the cryptographic operations for RSA and ECC with increasing keysizes, making sure the keysizes provide the same security level for both the cryptosystem.

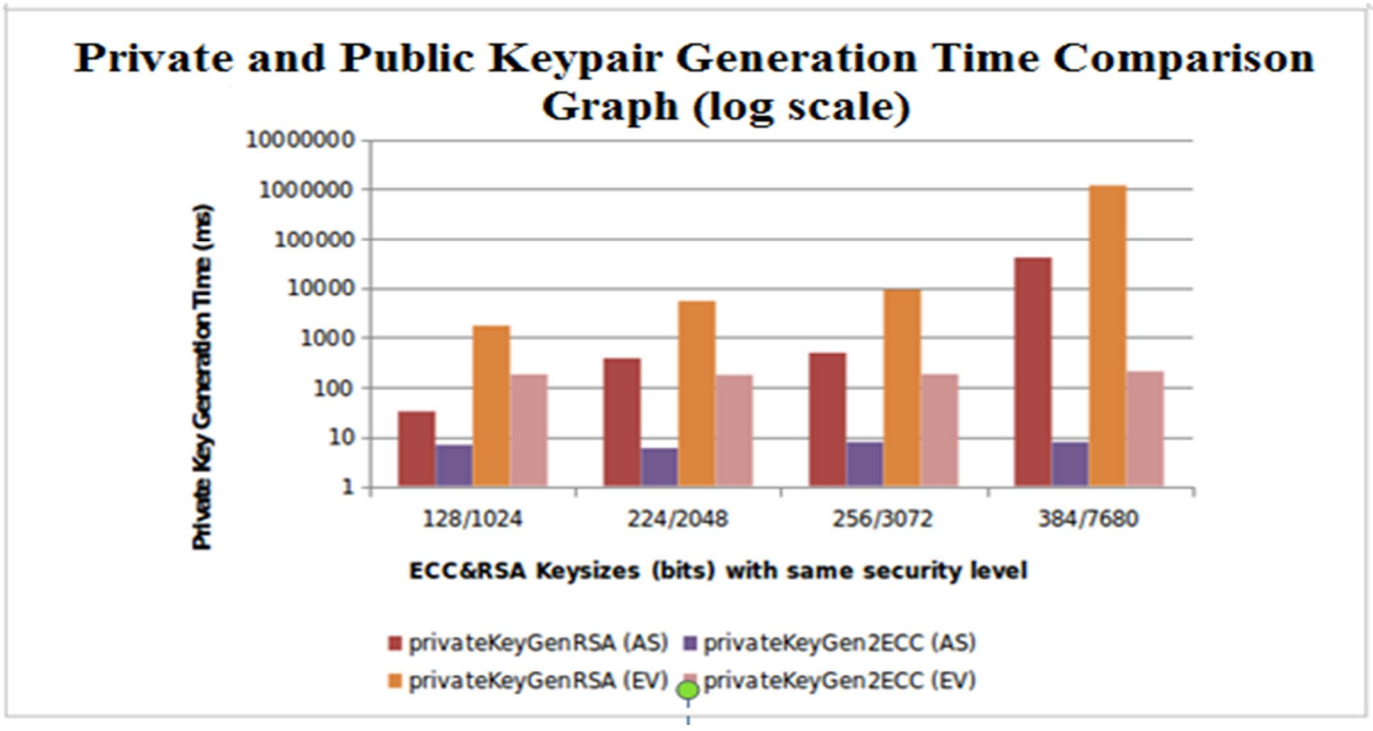


Figure 5-14: Private and Public Keypair Generation Time Comparison Graph (AS and EV)

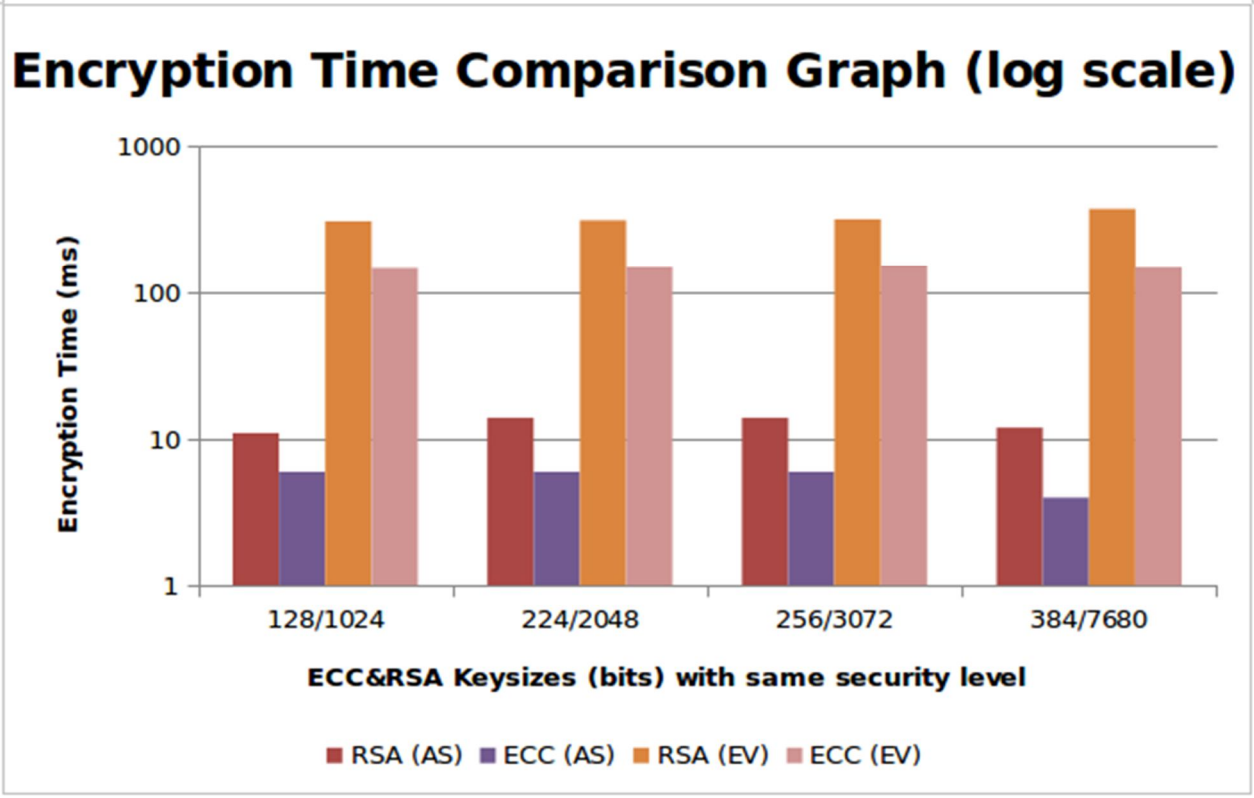


Figure 5-15: Encryption Time Comparison Graph (AS and EV)

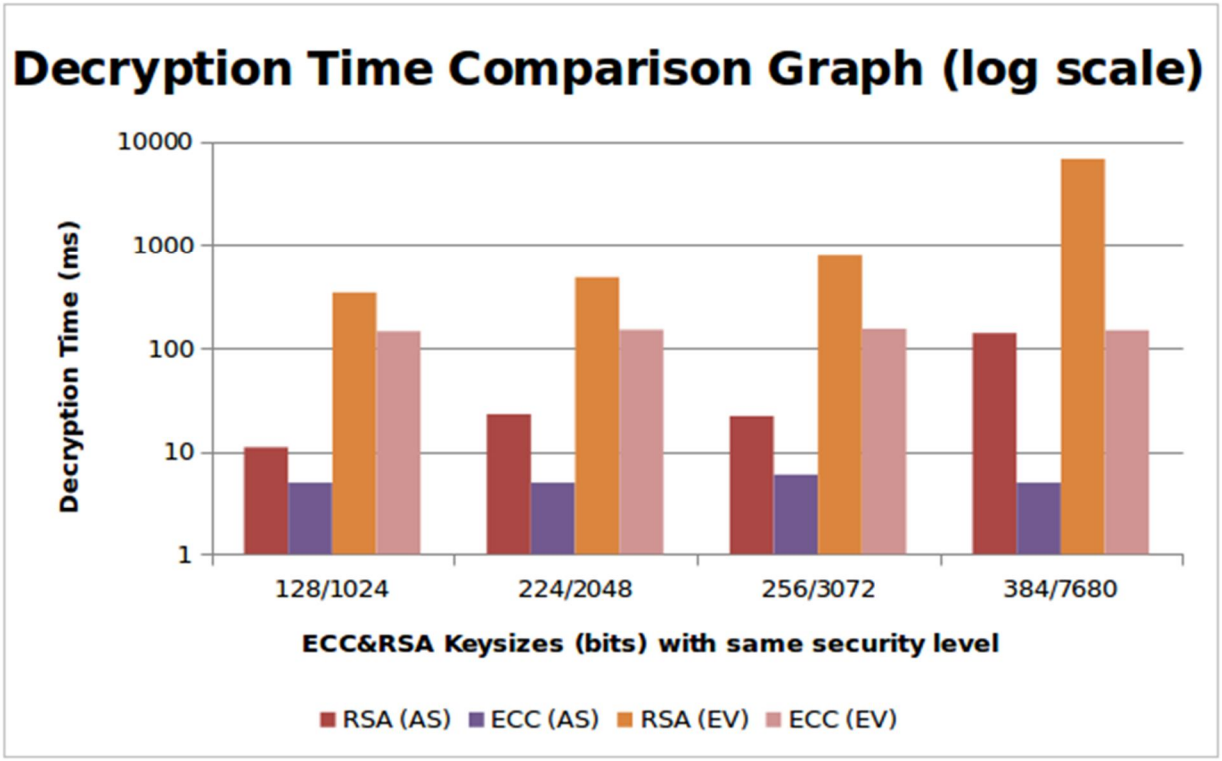


Figure 5-16: Decryption TIme Comparison Graph (AS and EV)

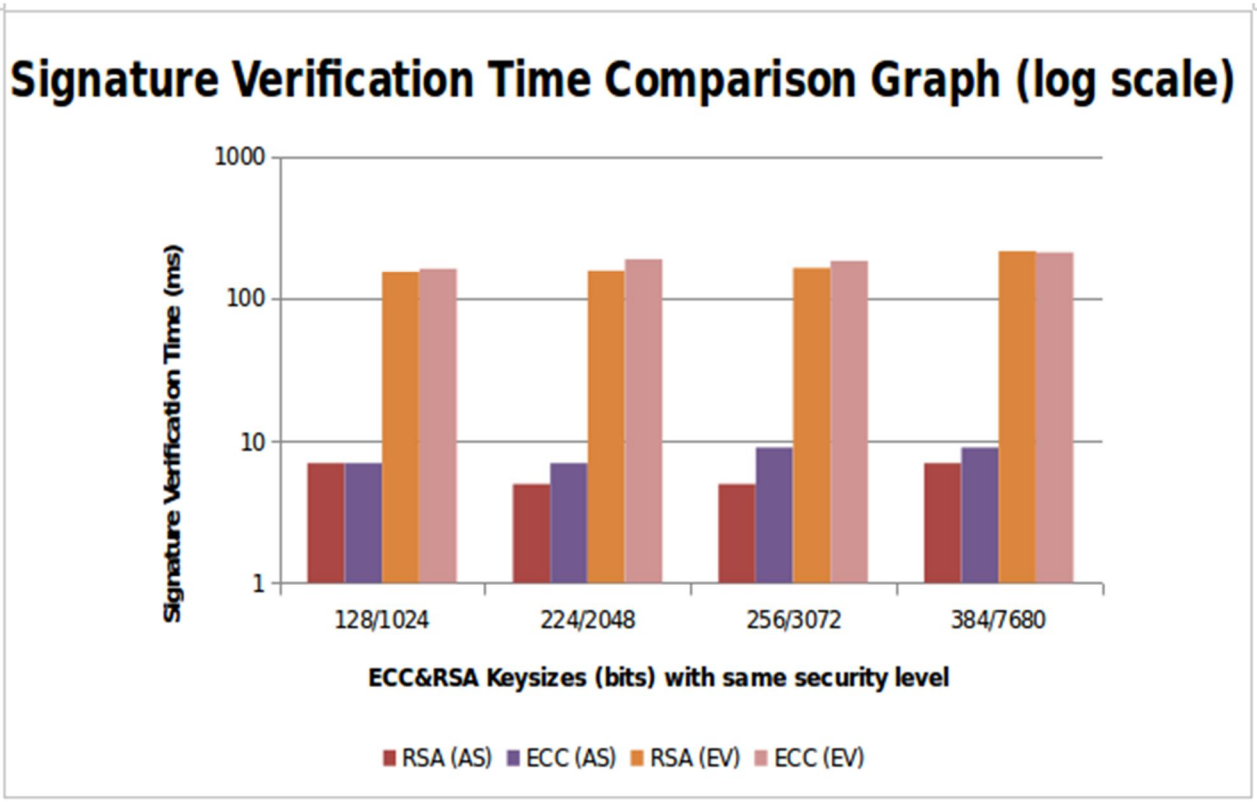


Figure 5-17: Signature Verification Time Comparison Graph (AS and EV)

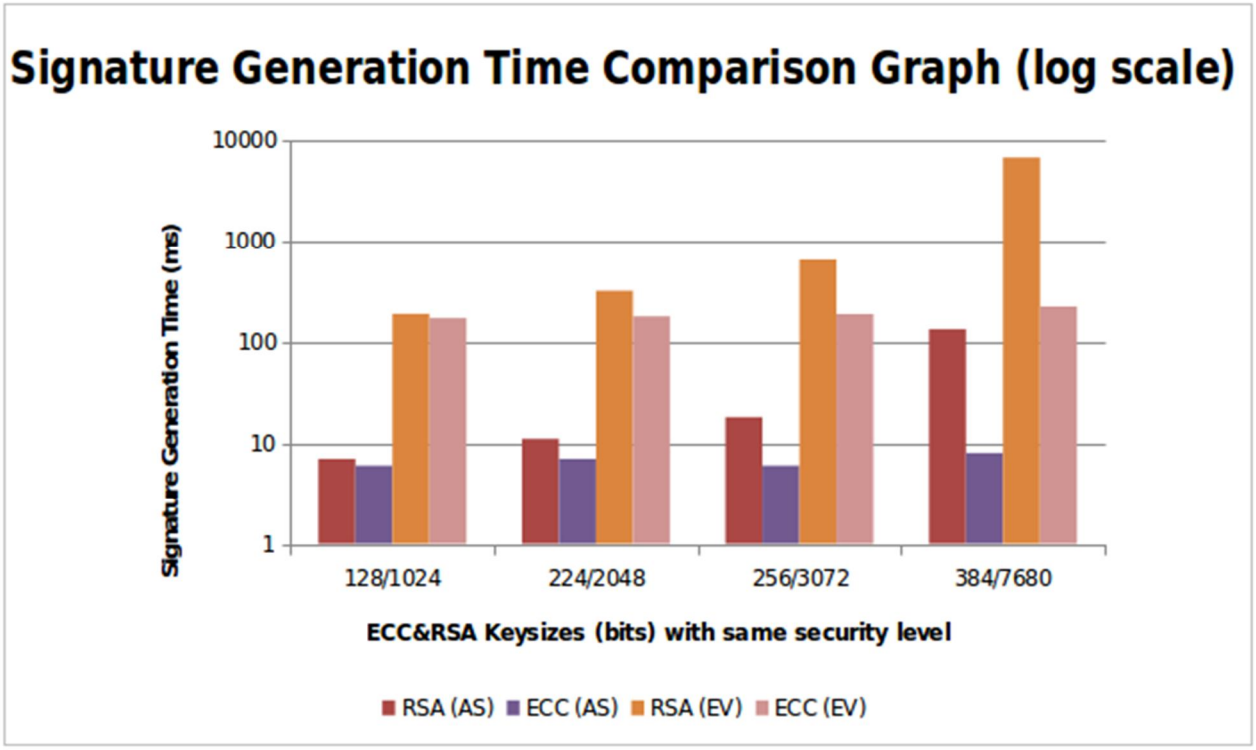


Figure 5-18: Signature Generation Time Comparison Graph (AS and EV)

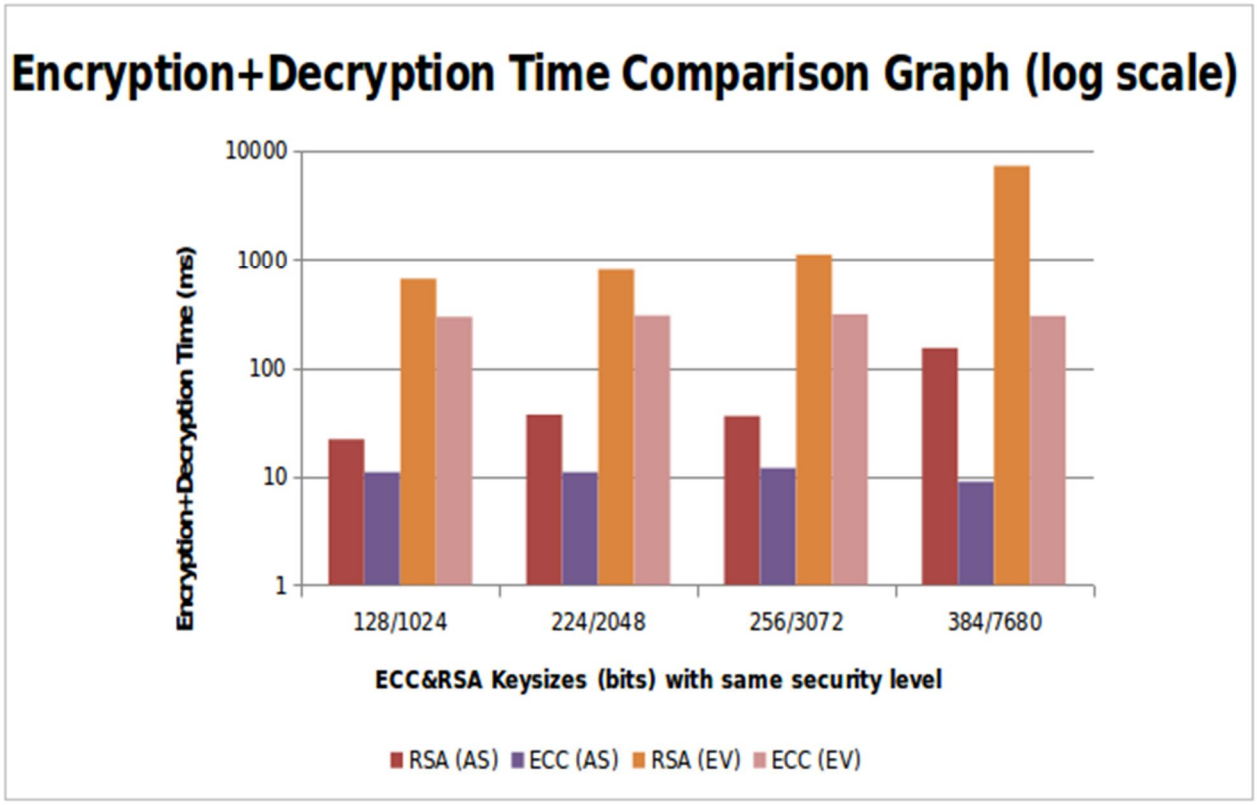


Figure 5-19: Encryption+Decryption Time Comparison Graph (AS and EV)

The time taken to generate RSA keypairs is longer in the ARM processor than in the Intel processor. The ECC operations are executed with less time compared to the operations with RSA on both architectures. It is interesting to note how a private key generated for RSA with a 1024-bit key on the Intel architecture is much quicker than an ECC key with 128 bits (same security level) on ARM processor. However, with increasing keysizes, the operations on the RSA side for that same scenario become more time consuming as opposed to the ECC operations.

The encryption and decryption times for RSA and ECC on the Intel board are comparable. However, the execution time increases significantly for RSA operations on both Intel and ARM architectures. But comparing the two, the rise is greater for the ARM architecture. The signature generation time using RSA keys of 7680 bits long on ARM board is large. The signature generation time with ECC is stable for increasing keysizes. Signatures are verified at around the same time for both RSA and ECC on the ARM processor, even with increasing keysizes. The time taken is around 200ms. For ECC and RSA keys with 128 bits and 1024 bits long respectively, the time taken to verify a signature on the Intel board is similar. However, with increasing keysize and the same security level, RSA comes off slightly

quicker compared to ECC on Intel board. For the last analysis, the processing times (encryption + decryption) for operations on Intel and ARM processors for both RSA and ECC are quite significant.

5.2.3 Comparison of RSA and ECC performance with State of The Art designs

The following section compares our RSA and ECC operation performance with state of the art designs. Several recent articles have been discussed with results and graphs in Chapter 2 of the thesis. This section will present and compare the results from those articles discussed with our results.

Article [QUI13]:

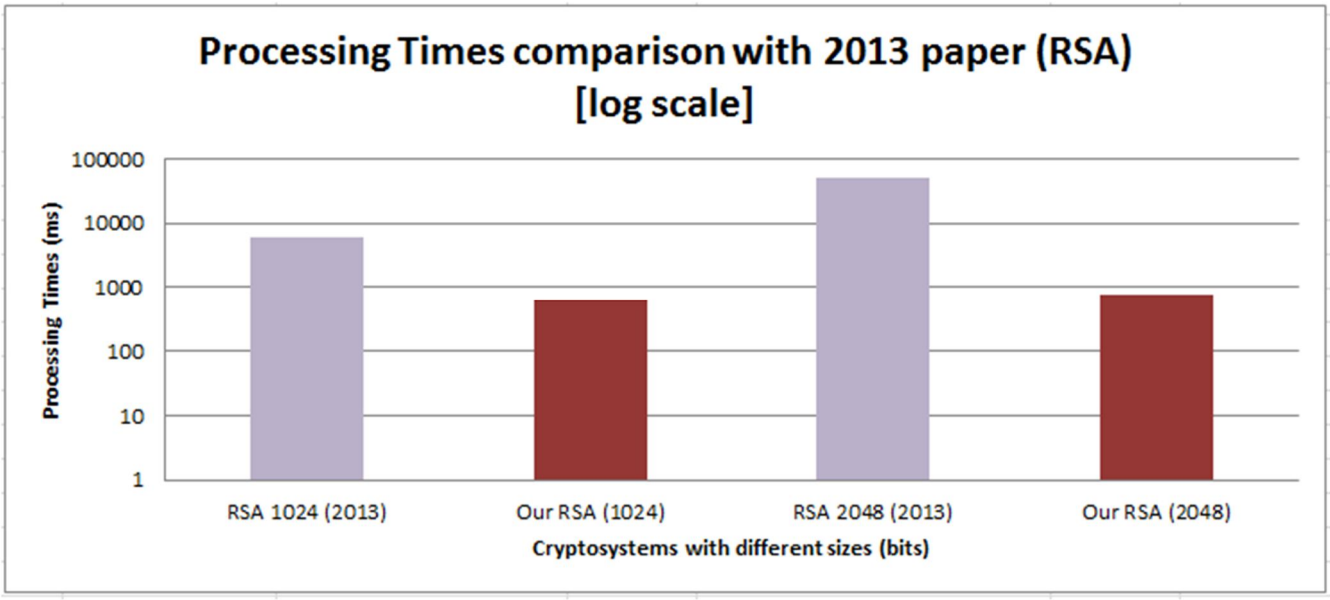


Figure 5-20: Processing Time Comparison with 2013 paper

Figure 5-20 above compares the processing time (encryption + decryption) with regards to RSA operations as stated in the article cited [QUI13] with our results. In the first two cases, with RSA keys of 1024 and 2048 bits used, the processing times with regards to our operations are faster.

The difference in the results can be attributed to the following reasons:

1) In both cases, the RSA encryption and decryption operations were conducted on an ARM architecture. As stated in the article, the simulator ran on StrongArm SA-110 platform, a 32-bit RISC processor. Our experiments were conducted on the EVACharge SE Board with Freescale i.MX287 microcontroller. Thus, despite similar architectures used, the system configurations varied. The difference in the processing speed in the architectures has contributed to the execution times derived.

2) Our cryptographic operations were implemented using OpenSSL library. As per the article, MIRACL library was used for cryptographic operations. The difference in the library used and the way the code was implemented also contributed to the difference in results.

Article [ANJ14]:

Figure 5-21 compares the encryption and decryption times for the RSA operations with our results and the results presented in the article cited [ANJ14]:

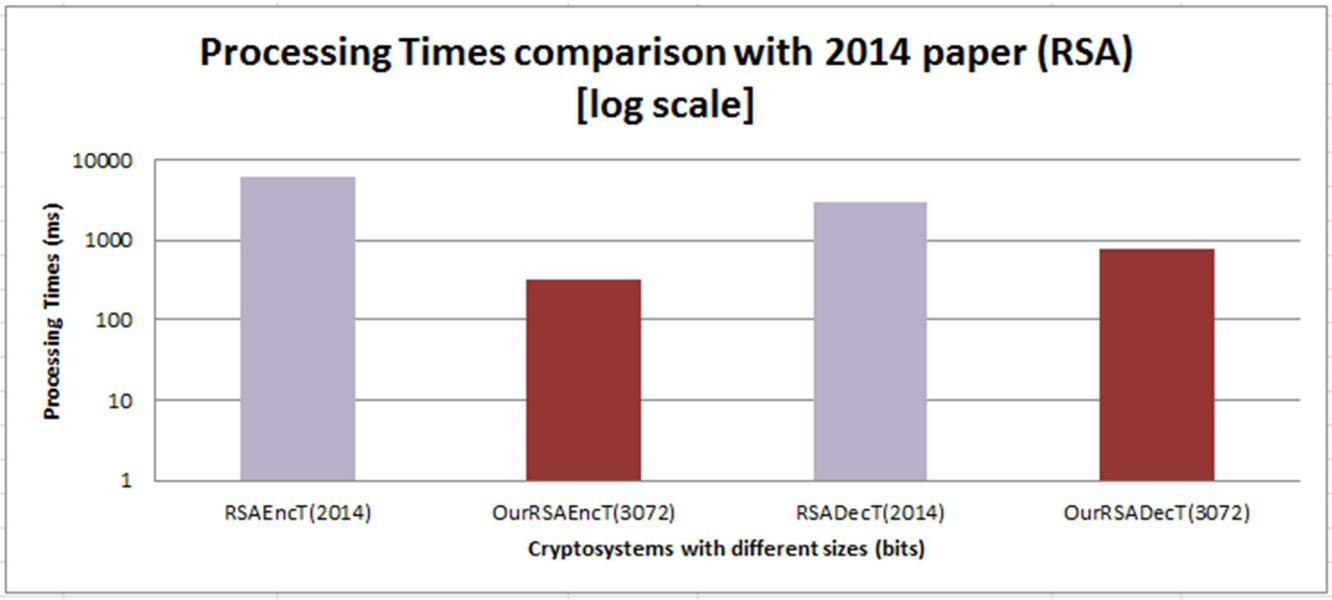


Figure 5-21: Processing Time Comparison with 2014 paper

In both cases (RSA encryption, RSA decryption) in the figure above, our operations were much more quickly carried out once again.

The difference in the results can be attributed to the following reasons:

1) In both cases, the RSA encryption and decryption operations were conducted on an ARM architecture. However, despite similar architectures used, the system configurations were different. The difference in the processing speed in the architectures has contributed to the execution times derived.

2) Our cryptographic operations were implemented using OpenSSL library. The library used for implementing the cryptographic operations have not been specified in the article. There is a high chance of the library used being different from ours. The difference in the library used and the way the code was implemented have played a part in the difference in execution times of the RSA encryption and decryption operations.

Chapter 6 - Conclusion and Future Work

6.1 Concluding Remarks

In this paper, we have proposed a multi-vendor model to allow for the authentication and authorization of Electric Vehicles (EV) in Electric Vehicle Supply Equipments (EVSE). In order to set up the model, we used two EVACharge boards to emulate the actions of the Electric Vehicle and the Electric Vehicle Supply Equipment. We used a high-configuration Intel processor to simulate the actions of the Application Server and Authorization Server, as well as our Database server. Each of the vendors, Vendors A and B have their own Application Server, Authorization Server and Database Server deployed. The RSA cryptosystem was used to serve cryptographic purposes within the authentication process of the system with regards to Vendor A. As for Vendor B, the ECC cryptosystem was used for mainly generating keys and signing purposes combined with the One-Pass Diffie Hellman used as key establishment protocol. The cryptographic operations were based largely on the ISO 15118 standard on the Vendor B end.

After building the multi-vendor model, the code fragments covering the cryptographic operations were plucked out from the system and deployed on independent test modules for further analysis. The cryptographic operations studied were the private and public keypair generation, signature generation, signature verification, encryption and decryption. The execution times for each of these operations were noted with different keysizes and different plaintext sizes. The results were later compared with State of The Art schemes for the purposes of performance evaluation.

Chapter 2 of the thesis discussed the State of The Art schemes and highlighted the results from the papers. In Chapter 3 of our thesis, we defined our System Architecture. The system components within the model were defined and their functions were highlighted. In Chapter 4 concerning the System Implementation, the four main use case scenarios were presented with sequence diagrams and their steps explained in detail. An additional fifth use case was also presented to show how the system is able to recover from failure. The last major chapter, Chapter 5, covered the performance evaluation section of the thesis. The first portion of this chapter highlighted the system timing and the overall authentication delay in our system concerning EVs associated with both vendors, Vendor A and B. The authentication delay was then recorded by varying the keysizes for RSA and ECC and analyzed after

plotting the results in a line graph. Then, the execution times for each of the five stated cryptographic operations were compared within both the Intel and ARM architectures, wherein the Intel architecture was used for simulating the Authorization Server and the ARM architecture for emulating Electric Vehicles. Next, the execution times and performance of the RSA and ECC cryptosystems within our own model were compared using different keysizes but equivalent security levels. The last portion of Chapter 5 was dedicated to comparing our results with results from State of The Art schemes. The results were plotted as column graphs or line graphs for ease of comparison and analysis.

The outcomes from the performance evaluation section were as follows:

1) The system authentication delay involving installation and validation of a Contract Certificate for Vendors A and B were 21.52s and 17.44s respectively. The certificate installation process for Vendors A and B took 11.19s and 8.02s respectively whereas the certificate validation process took 7.2s and 6.8s respectively. The authentication delay was lower with respect to Vendor B in all scenario.

2) The execution times for all the five operations were lower in the Intel processor as compared to the same operations in the ARM architecture within the system. The execution times for all five operations were lower using the ECC cryptosystem as compared to the ones using the RSA cryptosystem.

3) Results were compared with the two State of The Art schemes and shown in graphs. Our execution times were either similar or lower with respect to all the six operations compared with results from State of The Art schemes.

The limitations and assumptions made in this thesis are as follows:

1) Centralized Server: The model is based on a centralized server (associated with Vendor A or B) responsible for authenticating the electric vehicles in the system. Decentralized servers would avoid a single point of failure and reduce authentication delay even more.
2) Small-scale model: The testing scope of the model was limited within the research lab premises. The distance factor would have accounted for an increase in the overall system delay due to an increase in the propagation time.
3) Same boards and processor used to emulate EV and EVSE: The same EVACharge boards have been used to emulate the vehicle and charging station for authenticating with both vendors. This is unlikely

to be the case in the real life scenario, which might affect operation execution times depending on the boards used.

4) There are limitations in security as far as hacking into the systems are concerned. The Authorization Servers (AS) and the EVs are vulnerable to attack from hackers. There is a possibility of confidential data such as the ephemeral private key on the Authorization Server side or the private key of the contract certificate on the EV side being stolen by hacking to gain access to the contents inside the systems.

6.2 Future Work

Future work is as follows:

1) For now, two vendors are connected with the model using two different cryptosystems. The model can be extended to allow for more vendors using other cryptosystems.

2) The model can be tested in a larger scale. More boards can be used in the testing process. The model can be tested by authenticating multiple electric vehicles in different charging stations to test for scalability. The servers can be remotely deployed and the authentication process tested with the increase in distance of the server from the EVACharge boards. This will provide us with a more realistic system delay, taking propagation time and signal attenuation into consideration.

3) Functionalities can be extended within the system to allow for selecting and reserving charging stations from web or mobile applications prior to electric vehicle charging onto a specific charging station. This would make the model more realistic.

4) A user-friendly interface can be created at the Electric Vehicle Supply Equipment end. In this way, the EV owner can be informed of the stage they are at when their vehicle is being authenticated.

References

[ANJ14] K. Anjan et al, “Comparative Study and Performance Analysis of Encryption in RSA, ECC and Goldwasser-Micali Cryptosystems,” International Journal of Application or Innovation in Engineering & Management, Vol 3, Issue 1, January 2014, pp. 111-118.

[ARM] “ARM Compilers,” [Online] Available at: http://elinux.org/ARMCompilers/, Date Accessed: March 1, 2016.

[AUE95] D. Auer and M. Buer, "A design flow for embedding the ARM processor in an ASIC," *In Proceedings* Eighth Annual IEEE International, September 1995, pp. 342-345.

[AUS02] T. Austin, E. Larson and D. Ernst, "SimpleScalar: an Infrastructure for Computer System Modeling," IEEE in Computer, Vol 35, Issue 2, February 2002, pp. 59-67.

[BAD09] M. Badra, T. Guillet and A. Serhrouchni, "Random Values, Nonce and Challenges: Semantic Meaning versus Opaque and Strings of Data," *In Proceedings* IEEE 70th Vehicular Technology Conference Fall (VTC 2009-Fall), September 2009, pp. 1-5.

[BAU11] T. Baumeister "Adapting PKI for the smart grid," *In Proceedings* 2011 IEEE International Conference on Smart Grid Communications (SmartGridComm), October 2011, pp. 249-254.

[BHA11] J. Bhatta and L. P. Pandey, "Performance Evaluation of RSA Variants and Elliptic Curve Cryptography on Handheld Devices," IJCSNS International Journal of Computer Science and Network Security, Vol 11, Issue 11, November 2011.

[BIS04] M. Bishop, “Introduction to Computer Security,” Upper Saddle River, Addison-Wesley Professional, 2004, pp. 784.

[BLA99] I. Blake, G. Seroussi and N. Smart, “Elliptic Curves in Cryptography”, Vol 265, Cambridge University Press, 1999.

[BOU] “The Bouncy Castle Crypto APIs for Java,” [Online] Available at: http:// www.bouncycastle.org/, Date Accessed: March 1, 2016.

[BRO13] K. J. Brown, "Electric vehicle supply equipment; a safety device," *In Proceedings* Transportation Electrification Conference and Expo (ITEC), 2013 IEEE, 16-19 June 2013, pp. 1-5.

[CHA93] C.C. Chan, "An overview of electric vehicle technology," IEEE Proceedings, Vol 81, Issue 9, September 1993, pp. 1202-1213.

[CHA05] T. Chang, W. Yang and M. Hwang, “Simple Authenticated Key Agreement and Protected Password Change Protocol,” An International Journal Computers and Mathematics with Applications, Vol 49, Issues 5-6, April – May 2005, pp. 703-714.

[CHA14] A. C. F. Chan and J. Zhou, "Cyber–Physical Device Authentication for the Smart Grid Electric Vehicle Ecosystem," IEEE Journal on Selected Areas in Communications, Vol 32, Issue 7, July 2014, pp. 1509-1517.

[CHE05] S. Ahmad, A. H. Mir and G. R. Beigh, "Latency evaluation of extensible authentication protocols in WLANs," *In Proceedings* 2011 IEEE 5th International Conference on Advanced Networks and Telecommunication Systems (ANTS), December 2011, pp. 1-5.

[CRYA] "Crypto++," [Online] Available at: http://www.cryptopp.com/, 2015.

[CRYB] “cryptlib,” [Online] Available at: http://www.cryptlib.com/, 2015.

[DIF76] W. Diffie and M. Hellman, “New Directions in Cryptography,” IEEE Transactions on Information Theory, Vol 22, Issue 6, November 1976, pp. 644-654.

[ELE] “Electric Vehicle Communication,” [Online] Available at: http://www.i2se.com/ev-communication/, Date Accessed: March 1, 2016.

[ELH08] M. El-Hadedy, D. Gligoroski and S. J. Knapskog, "High Performance Implementation of a Public Key Block Cipher - MQQ, for FPGA Platforms," *In Proceedings* International Conference on Reconfigurable Computing and FPGA, ReConFig '08, December 2008, pp. 427-432.

[FON11] A. Fongen, "Optimization of a Public Key Infrastructure," *In Proceedings* Military Communications Conference, MILCOM 2011, November 2011. pp. 1440-1447.

[FUR96] S. Furber, “ARM System Architecture,” Addison-Wesley Longman Inc, 1996.

[GAS02] M. Gast, “802.11 Wireless Networks - The Definitive Guide,” O’Reilly, December 2002.

[GNU] “The GNU MPFR Library,” [Online] Available at: http://www.mpfr.org/, Date Accessed: March 24, 2016.

[HAL96] N. Haller et al, “A One-Time Password System,” IETF RFC 2289, May 1996.

[HAN09] K. Hansen, T. Larsen and K. Olsen, “On the Efficiency of Fast RSA Variants in Modern Mobile Phones,” International Journal of Computer Science and Information Security, Vol 6, Issue 3, 2009.

[HOL10] M. Holbl, T. Welzer and B. Brumen, “Two Proposed Identity-Based Three-Party Authenticated Key Agreement Protocols from Pairings,” Computers and Security, ScienceDirect, Vol 29, Issue 2, March 2010, pp. 244-252.

[HPG] “HP G62 Series,” [Online] Available at: http://www.notebookcheck.net/HP-G62-Series.31933.0.html/, Date Accessed: March 1, 2016.

[IEE2012] “IEEE Standard for Information technology--Telecommunications and information exchange between systems Local and metropolitan area networks--Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," IEEE Std

802.11-2012 (Revision of IEEE Std 802.11-2007), 29 March 2012, pp. 1-2793.

[INT] F. Delattre and M. Prieur, “Intel Core 2 Duo - Test,” [Online] Available at: http://www.behardware.com/articles/623-16/intel-core-2-duo-test.html/, Date Accessed: March 1, 2016.

[ISO13] “Road vehicles — Vehicle to grid communication interface — Part 1: General information and use-case definition,” Reference number ISO 15118-1, 2013.

[JAN04] N. Jansma and B. Arredondo, “Performance Comparison of Elliptic Curve and RSA Digital Signatures,” Technical report, University of Michigan College of Engineering, April 2004.

[KAH03] A. Kahate, Cryptography and Network Security, Tata McGraw-Hill, 2003.

[KAL09] F. R. Kalhammer et al, "Plug-in hybrid electric vehicles: Promises, issues, and prospects," *In Proceedings* EVS24 International Battery, Hybrid and Fuel Cell Electric Vehicle Symp., 2009, pp. 1 - 11.

[KEO03] J. Keogh, “J2ME: The Complete Reference,” Liber Ekonomi, 2003.

[KHU10] H. Khurana et al, "Design Principles for Power Grid Cyber-Infrastructure Authentication Protocols," *In Proceedings* 43rd Hawaii International Conference on System Sciences (HICSS), January 2010, pp. 1-10.

[KUN11] N. Kuntze et al, “Interoperable Device Identification in Smart-Grid Environments,” *In Proceedings* IEEE PES General Meeting, July 2011, pp. 1-7.

[KUT09] V. Kute et al, "A Software Comparison of RSA and ECC," International Journal Computer Science and Applications, Vol 2, Issue 1, April-May 2009, pp. 43-59.

[LAK11] S. Lakshminarayanan, "Authentication and authorization for Smart Grid application interfaces," *In Proceedings* IEEE/PES Power Systems Conference and Exposition (PSCE), March 2011, pp. 1-5.

[LEN01] A. Lenstra and E. Verheul, "Selecting Cryptographic Key Sizes," Journal of Cryptology, August 2001, pp. 255-293.

[LIG99] Li Gong and S. Dodda, "Security assurance efforts in engineering Java 2 SE (JDK 1.2)," *In Proceedings* 4th IEEE International Symposium on High-Assurance Systems Engineering, 1999, pp. 1-93.

[LU11] X. Lu et al, "The Pulse Width Modulation and its Use in Induction Motor Speed Control," *In Proceedings* 2011 Fourth International Symposium on Computational Intelligence and Design (ISCID), Vol 2, October 2011, pp. 195-198.

[MIN] “Minicom(1)-Linux man page,” [Online] Available at:http://www.linux.die.net/man/1/minicom/, Date Accessed: March 1, 2016.

[MEN96] A. Menezes et al, *Handbook of Applied Cryptography*, 2nd edition, CRC Press, June 1996.

[MEN08] T. Mens, J. Fernandez-Ramil and S. Degrandsart, "The evolution of Eclipse," *In Proceedings* IEEE International Conference on Software Maintenance (ICSM 2008), September – October 2008, pp. 386-395.

[MIL86] V. Miller, “Use of Elliptic Curves in Cryptography,” Lecture Notes in Computer Sciences (218) on Advances in Cryptology---CRYPTO 85, June 1986, pp. 417-426.

[MIR] “MIRACL,” [Online] Available at: http://info.certivox.com/docs/miracl/, Date Accessed: March 1, 2016.

[MUL12] M. Multin, F. Allerding and H. Schmeck, "Integration of electric vehicles in smart homes - an ICT-based solution for V2G scenarios," *In Proceedings* IEEE/PES Innovative Smart Grid Technologies (ISGT), January 2012, pp. 1-8.

[NIC14] H. Nicanfar et al, “Efficient Authentication and Key Management Mechanisms for Smart Grid Communications,” IEEE Systems Journal, Vol 8, June 2014, pp. 629-640.

[NIM12] S. U. Nimbhorkar and L. G. Malik, "A Survey on Elliptic Curve Cryptography (ECC)," International Journal of Advanced Studies in Computers, Science and Engineering, Vol 1, Issue 1, 2012, pp. 1-5.

[OKA89] E. Okamoto and K. Tanaka, "Key Distribution System Based on Identification Information," IEEE Journal on Selected Areas in Communications, Vol 7, Issue 4, May 1989, pp. 481-485.

[ORD10] M. Ordean and M. Giurgiu, "Implementation of a security layer for the SSL/TLS protocol," *In Proceedings* 9th International Symposium on Electronics and Telecommunications (ISETC), 11-12 November 2010, pp. 209-212.

[PAN15] J. Pan et al, "An Internet of Things Framework for Smart Energy in Buildings: Designs, Prototype, and Experiments," IEEE Internet of Things Journal, Vol 2, Issue 6, December 2015, pp. 527-537.

[PAV13] A. J. Paverd and A. P. Martin, “Hardware Security for Device Authentication in the Smart Grid,” J. Cuellar (Ed.): SmartGridSec’12, Springer LNCS, Vol 7823, 2013, pp. 72-84.

[PIG11] D. Pigatto, N. Silva and K. Branco, “Performance Evaluation and Comparison of Algorithms for Elliptic Curve Cryptography with El-Gamal based on MIRACL and RELIC Libraries,” Journal of Applied Computing Research, Vol 1, Issue 2, 2011, pp. 95–103.

[QUI13] G. S. Quirino and E. D. Moreno, "Architectural Evaluation of Asymmetric Algorithms in ARM Processors," International Journal of Electronics and Electrical Engineering, Vol 1, Issue 1, March 2013, pp. 39-43.

[RAB10] N. M. Rabadi, "Anonymous group implicit certificate scheme," *In Proceedings* 7th IEEE Conference in Consumer Communications and Networking, January 2010, pp. 308-312.

[REL] “Relic-Toolkit,” [Online] Available at: http://code.google.com/p/relic-toolkit/, Date Accessed: March 1, 2016.

[SAD13] S. S. Sadistap et al, "Design and development of digital PID controller for DC motor drive system using embedded platform for mobile robot," *In Proceedings* IEEE 3rd International Conference on Advance Computing Conference (IACC), February 2013, pp. 52-55.

[SCI15] S. Sciancalepore et al, “Key Management Protocol with Implicit Certificates for IoT systems,” *In Proceedings* Workshop on IoT challenges in Mobile and Industrial Systems, May 2015, pp. 37-42.

[SHA15] S. Shah “Electric Vehicle to Supply Equipment Connection and Transfer of Data using I2SE Emulators,” 2015.

[SIF10] J. Sifakis, "Embedded systems design — Scientific challenges and work directions," *In Proceedings* Formal Methods in Computer-Aided Design (FMCAD), October 2010, pp. 11-11.

[SIN01] M. Singh, "Peering at Peer-to-Peer Computing," IEEE Internet Computing, Vol 5, Issue 6, November - December 2001, pp. 4-5.

[SSH] “OpenSSH,” [Online] Available at: http://www.openssh.com/, Date Accessed: March 1, 2016.

[SSL] “OpenSSL,” [Online] Available at: https://www.openssl.org/, Date Accessed: March 1, 2016.

[STA05] W. Stallings, Cryptography and Network Security, 4th edition, Pearson Education, 16 November 2005.

[TOY10] M. Toyran and S. Berber, "Efficient implementation of elliptic curve Diffie-Hellman (ECDH) key distribution algorithm in pool-based cryptographic systems (PBCSs)," *In Proceedings* IEEE 18th Signal Processing and Communications Applications Conference (SIU), April 2010, pp. 780-783.

[VAI06] B. Vaidya, D. Makrakis and H. T. Mouftah, “Authentication mechanism using one-time password for 802.11 wireless LAN,” *In Proceedings* First international computer science conference on Theory and Applications, Springer-Verlag, 2006, pp. 619-628.

[VAI11] B. Vaidya, D. Makrakis and H. T. Mouftah, "Efficient Authentication Mechanism for PEV Charging Infrastructure," *In Proceedings* IEEE International Conference on Communications (ICC), June 2011, pp. 1-5.

[VAI13] B. Vaidya, D. Makrakis and H.T. Mouftah, "Secure Communication Mechanism for Ubiquitous Smart Grid Infrastructure,” The Journal of Supercomputing, Vol 64, Issue 2, May 2013, pp. 435-455.

[VIE02] J. Viega, M. Messier and P. Chandra, “Network Security with OpenSSL: Cryptography for Secure Communications,” O'Reilly, June 2002.

[VIJ12] P. R. Vijayalakshmi and K. B. Raja, "Performance analysis of RSA and ECC in identity-based authenticated new multiparty key agreement protocol," *In Proceedings* International Conference on

Computing, Communication and Applications (ICCCA), February 2012, pp. 1-5.

[WAN06] Y. Wang et al. "Unified Signed-Digit Number Adder for RSA and ECC Public-Key Cryptosystems," *In Proceedings* IEEE Asia Pacific Conference on Circuits and Systems, December 2006, pp. 1655-1658.

[XUE04] J. Xue-juan, “The origin and the solution of the ‘Chinese remainder theorem’,” Journal of Jiujiang University (natural sciences), Issue 3, 2004, pp. 102-108.

[ZHA12] F. Zhao et al, "Secure authenticated key exchange with revocation for smart grid," *In Proceedings* 2012 IEEE/PES Innovative Smart Grid Technologies (ISGT), January 2012, pp. 1-8.

[ZHO11] X. Zhou and X. Tang, "Research and implementation of RSA algorithm for encryption and decryption," 2011 6th International Forum Strategic Technology (IFOST), Vol 2, August 2011, pp. 1118-1121.