

**IMPLEMENTATION OF THE IEEE 1609.2 WAVE
SECURITY SERVICES STANDARD**

by

Chad Christopher Mandy Jr

A Thesis Submitted to the Faculty of
The College of Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Science

Florida Atlantic University

Boca Raton, Florida

August 2016

Copyright 2016 by Chad Christopher Mandy Jr

**IMPLEMENTATION OF THE IEEE 1609.2 WAVE
SECURITY SERVICES STANDARD**

by


Chad Christopher Mandy Jr

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Imad Mahgoub, Department of Computer and Electrical Engineering and Computer Science, and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and Computer Science and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

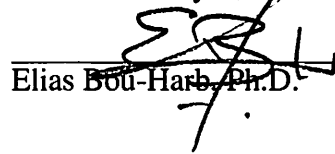
SUPERVISORY COMMITTEE:



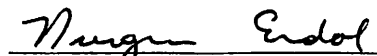
Imad Mahgoub, Ph.D.
Thesis Advisor



Mohammad Ilyas, Ph.D.



Elias Bou-Harb, Ph.D.



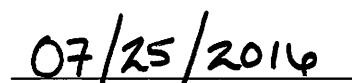
Nurgun Erdol, Ph.D.
Chair, Department of Computer and
Electrical Engineering and Computer Science



Mohammad Ilyas, Ph.D.
Dean, College of Engineering and
Computer Science



Deborah L. Floyd, Ed.D.
Dean, Graduate College


Date

ACKNOWLEDGEMENTS

I wish to express my deepest of thanks to my advisor, Dr. Imad Mahgoub, for his continual encouragement towards my research. His praise and guidance allowed me to overcome all of the obstacles that I ran into. Without his help, this work would not have been able to be accomplished. I would like to thank my committee members, Dr. Mohammad Ilyas and Dr. Elias Bou-Harb for taking the time to review and critique my work. Their feedback helped reinforce this work. I would also like to thank Dr. Monika Rathod for her continual dedication and devotion throughout the course of my research. I would also like to thank Alain, Henley, and Chancey for their assistance with the VMCD. Finally, I would like to thank Alexis and Travis for their continual support and for bearing with me as I conducted my research.

ABSTRACT

Author: Chad Christopher Mandy Jr
Title: Implementation of the IEEE 1609.2 WAVE Security Services Standard
Institution: Florida Atlantic University
Thesis Advisor: Dr. Imad Mahgoub
Degree: Master of Science
Year: 2016

This work presents the implementation of the the IEEE 1609.2 WAVE Security Services Standard. This implementation provides the ability to generate a message signature, along with the capability to verify that signature for wave short messages transmitted over an unsecured medium. Only the original sender of the message can sign it, allowing for the authentication of a message to be checked. As hashing is used during the generation and verification of signatures, message integrity can be verified because a failed signature verification is a result of a compromised message. Also provided is the ability to encrypt and decrypt messages using AES-CCM to ensure that sensitive information remains safe and secure from unwanted recipients. Additionally this implementation provides a way for the 1609.2 specific data types to be encoded and decoded for ease of message transmittance. This implementation was built to support the Smart Drive initiative's VANET testbed, supported by the National Science Foundation and is intended to run on the Vehicular Multi-technology Communication Device (VMCD) that is being developed. The VMCD runs on the embedded Linux operating system and this implementation will reside inside of the Linux kernel.

IMPLEMENTATION OF THE IEEE 1609.2 WAVE SECURITY SERVICES STANDARD

LIST OF TABLES	viii
LIST OF FIGURES	ix
1 Introduction	1
1.1 VANET	2
1.2 Problem Statement	2
1.3 Related Work	4
1.4 Contributions	6
1.5 Organization	7
2 IEEE WAVE Standards	8
2.1 WAVE Architecture	8
2.2 IEEE 1609.2: Security Services	10
3 Implementation	14
3.1 Hardware and Software Platform	14
3.2 Approach	15
3.3 Data Types	17
3.4 Key Pair Generation	18
3.5 Key Pair Validity	22
3.6 Elliptic Curve Digital Signature Algorithm	22
3.7 Elliptic Curve Integrated Encryption Scheme	27

3.7.1	Elliptic Curve Secret Value Derivation Primitive	27
3.7.2	Key Derivation Function	28
3.7.3	Message Authentication Code	29
3.7.4	Encryption	30
3.7.5	Decryption	31
3.8	Cryptomaterial Data Storage	33
3.9	Signing Ieee1609Dot2Data	36
3.10	Verifying Ieee1609Dot2Data	39
3.11	Encrypting Ieee1609Dot2Data	43
3.12	Decrypting Ieee1609Dot2Data	46
4	Protocol Conformance	51
5	Testing and Verification	54
6	Conclusion	64
6.1	Future Work	65
	Bibliography	66

LIST OF TABLES

1.1	Comparable key sizes [4]	3
5.1	Time required to generate a key pair in milliseconds.....	60
5.2	Time required to generate a signature in milliseconds	60
5.3	Time required to verify a signature in milliseconds	60
5.4	Time required to encrypt a symmetric key using ECIES in milliseconds	61
5.5	Time required to decrypt a symmetric key using ECIES in milliseconds	61
5.6	Time required to encode and decode sample <i>Ieee1609Dot2Data</i> in microseconds	61
5.7	Time required to generate self signed SPDUs of either raw data or an already encoded <i>Ieee1609Dot2Data</i> in milliseconds.....	61
5.8	Time required to both statically and ephemerally encrypt an SPDU in milliseconds	61
5.9	Time required to both statically and ephemerally decrypt an SPDU in milliseconds	61

LIST OF FIGURES

2.1	WAVE channel allocation [7]	8
2.2	WAVE protocol stack [7]	9
2.3	WAVE Security Services scope [10]	10
2.4	Data transfer between SDEEs [10]	12
3.1	Mother and daughter board for the VMCD [11]	14
3.2	Public and private key pair for NIST P-256	22
3.3	Public and private key pair for brainpoolP256r1	22
3.4	Signature generated for NIST P-256	23
3.5	Signature generated for brainpoolP256r1	23
3.6	A self-signed data structure containing the signature generated from the included <i>Ieee1609Dot2Data</i> message	36
3.7	A self-signed data structure containing the signature generated from the included raw message	37
3.8	Relevance tests for signed data [10]	42
5.1	Curve parameters and test vector verification for NIST P-256	55
5.2	5 freshly generated key pairs for NIST P-256	55
5.3	Curve parameters and test vector verification for brainpoolP256r1	56
5.4	5 freshly generated key pairs for brainpoolP256r1	56
5.5	The sample code used to verify encryption and decryption of a symmetric key using the NIST P-256 curve.	57

5.6	The results of the encryption and decryption of a symmetric key using NIST P-256	57
5.7	The sample code used to verify encryption and decryption of a symmetric key using the brainpoolP256r1 curve.	58
5.8	The results of the encryption and decryption of a symmetric key using brainpoolP256r1	58
5.9	The results of a sample ephemeral encryption and decryption.....	59
5.10	The results of a sample static encryption and decryption	60
5.11	The encoded structure of the decoded, encoded SPDU provided by [10]	60

CHAPTER 1

INTRODUCTION

Vehicular networks have been steadily increasing in popularity, all thanks to their proposed capabilities. VANET (Vehicular Ad Hoc Network) has two main categories of applications, safety-related and comfort/commercial [16]. Safety applications allow for the enhancement of human safety with applications such as driver assistance, alert information and warning alerts to be disseminated to other vehicles. Comfort applications are intended to increase the quality of life with applications such as weather reports, advertisements for nearby products and points of interest to look out. As information about everything going on around you is constantly transmitted, the need for security and privacy arises.

The *IEEE Standard for Wireless Access in Vehicular Environment - Security Services for Applications and Management Messages* [10], henceforth referred to as 1609.2, provides the ability to secure safety-critical WAVE (*Wireless Access in Vehicular Environments*) applications. In this standard, services and functionality are defined that are used to protect messages from various attacks, such as eavesdropping, spoofing and alteration of messages. Additionally methods have been defined that help prevent the leakage of personal or identifiable information to unauthorized parties. This implementation aims to provide these services to be used in conjunction with an existing WAVE protocol stack running on the VMCD (Vehicular Multi-technology Communication Device) [23].

1.1 VANET

Beginning with car phones, a rapid progression of communication technology in vehicles has lead towards an increasing interest in Vehicular Ad Hoc Networks (VANETs). This is not a new concept, and is instead an augmentation of Mobile Ad Hoc Networks (MANETs), with more constrained and predictable movement patterns [13]. These VANETs are collective groups of vehicles, exchanging data in a wireless network. Communication in VANET can take place over Wi-Fi, IEEE 802.11p, and cellular communication such as WiMAX or LTE. The creation of VANETs has driven the development of WAVE, a protocol stack that is built upon the vehicle specific IEEE 802.11p WLAN standard.

As VANET is an ad hoc network, there is no centralized infrastructure for communication. Each vehicle is a node, capable of transmitting to any other node in the network, this is known as vehicle-to-vehicle (V2V). Additionally, various standalone road side units (RSUs) are used to provide additional communication means, such as vehicle to infrastructure (V2I). No matter where the message comes from, each message is transmitted in a multi-hop fashion, originating from the source node and ending at the destination node. Routing protocols are various schemes that aim to efficiently disseminate data in VANET [16].

1.2 Problem Statement

Due to the nature of the messages in a VANET, it is important that messages are to be secured, unaltered and legitimate [28]. In addition to the message integrity requirements, VANET has other challenges: real time constraint, data consistency liability, and high mobility. Although the hardware that is eventually going to implement the WAVE protocol stack will be sufficiently powerful, a constraint for message transmission requires fast, but secure cryptographic algorithms. The use of Elliptic Curve Cryptography (ECC), as

Symmetric	ECC	RSA
80	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	15360

Table 1.1: Comparable key sizes [4]

opposed to the standard RSA cryptosystem, allows for comparable security with decreased key sizes. An ECC key of 233 bits has comparable strength to an RSA key of 2048 bits, see Table 1.1. Different ECC curves, such as NIST P-256 [22] and brainpoolP256r1 [24] will provide similar levels of security strength with different performance costs, depending on the reduction routines available. In regards to data consistency, receivers need to be able to filter out malicious messages from fake authenticated nodes.

With these constraints, the IEEE 1609.2 standard [10] was created for the WAVE (*Wireless Access in Vehicular Environments*) protocol stack. It aims to bring authentication, data verification, non-repudiation, and privacy protection. Authentication and data verification will be taken care of through verifying signatures. A signed message will contain a signature that can only be generated by some secret key that the sending node possesses, while allowing any node receiving the message to be able to verify the signature with the included public key. Signature generation and verification is handled by the Elliptic Curve Digital Signature Algorithm (ECDSA) as specified in 1609.2. A detailed description of this can be found in Section 3.6. As long as the sender keeps the private key unknown to other users, non-repudiation exists.

Privacy protection in VANET comes in many forms. It could be as simple as omitting personally identifiable information or as complex as obfuscating and encrypting said information. The 1609.2 standard provides methods to ensure that only the intended recipient has access to data contained within a message. The recipients section of a message allows for each recipient to check if they are allowed to view the message, by providing a way to decrypt an encrypted message to each recipient.

In this thesis, an implementation of the IEEE 1609.2 Security Services standard is provided to address the privacy and security concerns of vehicular communication. This implementation will support the Smart Drive Initiative, supported by the National Science Foundation, by providing another layer to the WAVE protocol stack, which will allow the VMCD to protect the end user and their information. As this is a critical feature of the WAVE stack, this thesis also provides testing and verification of the implementation to ensure its correctness and reliability.

1.3 Related Work

Since the newest version of the IEEE 1609.2 standard was ratified in March 2016 [10], there have been no implementations of the standard at the time of this research. There have however, been implementations of previous versions of the standard. The previous versions of 1609.2 specified different curves and cryptographic methods, causing slight compatibility issues with the current version.

[20] aims to implement a cryptographic system into a field-programmable gate array (FPGA). This implementation allows for a fast and efficient way to handle the security aspect of 1609.2, however the brainpoolP256r1 curve was not included in the standard and this implementation has no support for it. Based on the implementation of [20], the missing curve could be added with relatively trivial work. A major downside to the implementation is that it is hardware restricted. In order for someone to take advantage of it, they would need an FPGA added to the system. This lacks the portability that is addressed in the implementation that is being presented, as a result of the software implementation. [20] also only supports the cryptographic aspect of [10], meaning there is no protocol data unit support. A more complete implementation of the IEEE 1609.2 standard would include the encoding and decoding of messages, as well as the proper data format for transmission.

[21] provides an implementation that is more similar to the implementation provided in this thesis. The implementation was written in C, the same language as the presented

implementation. The software takes advantage of the openssl [30] toolkit to utilize pre-existing cryptographic functions, such as AES-CCM and SHA-256. As the software runs inside the user space section of the Linux operating system, it becomes challenging to handle cryptographic operations for data that will be received at the kernel level in some implementations [11]. Although [21] is able to encode messages in the proper octet format, this format has changed in newer versions of the standard. [21] was able to provide valuable algorithm description and testing data to be considered throughout the course of my research.

[19] proposed a group-based use of the IEEE 1609.2 standard for authentication. Their scheme relied on ECDSA for digital signature generation and AES-CCM for symmetric encryption. In this scheme, groups were composed of vehicles within 300 meters of each other. Each group selects a group leader, who generates a symmetric encryption key along with a public and private key pair for ECDSA. Whenever messages need to be disseminated amongst the group, the message is encrypted with the symmetric key and signed with the private key. For each message transmission this process will continue, resulting in authentication and confidentiality throughout transmission. As the 1609.2 standard is a framework message authentication, integrity and confidentiality, it is important for schemes like this to be created in order to fully utilize the standard.

Apart from the use cases of the IEEE 1609.2 standard, it is necessary for implementations to come up with ways to protect against attacks that target the various cryptographic operations themselves. In this implementation, side channel attacks are included within the Crypto portion, by means of extraneous random number generation and mathematical operations, at a very small cost in performance. Additional attacks such as denial of service (DoS) using invalid signatures, could be targeted towards the ECDSA signature verification aspect of 1609.2. [31] aims to mitigate signature based DoS attacks from insiders and outsiders. Outsider attacks, which are users outside of a group of vehicles sending invalid message signatures, are mitigated using an HMAC appended to

messages to ensure that message signatures will only be verified if the sender was authenticated to send the message. Insider attacks, which are authenticated users sending invalid message signatures, can be mitigated by using a threshold value to blacklist senders who continue to send invalid message signatures.

1.4 Contributions

The contributions of this work are as follows

- Implementation of the IEEE 1609.2 standard, including:
 - Support for both elliptic curves specified: NIST P-256 and brainpoolP256r1. This includes key generation, ECDSA signature generation and verification.
 - Support for ECIES encryption with either static provided keys or using ephemeral generated keys for the purpose of encrypting symmetric keys.
 - Support for the management of cryptomaterial, such as public and private key pairs, certificates and symmetric keys. The management functionality allows for the storage of user provided cryptomaterial or on the fly generation of new cryptomaterial. The storage is secured through an abstraction of the data via handles and only allows the SDS to view and use the data stored within. A user can only instruct the SDS how to handle the data.
 - Support for the generation of unsecured protocol data units (PDUs) for use internally in the SDS or for transmission of formatted data.
 - Support for the generation of signed PDUs. The signed PDU contains a signature that can be verified by an included public key, in order to determine the authenticity of a message. This is known as *self signing* and is the only supported method of signature generation in this implementation.
 - Support for the generation of encrypted PDUs, which allow for data to be transmitted securely without worrying of data being compromised. In this

implementation an encrypted PDU can contain an encrypted symmetric key to be decrypted with an already shared key, or can contain IDs of recipients who already have the means to decrypt the message directly.

- Support for the encoding and decoding of the *Ieee1609Dot2Data* for use within the SDS and for message transmission.
- Testing and Verification of the 1609.2 implementation, including:
 - Performance benchmarks for functionality provided in this implementation.
 - Verification that the key pairs generated are valid.
 - Verification that information is properly encrypted and decrypted.
 - Verification that data is properly encoded and decoded.

1.5 Organization

Following this chapter, the organization of this thesis is as follows: Chapter 2 describes a higher level view of the IEEE WAVE architecture and then an in depth description of the IEEE 1609.2 protocol. Chapter 3 details the actual implementation of the IEEE 1609.2 standard, with algorithms defining the functionality implemented. Chapter 4 describes how this implementation conforms to the protocol as described at the end of [10]. Chapter 5 outlines the testing and verification methodology used along with the results. Chapter 6 concludes the research done as well as discusses future research options.

CHAPTER 2

IEEE WAVE STANDARDS

2.1 WAVE Architecture

WAVE (wireless access in vehicular environments) is a new platform that allows for services to be provided to transportation [7]. It allows for communications between vehicles (V2V) as well as communication to transportation infrastructure (V2I). Communication is done via DSRC (dedicated short range communications) on 5.9GHz spectrum. This spectrum is divided into 10MHz sections, allocating room for 7 channels (1 control and 6 service channels), see Figure 2.1. The capability for an additional 2 channels (175 and 181) is provided by combining channels 174 and 176 and channels 180 and 182, respectively. Each of the channels available for use are defined in the IEEE 1609.4 Management Information Base [8]. The WAVE architecture itself, is broken up into a series of layers that form the WAVE protocol stack, see Figure 2.2. There are 4 standards that make up the majority of the protocol stack:

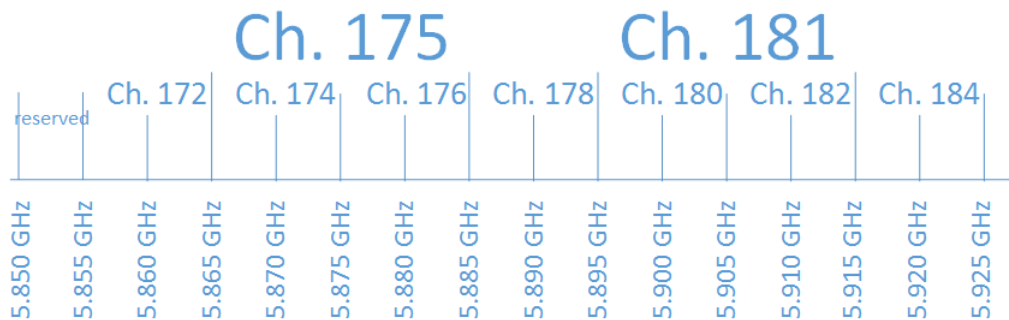


Figure 2.1: WAVE channel allocation [7]

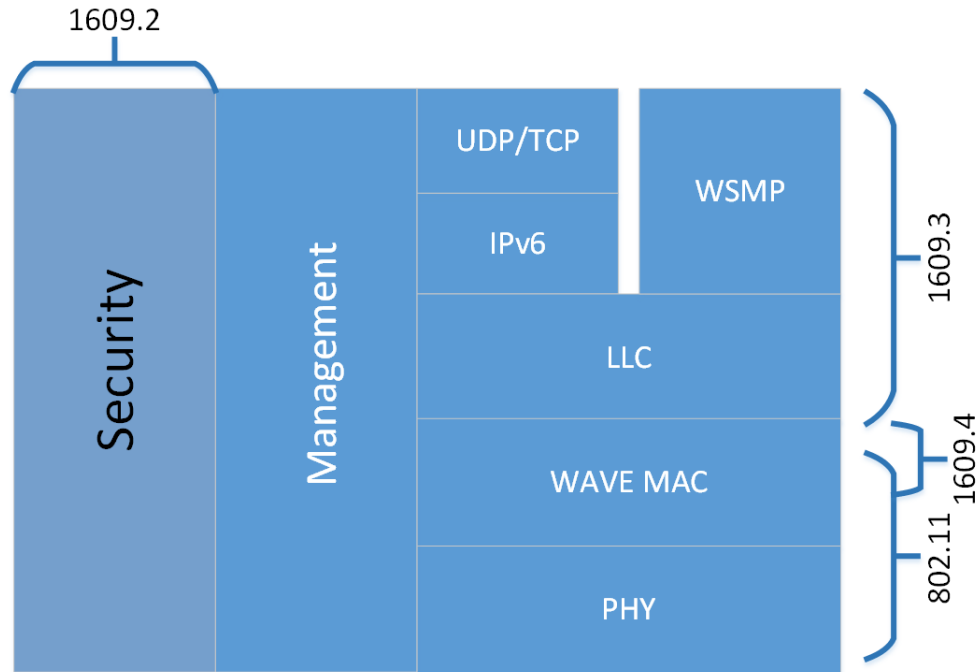


Figure 2.2: WAVE protocol stack [7]

- **1609.4** - This layer extends the MAC layer protocol to allow for multi-channel operations, such as channel switching and channel timing. It also provides support for vendor specific action (VSA) and timing advertisement (TA) frames [8].
- **1609.3** - This layer provides the WAVE short message protocol (WSMP) for transmitting messages. It also provides channel scheduling, service advertisements and the ability to use the existing internet protocols (IPv6 and LLC) for data transmission [9].
- **1609.2** - This layer is what the presented implementation focuses on. It provides communications security and authentication capabilities for service advertisements and WAVE short messages [10].
- **802.11p** - This layer manages the physical communications aspect of WAVE. 802.11p is an augmentation to the traditional 802.11 WLAN standard and it allows for communication to be handled on the 5.9GHz spectrum.

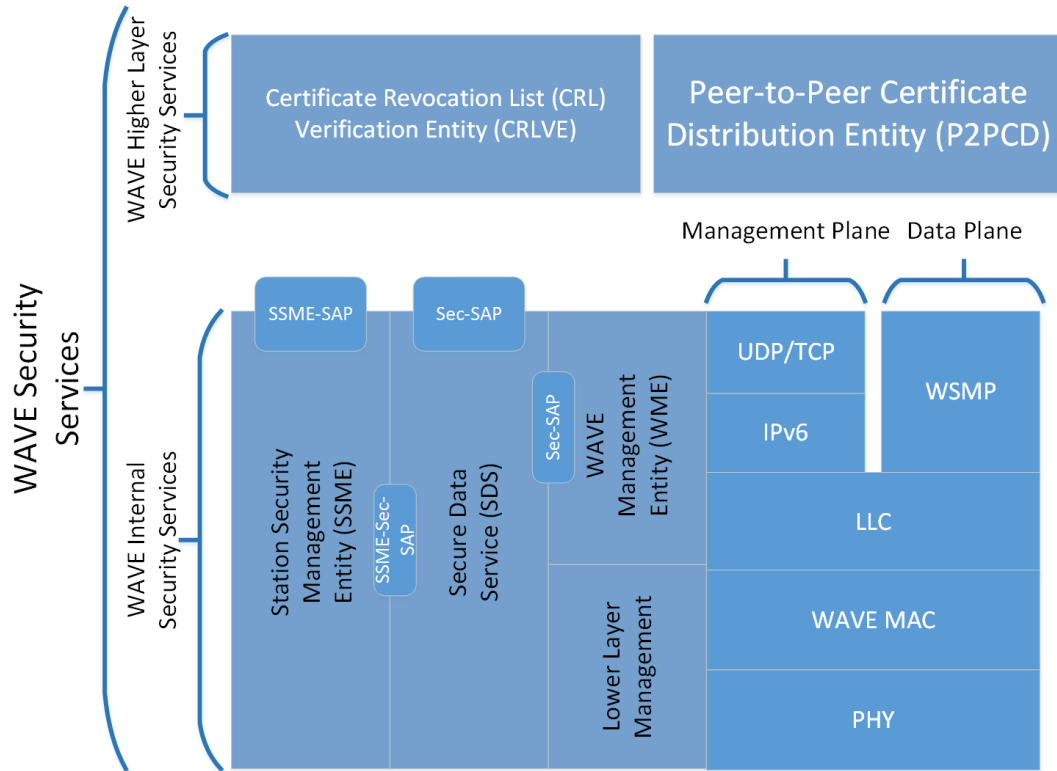


Figure 2.3: WAVE Security Services scope [10]

2.2 IEEE 1609.2: Security Services

The IEEE 1609.2 standard specifies the WAVE Security Services that should be available for WAVE devices, however this is not a constraint. These security services manage everything dealing with the encryption and decryption of secured data, certificate generation and validation, as well as signature generation and validation, all of which are supported by two elliptic curves for their cryptographic methods (NIST P-256 [22] and brainpoolP256r1 [24]). The security services are split into two sections, see Figure 2.3:

- **WAVE Internal Security Services**

- The primary aspect of the internal services is the secure data service (SDS). The SDS is responsible for the management of protocol data units (PDUs), including transforming PDUs into secured protocol data units (SPDUs) for transference between entities, as well as transforming SPDUs back into PDUs. Only the

management of PDUs is specified in this service; it is up to external secure data exchange entities (SDEE) to process the data held within.

- The security services management entity (SSME) is responsible for the management of certificate information and the storage of certificates. The SSME stores certificate information for certificates stored in the SDS and certificates that belong to Certificate Authorities (CAs). Information stored includes, the certificate data, when relevant revocation data was last received, when revocation data is expected to be received, the verification status of the certificate and whether the certificate is a trust anchor [10].

- **WAVE Higher Layer Security Services**

- The certificate revocation list (CRL) verification entity (CRLVE) is contained here. This is used to validate any CRL that is received and passes along any revocation data to the SSME.
- This section also holds the peer-to-peer certificate distribution (P2PCD) entity (P2PCDE). This allows devices unable to recognize certificate information to request the necessary information from peer devices.

For each of the services, there are service access points (SAPs) that allow the communication of information from the 1609.2 entities to other entities. In order for data to be exchanged, SDEEs need to invoke the SDS with a request to process the data, see Figure 2.4. This will be provided by the primitives defined in this implementation. The standard specifies that a data exchange takes place between two SDEEs, however the standard allows for either a single SDEE to be identified by the SDS or for multiple SDEEs to be identified. This implementation adheres to the former, with the SDEE ID of 0. In this implementation, the focus will be on the internal security services, which will include the generation and verification of signed SPDUs and the generation and decryption of encrypted SPDUs. SPDUs are created to either cryptographically protect the

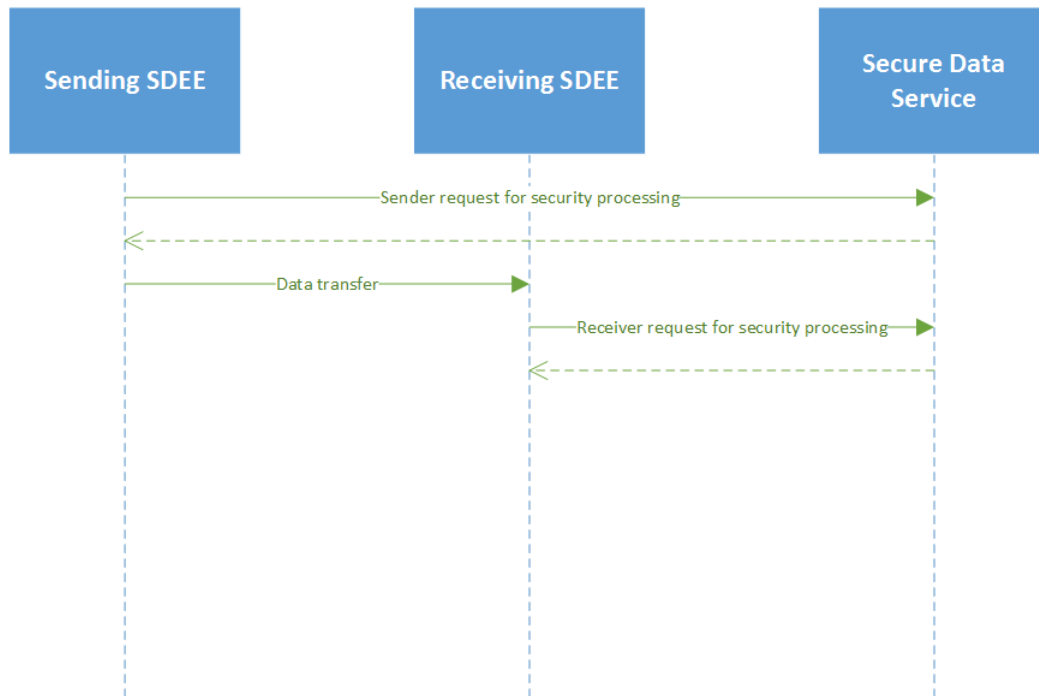


Figure 2.4: Data transfer between SDEEs [10]

data contained within or to provide security information that can be exchanged allowing for the processing of additional SPDUs. An SPDU is one of three types, unsecured, signed or encrypted and may itself contain an additional SPDU. For instance, an encrypted SPDU may be held within a signed SPDU to allow the received to be sure of the authenticity of the message before attempting to decrypt its contents.

In order to generate a signed SPDU, a call to the *Sec_SignedData(3.9)* primitive must be made. A successful call to this will return an encoded *Ieee1609Dot2Data* to the calling SDEE. A signed SPDU has 3 purposes:

1. To provide assurance that the sender is authenticated to be who they say they are.
2. To provide assurance that the signed SPDU cannot be modified without being detected.
3. To provide the assurance of the 2 aforementioned assurances to a third party recipient.

Encrypted SPDUs are generated with a call to the *Sec_EncryptedData(3.11)* primitive. The SDEE calling this primitive must supply it with either a PDU or SPDU and a key to

be used for the encryption. A successful call to this returns an encoded *Ieee1609Dot2Data* encrypted as requested by the SDEE. The purpose of an encrypted SPDU is to provide confidentiality, allowing only the intended recipients to view the contents of the SPDU. SPDUs may be cryptographically protected in more than one way. In fact, there is a potential unlimited number of ways to ensure the cryptographic protection of a message by simply continuing to pass along the SPDU returned from either of the 2 functions as the payload into another primitive, as many times as one desires.

Once an SPDU has been cryptographically protected, it is ready for transmission to another entity. Upon receipt of the SPDU, the entity may either request the verification of the signature with a call to the *Sec_SignedDataVerification*(3.10) primitive, or may request that the SPDU be decrypted with a call to the *Sec_EncryptedDataDecryption*(3.12) primitive. Throughout this standard, the reference to *Cryptomaterial* refers to symmetric keys, private keys and associated certificates, and digital certificates[10]. Instead of passing along sensitive information directly, an abstracted reference to the data is passes to the primitive. This is described in detail in (3.8).

The IEEE 1609.2 standard specifies the algorithms that must be used for cryptographic operations. The Elliptic Curve Digital Signature Algorithm (ECDSA) [22] is the only approved algorithm for signing and verifying messages. It is further described in (3.6). As the cryptographic operations rely upon the use of elliptic curve cryptography, two curves are specified for these operations, NIST P-256 [22] and brainpoolP256r1 [24], which restrict the private key size to be 256 bits, or 32 bytes. Any time data is to be hashed, the SHA-256[32] algorithm must be used, providing a 256 bit (32 byte) hash of any given data. There are two algorithms specified for encryption in 1609.2, the asymmetric encryption algorithm ECIES (Elliptic Curve Integrated Encryption Scheme), which is further described (3.7) and the symmetric encryption algorithm AES-CCM (Advanced Encryption Standard in Counter Mode with Cipher Block Chaining Message Authentication Code) [15].

CHAPTER 3

IMPLEMENTATION

3.1 Hardware and Software Platform

This implementation is an enhancement that is built on top of an existing Vehicular Multi-technology Communication Device (VMCD) [23]. This platform houses technology to provide cellular data, WiFi and WAVE communications. The VMCD is also equipped with a GPS sensor, USB 2.0, Ethernet, an SD/MMC card reader, audio I/O and VGA output. Being a two part system, see Figure 3.1, the VMCD uses a motherboard to control the primary I/O ports as well as housing an AMD Geode LX800 drive the software. The daughter board is used to provide extra I/O ports as well as the location sensors. On the software side, an embedded version of the Linux operating system is running on a modified Linux kernel based on version 3.4.39.

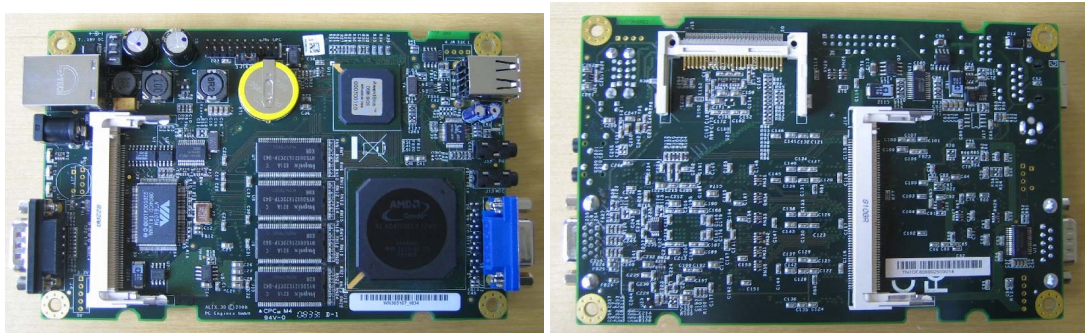


Figure 3.1: Mother and daughter board for the VMCD [11]

3.2 Approach

The VMCD is where all software developed in this implementation shall run, with the implementation being carried out inside of the Linux kernel. As the VMCD is an existing platform for the WAVE architecture, this implementation was done inside of a custom kernel module, in order to coexist with the existing 1609.3 and 1609.4 kernel modules. Doing this allows for easier integration with the rest of the WAVE stack, as a result of the kernel modules executing side by side. Due to software limitations on the VMCD itself, development of the kernel module is done inside of a virtual machine running the 32bit version of Ubuntu 12.04. Once the code is ready for execution on the VMCD, it is cross compiled using a custom toolchain created with [5]. The kernel module is divided into 4 main components:

1. IEEE1609dot2 — This component houses the data types and definitions for use within the IEEE 1609.2 Security Services Standard. Each of these data types were defined based on ASN.1 [1] modules designed to enable network transport of secure and unsecured 1609.2 specific data.
2. Crypto — This component provides an API for the following crypto-security functionality: Hashing octet strings using SHA-256 [32], symmetric encryption using AES-CCM [15] with a 128 bit block cipher, generation and validation of ECC key pairs using the NIST P-256 [22] and brainpoolP256r1 [24] elliptic curves, ECDSA signature generation and verification [3], asymmetric encryption using ECIES, an internal cryptomaterial storage, and internal functions to provide the aforementioned functionality.
3. Sec-SAP — This component is used to provide an API to higher layer entities and the WME for PDU (protocol data unit) management, such as creating PDUs, transforming PDUs into SPDUs (secure protocol data unit) and transforming

SPDUs into PDUs on receipt. High level cryptomaterial management is also contained within this component.

4. Utilities — This component provides functionality to aid the Sec-SAP with the management of PDUs.

As this implementation takes place in the Linux kernel, development is restricted to using the *C* programming language. With *C* being a procedural language, the object-oriented paradigm of higher level languages, such as C++, cannot be completely utilized, leaving an abstracted object based data structure based on the *C* struct. Data structures in this implementation are left visible, allowing users to inspect every field contained within. This is important as 1609.2 only provides a way for managing the security aspect of a PDU, not as much what to do with the information, leaving the data structures to be used as a way to easily organize data. As the messages are transmitted over an open medium, data is left visible by all parties and inherently does not contain sensitive information unless it has been secured by 1609.2

This implementation utilizes two libraries for the *Crypto* API. [12] provides the AES-CCM and SHA-256 implementations used within this standard. As the algorithms are fully implemented and have been tested against test vectors, no modifications were made to the library apart from reorganization and cleanup. The second library [25] used provided a framework for the Elliptic Curve Cryptographic operations. This library was heavily modified and the modifications are detailed in (3.4).

Throughout this implementation, functions and primitives will be defined and utilized. It is important to note that from a programming standpoint, the two are equivalent, however in this implementation, primitives are functions that are defined by the 1609.2 standard [10]. Primitives are given a specific naming convention, the various parameters that are to be passed in the function call, the effect of the function call, and how the call is created. In many cases, the standard calls for optional parameters to be used in the primitives. Due to limitations of the *C* standard, there is no strictly defined way for optional parameters to be

used in a function call. In this implementation, all optional parameters must be supplied when calling a primitive. If a caller wishes to exclude the parameter, the value of the parameter must be 0, NULL, empty, or any of the equivalent values. Each of the primitives will check the value of the parameters before attempting to use them.

The standard also calls for optional parameters in data structures. While dynamically created structure types are not possible, this implementation takes the same approach as optional parameters in primitives. All optional values in data types are to have their values set as 0 or some equivalent value for the respective type. For the purpose of encoding and decoding, data structures with optional parameters will contain a *bit_mask* field, that uses a defined list of flags to inform the user of the data, which fields are available. For data types containing variable types of information (PDUs can be either unsecured, signed or encrypted), a *choice* field is provided that uses an enumerator to identify the type of data held within. The encoding/decoding functions use this field to properly fill out the data.

3.3 Data Types

In order to allow data to be easily encoded/decoded [1], basic data types have been defined that are byte aligned. The basic types are named based on the number of bits they are defined from: *Uint3*, *Uint8*, *Uint16*, *Uint32*, and *Uint64*. Throughout this standard, the use of an Opaque data type is employed to hide sensitive information. Henceforth, *Opaque* will be used to describe a data structure that contains an octet string, defined as a byte/character array, and the length of said string, defined as an unsigned integer.

The PDU used by this implementation is represented as *Ieee1609Dot2Data*. This data structure contains the version of the protocol defined as an unsigned short, which in this implementation is 3, and it also contains data of the forms unsecured, signed and encrypted. Unsecured data is of the form *Opaque* and holds data that is to be used outside of the SDS. *SignedData* consists of an enumerated *HashAlgorithm*, representing the hash algorithm used for signature operations. The only algorithm supported by this standard is SHA-256

[32]. Each signed PDU contains security headers, stored in *HeaderInfo* that are used to describe the PSID of the message and the optional fields: generation time and location, expiry time, permission lists and an encryption key to be used with further communication.

EncryptedData contains the cipher text that has been symmetrically encrypted by AES-CCM [15] along with the *nonce*, a randomly generated value that is used to protect against replay attacks, used during the encryption. The only supported algorithm is represented as *aes128Ccm*. Also contained within *EncryptedData* is a list of *RecipientInfo* which contains information pertaining to the intended recipients of a message and how they are to decrypt the message.

One of the most used data structures, apart from the PDU (*Ieee1609Dot2Data*) itself, is the *EccP256CurvePoint*. This structure houses keys such as public and private keys, as well as signatures. The data structure is capable of holding the uncompressed (x and y) coordinates of the key, only the x coordinate, or the compressed y coordinate, with the later being unsupported in this implementation. Symmetric keys are stored in *SymmetricEncryptionKey*, which describes the type of key and holds the key itself. *PublicEncryptionKey* is derived from a *BasePublicEncryptionKey*, which contains an *EccP256CurvePoint* and a description of the algorithm used to create it. *EncryptionKey* is derived from either *SymmetricEncryptionKey* or *PublicEncryptionKey* and is used within the security headers.

3.4 Key Pair Generation

Key pairs for elliptic curves are generated on demand and as needed by the SDS using the *Crypto* API, which includes an enhanced version of an existing elliptic curve microprocessor library [25]. In order to support 256-bit numbers, the library provides methods for the manipulation of very large integers (VLI) for use with the elliptic curves. Existing routines implementing the key pair generation were modified to target the VMCD platform as follows:

1. Data types were adjusted to fit the x86 architecture.
2. Methods were adapted to utilize kernel specific libraries instead of standard C libraries.
3. Microprocessor specific assembly instructions were removed.
4. Various preprocessor directives and macros were removed to reduce code complexity and size.
5. Removed unnecessary functionality.
6. Support for the brainpoolP256r1 [24] curve was added.
7. Random number generation was modified to take advantage of the AMD Geodes random number generation functionality.
8. Added the ability to easily print out the curve parameters.
9. Added support for the key derivation function KDF2.
10. Added support for the message authentication code MAC1.
11. Added support for ECIES encryption and decryption.

In order to create a key pair, one can use one of two methods: *uECC_make_key(uint8_t *public_key, uint8_t *private_key, uECC_Curve)* or *uECC_compute_public_key(uint8_t *private_key, uint8_t *public_key, uECC_Curve)*. Both of these methods require the same parameters, however *uECC_make_key* returns a new private key and its corresponding public key, while *uECC_compute_public_key* returns the corresponding public key for the given private key. When using the *Crypto* API, cryptographic keys are represented as *uint8_t* arrays, which are single byte arrays. For the curves available for both methods, the public key must be 64 bytes long and the private key must be 32 bytes long, see Figure 3.2 for a NIST P-256 key pair and Figure 3.3 for a brainpoolP256r1 key pair. Both methods

are expecting initialized arrays and therefore must be initialized prior to being used. The API does not care what the public key is initialized to, as long as it points to 64 bytes of allocated memory. The private key needs to be initialized to a random number on either curve for use with *uECC_compute_public_key*, while the private key only needs to point to 32 bytes of allocated memory for *uECC_make_key*. Following the IEEE 1609.2 standard [10], two curves are available, the NIST P-256 [22] and the brainpoolP256r1 [24]. In this implementation it is recommended to use the NIST P-256 [22] curve if computational performance is desired. Specialized routines for use with the NIST P-256 [22] curve were provided by [2] to reduce the computational cost of mathematical operations. The curves NIST P-256 [22] and brainpoolP256r1 [24] can be initialized by calling *uECC_secp256r1* and *uECC_brainpoolP256r1* [24] respectively. Each function will return a static object of type *uECC_Curve*. *uECC_make_key* computes a key pair with Algorithm 1 and *uECC_compute_public_key* computes a public key with Algorithm 2.

Algorithm 1 Public and private key pair generation

```

1: procedure UECC_MAKE_KEY
2:   for 0 to 64 do
3:     Attempt to generate a random integer private_key
4:     if private_key is zero or the n domain parameter then
5:       Re-attempt to generate
6:       if last iteration then
7:         Return false
8:       end if
9:     else
10:      Break loop
11:    end if
12:  end for
13:  Multiply the curve base point  $G(G_x, G_y)$  with the private_key and save the result in
    public_key
14:  if public_key is zero then
15:    Return false
16:  else Return true
17:  end if
18: end procedure

```

Algorithm 2 Public key generation

```
1: procedure UECC_COMPUTE_PUBLIC_KEY
2:   if private_key is zero then
3:     Return false
4:   end if
5:   if private_key is the n domain parameter then
6:     Return false
7:   end if
8:   Multiply the curve base point  $\mathbf{G}(\mathbf{G}_x, \mathbf{G}_y)$  with the private_key and save the result in
   public_key
9:   if public_key is zero then
10:    Return false
11:   else
12:    Return true
13:   end if
14: end procedure
```

The byte array representation of public and private keys is only used to simplify cryptographic operations in the *Crypto* library, while use within the 1609.2 operations, the *EccP256CurvePoint* type is used. In this implementation, objects of type *EccP256CurvePoint* are limited to *x_only_type* for 32 byte long keys (private keys or the **r** component of an ECDSA signature) and *uncompressed_type* for 64 byte long keys (public keys). Helper functions `key_to_curvepoint(uint8_t *key, int size)` and `curvepoint_to_key(EccP256CurvePoint point, uint8_t *key)` were created and are used to convert between the two types. The `key_to_curvepoint` method expects either a 32 byte or 64 byte *key*, specified by the *size* parameter with 32 bytes returning an *EccP256CurvePoint* initialized to the type as specified above. The `curvepoint_to_key` method expects an initialized point of type *x_only_type* or *uncompressed_type*. The *key* parameter is expected to be initialized prior to the function call with a minimum of 32 or 64 bytes of memory respectively.

Representation of keys and signature is done via arrays, in order to simplify the elliptic curve operations. As a result the **r** and **s** components of a signature are contained within the same string, with the first 32 bytes being **r** and the last 32 bytes being **s**. The **x** and **y**

```
[ 20.385291] NIST P-256 Public key:
[ 20.408575] 5F 8C 74 7E 3A 70 BA DA AE 34 DF 68 50 42 5E FA BA 50 65 D1 E4 2F 73 B1 82 4D 8C FC AB 74 E6 64
[ 20.453437] 2A 20 1A 06 E1 26 8F F2 6D 7A 9C F4 69 95 2F 0F 75 50 3B 0C 17 18 AB 95 00 DA D9 8F C3 DE 52 9E
[ 20.491770] NIST P-256 Private key:
[ 20.515317] 58 32 59 F8 A2 C9 17 D5 6D 48 D2 85 3B 8F 71 89 70 11 14 57 54 D6 D8 31 47 31 01 BC D1 D2 3D 0F
```

Figure 3.2: Public and private key pair for NIST P-256

```
[ 21.956693] brainpoolP256r1 Public key:
[ 21.981000] 3F 6A 6D 7C DC F1 8E DD 03 29 09 59 06 43 81 75 8B 8D 55 54 3A 51 DA 20 FF D9 61 40 13 44 1D 97
[ 22.025350] 37 75 5F 33 A7 9C 1C 78 74 2F 03 72 42 C4 61 9B E1 0D 17 23 B7 D3 EE FF 16 64 9D 62 43 20 BC CB
[ 22.063374] brainpoolP256r1 Private key:
[ 22.087686] 74 E4 AD 65 C6 41 7D 0B 10 0A 9C D5 81 FE 7D 71 07 75 BD 10 F9 17 51 22 99 1C 20 A0 34 3E C8 BA
```

Figure 3.3: Public and private key pair for brainpoolP256r1

coordinates of a public key are represented in a similar fashion.

3.5 Key Pair Validity

Throughout this implementation, checks to determine if a given key pair is valid are performed. The function `is_keypair_valid(EccP256CurvePoint private_key, EccP256CurvePoint public_key, PKAlgorithm algorithm)` returns a boolean value indicating whether the two keys form a valid pair for the given *algorithm*. The function expects that the two keys are properly initialized and the *algorithm* is of a valid *PKAlgorithm* type. Part of the algorithm, see Algorithm 4 to determine if a key pair is valid, relies upon the use of `uECC_valid_public_key(const uint8_t *public_key, uECC_Curve curve)` which returns true if the key is a valid public key for the given curve using Algorithm 3.

3.6 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) [3] allows a signatory to sign a message of any given length, as long as the bit string of the message hash is no longer than **n** [10], while still allowing the verifier to verify that same signature. The signature is created and verified respectively using the private and public keys of the signatory. An ECDSA digital signature is generated by a call to `uECC_sign(const uint8_t *private_key, const uint8_t *message_hash, unsigned hash_size, uint8_t *signature, uECC_Curve curve)`

Algorithm 3 Determine if public key is valid

```
1: procedure UECC_VALID_PUBLIC_KEY
2:   if public_key is zero then
3:     Return false
4:   end if
5:   if coordinates  $X_Q$  and  $Y_Q$ , for public_key  $Q$ , are not in the range  $[1, q - 1]$  then
6:     Return false
7:   end if
8:   Compute  $((Y_Q)^2 \bmod q)$  and set it to a temporary variable tmp1
9:   Compute  $((X_Q)^3 + a * X_Q + b) \bmod q$  and set it to a temporary variable tmp2
10:  if tmp1 is equal to tmp2 then
11:    Return true
12:  else
13:    Return false
14:  end if
15: end procedure
```

```
31.556582] NIST P-256 signature (r,s):
31.574059] 02 93 CB FF 98 62 39 FB EA 9C 0E F5 EF FD 13 C8 FB 36 ED 60 BA 38 02 54 13 DD 50 D3 9A 9D 63 AB
31.617062] E6 E6 2A 0D CF 3B 14 29 41 05 30 B9 AA E0 C9 A0 39 B7 4E 97 B1 84 23 CE AC 24 24 3E 9D 89 F0 06
```

Figure 3.4: Signature generated for NIST P-256

with a return value of true for a successful signature generation and false for a failed signature generation. The *private_key* of the signatory is expected to be valid and not equal to zero. In this implementation, the signature generation function expects an already computed *message_hash* of specified *hash_size*. The version of IEEE 1609.2 [10] that the implementation is based on, only allows SHA-256 [32] to be used as a hash function, limiting *hash_size* to 32 bytes. Upon successful generation of the message signature, the resulting (r,s) is stored in *signature*, which is expected to be an array of 64 bytes, with memory already allocated. The *curve* to be used with the signature generation must be the same curve for the given *private_key* and its corresponding public key, see Figure 3.4 for a NIST P-256 signature and Figure 3.5 for a brainpoolP256r1 signature. The algorithm for *uECC_sign* can be seen in Algorithm 5. In order for the verifier to verify the signature

```
34.718138] brainpoolP256r1 signature (r,s):
34.736050] A8 A2 D4 C6 10 6F F8 62 28 35 3F 53 5E 6D 03 51 66 BC E0 A8 C7 56 9E 90 BF A5 0E 7A D6 62 91 65
34.777712] 8A D7 CA 7F 54 22 59 AB AE 5A 51 8D 21 B1 EB 23 FE 30 C2 AE 37 10 58 37 3A 40 40 35 2E 97 70 15
```

Figure 3.5: Signature generated for brainpoolP256r1

Algorithm 4 Determine if key pair is valid

```
1: procedure IS_KEYPAIR_VALID
2:   Convert the private key from an EccP256CurvePoint to a 32 byte array
3:   if curvepoint_to_key returns false then
4:     Return false
5:   end if
6:   Convert the public key from an EccP256CurvePoint to a 64 byte array
7:   if curvepoint_to_key returns false then
8:     Return false
9:   end if
10:  Determine the curve to be used based on the given algorithm
11:  if algorithm is eciesNistP256 or ecdaNistP256WithSha256 then
12:    initialize the curve to NIST P-256 [22]
13:  else if algorithm is eciesbrainpoolP256r1 [24] or ecdsabrainpoolP256r1
    [24]WithSha256 then
14:    Initialize the curve to brainpoolP256r1 [24]
15:  elseReturn false
16:  end if
17:  Determine if the public_key for the curve is a valid point with
    uECC_valid_public_key
18:  if uECC_valid_public_key returns false then
19:    Return false
20:  else
21:    Compute a public key, public, for the given private_key and curve
22:    if Failed to compute public then
23:      Return false
24:    else
25:      Compare the computed public to the given public_key
26:      if keys are equivalent then
27:        Return true
28:      elseReturn false
29:      end if
30:    end if
31:  end if
32: end procedure
```

Algorithm 5 Signature generation

```
1: procedure UECC_SIGN
2:   for 0 to 64 do
3:     Generate a random number and store it in  $k$ 
4:     if Generation is not successful then
5:       Break and try again
6:     end if
7:     if  $k$  does not fall in the range  $[1, n - 1]$  then
8:       Break and try again
9:     end if
10:    Compute  $(k * G)$  and store the result in  $p$ 
11:    if  $p$  is zero then
12:      Break and try again
13:    end if
14:    Generate a random number and store it in  $tmp$ 
15:    if Generation is not successful then
16:      Break and try again
17:    end if
18:    Compute  $((k * tmp) \bmod n)$  and store the result in  $k$ 
19:    Compute  $((k^{-1}) \bmod n)$  and store the result in  $k$ 
20:    Compute  $((k * tmp) \bmod n)$  and store the result in  $k$ 
21:    Set the first 32 bytes of signature equal to  $p$ 
22:    Compute  $((private\_key * p) \bmod n)$  and store the result in  $s$ 
23:    Convert message_hash into an integer and store the result in  $tmp$ 
24:    Compute  $((k * (s + tmp)) \bmod n)$  and store the result in  $s$ 
25:    if  $s$  has more than 256 bits then
26:      Break and try again
27:    end if
28:    Set bytes 33 to 64 of signature to  $s$ 
29:    Return true
30:  end for
31:  Return false
32: end procedure
```

generated by the signatory, a call must be made to `uECC_verify(const uint8_t *public_key, const uint8_t *message_hash, unsigned hash_size, const uint8_t *signature, uECC_Curve curve)`, which returns true if the signature is valid for the given public key. This function expects that `public_key` will be an initialized 64 byte array containing the public key corresponding to the signatory's private key. Similar to the signature generation function, the `message_hash` must be an already calculated hash of the message to be verified using SHA-256 [32] with a `hash_size` of 32 bytes. The `signature` parameter shall be the calculated signature for the given message, using the signatory's private key. The `curve` must be the same curve the signatory used to sign the message, which is either NIST P-256 [22] or brainpoolP256r1 [24]. The algorithm for the verification of a signature is can be seen in Algorithm 6.

Algorithm 6 Signature generation

```

1: procedure UECC_VERIFY
2:   Split signature into its two components r and s
3:   if r or s is not within the range [1, n-1] then
4:     Return false
5:   end if
6:   Compute  $((s^{-1}) \bmod \mathbf{n})$  and store the result in z
7:   Convert message_hash into an integer and store the result in u1
8:   Compute  $((z * u_1) \bmod \mathbf{n})$  and store the result in u1
9:   Compute  $((z * r) \bmod \mathbf{n})$  and store the result in u2
10:  Compute  $(\mathbf{G} + \textit{public\_key})$  and store the result in sum
11:  Compute  $((u_1 * \mathbf{G}) + (u_2 * \textit{public\_key}))$  using Shamir's Trick [14] and store the
    result in  $(x_R + y_R)$ 
12:  Compute  $(x_R \bmod \mathbf{n})$  and store the result in xR
13:  if r is equal to xR then
14:    Return true
15:  else
16:    Return false
17:  end if
18: end procedure

```

3.7 Elliptic Curve Integrated Encryption Scheme

The Elliptic Curve Integrated Encryption Scheme (ECIES) allows asymmetric encryption/decryption to be performed on supplied data [27]. In the 1609.2 standard [10], ECIES is reserved for encrypting symmetric keys to be used with AES-CCM [15] when encrypting PDU data. The standard specifies implementation of ECIES with the following constraints: ephemeral public key generation, utilization of Elliptic Curve Secret Value Derivation Primitive Diffie-Hellman with cofactor multiplication (ECSVDP-DHC) to derive secret values, a stream cipher based on Key Derivation Function 2 (KDF2) for encryption in non-DHAES mode, and the message authentication code should be MAC1 [10].

3.7.1 Elliptic Curve Secret Value Derivation Primitive

ECSVDP-DHC is used to create a shared secret key between two parties. Party A has a key pair ($public_A$, $private_A$) and party B has a key pair ($public_B$, $private_B$). Party A shares $public_A$, with party B who then computes the shared secret key, $shared_B$, using $public_A$ and $private_B$, while party B shares $public_B$, with party A who then computes the shared secret key, $shared_A$, using $public_B$ and $private_A$. The two shared secret keys will be equivalent. This works because of how the public keys are generated:

$$public_A = private_A * G$$

$$public_B = private_B * G$$

$$shared_A = public_B * private_A = private_B * G * private_A$$

$$shared_B = public_A * private_B = private_A * G * private_B$$

$$shared_A = shared_B$$

. The shared secret key is calculated by a call to *uECC_shared_secret(const uint8_t *public_key, const uint8_t *private_key, uint8_t *secret, uECC_Curve curve)* which returns true on a successful shared secret generation. The *public_key* and *private_key* are expected to be properly initialized keys, with the *public_key* coming from the other party and the *private_key* coming from the calling party. The resulting shared secret key will be stored in *secret*, which is expected to be an initialized 32 byte array. The *curve* used by both parties must be the same as the curve that was used to for the key pair generation. The algorithm for *uECC_shared_secret* is Algorithm 7.

Algorithm 7 Shared secret generation

```

1: procedure UECC_SHARED_SECRET
2:   Compute (public_key * private_key) and store the result in secret
3:   if secret is not zero then
4:     Return true
5:   else
6:     Return false
7:   end if
8: end procedure

```

3.7.2 Key Derivation Function

A key derivation function will be applied to the shared secret key returned from the aforementioned method in order to derive a stream cipher to be used with encryption. The KDF2 function is constrained to using SHA-256 for its hash function and the key derivation parameter P1, which is determined as follows: the hash of the certificate if the encryption key was obtained from it, the hash of the Ieee1609Dot2Data PDU if the encryption key was obtained from the SignedData within, or the hash of the empty string if the encryption key was obtained from another method. The counter threshold for the KDF2 function is hard coded to 2 and is obtained from the ceiling calculation with an output length of 48 bytes (16 bytes for the MAC1 tag and 32 bytes for the shared secret) and an input length of 32 bytes. In order to derive a key from the shared secret key a call to *KDF2(BYTE *K, BYTE *secret, BYTE *P1)* must be made. This function expects an

allocated 48 byte array K , an initialized 32 byte array $secret$ and the initialized $P1$ parameter as described above. The algorithm for KDF2 [26] is Algorithm 8.

Algorithm 8 Key derivation function 2

```

1: procedure KDF2
2:   Initialize  $d$  to  $((16 + 32) + 32 - 1) / 32 = 2$ 
3:   Initialize  $counter$  to 1
4:   for  $counter$  to  $d$  do
5:     Convert  $counter$  into a string and store it in  $C$ 
6:     Initialize the  $SHA256\_CTX$ ,  $ctx$ 
7:     Set the  $ctx$  buffer to  $secret$ 
8:     Append  $C$  to the buffer
9:     Append  $P1$  to the buffer
10:    Compute the hash of the concatenated strings and store the result in  $buf$ 
11:    Store  $buf$  into the index  $(32 * (counter - 1))$  of  $T$ 
12:    Increment  $counter$ 
13:  end for
14:  Store the first 48 bytes of  $T$  into  $K$ 
15: end procedure

```

3.7.3 Message Authentication Code

A message authentication code is used as an integrity check on secret keys when information is transmitted over an unreliable or untrustworthy medium [17]. MAC1 is a keyed-hash message authentication code that provides what is known as a tag, to be used by the recipient of the message. The MAC1 algorithm is also constrained in this standard [10] to using SHA-256 for the hash function, with the length of the tag being 128 bits, the input key length being 256 bits and then the empty string for its extra shared material. A tag is generated by calling $MAC1(BYTE *tag, BYTE *K, BYTE *m)$. The tag is expected to be a 16 byte array with memory already allocated, K is a 16 byte array initialized to something other than 0, and m is expected to be a 32 byte array initialized. In this implementation, $MAC1$ is reserved strictly for internal generation of tags and uses hard coded array lengths for simplicity. The algorithm for $MAC1$ is Algorithm 9.

Algorithm 9 Message authentication code 1

```
1: procedure MAC1
2:   Fill two 64 byte arrays with 0s and store them in opad and ipad
3:   Copy K into the first 32 bytes of opad and ipad
4:   Set i equal to 0
5:   for i to 64 do
6:     XOR index i of opad with 0x5c
7:     XOR index i of ipad with 0x36
8:     Increment i
9:   end for
10:  Initialize the SHA256_CTX, ctx
11:  Set the ctx buffer to ipad
12:  Append m to the buffer
13:  Compute the hash of the concatenated strings and store the result in buf
14:  Initialize the SHA256_CTX, ctx
15:  Set the ctx buffer to opad
16:  Append buf to the buffer
17:  Compute the hash of the concatenated strings and store the result in buf
18:  Store the first 16 bytes of buf into tag
19: end procedure
```

3.7.4 Encryption

When plaintext data is to be encrypted using the ephemeral approach, the symmetric data encryption key is encrypted with ECIES for each of the intended recipients. The ECIES encryption function generates an encrypted symmetric key using the function *ECIES_EncryptSymmKey*(*BYTE *ephemeral_public_key*, *BYTE *cipher_text*, *BYTE *tag*, *BYTE *message*, *BYTE *recip_info*, *BYTE *recip_public_key*, *uECC_Curve curve*). The *ephemeral_public_key* is expected to be a 64 byte array with previously allocated memory and will contain the public key used by the function for further handling. The *cipher_text* is expected to be a 16 byte array with memory allocated and will contain the encrypted symmetric key. The *tag* is expected to be a 16 byte array with memory allocated and will contain the resulting MAC1 authentication tag. The key derivation parameter will be stored in *recip_info* and is expected to be defined as mentioned in (3.7.2). The symmetric key to encrypt will be held in *message* and is expected to be an initialized 16 byte array. For each recipient a call will be made to this function with their respective public key and

will be stored in a 64 byte array *recip_public_key*. In this implementation, certificates are not supported, so *ECIES_EncryptSymmKey* will only be called one time. The standard does not specify a way for receiving the public key of a recipient, however the primitive defined to encrypt the PDU only supports multiple recipients through the use of certificates. Finally, the *curve* to be used with the encryption must be the same curve used to generate the ephemeral key pair, the recipients key pair and the shared secret key. *ECIES_EncryptSymmKey* will return the *ephemeral_public_key*, the *cipher_text* and the *tag* with Algorithm 10.

Algorithm 10 Elliptic curve integrated encryption scheme for symmetric key encryption

```

1: procedure ECIES_ENCRYPTSYMMKEY
2:   Generate a key pair (1) on the curve and store the public and private keys in
   ephemeral_public_key and ephemeral_private_key respectively
3:   Generate the shared secret value (7) with recip_public_key and
   ephemeral_private_key on the curve and store the result in ephemeral_secret_key
4:   Derive the shared secret key (8) using ephemeral_secret_key and recip_info and
   store the result in shared_secret_key
5:   Set i equal to 0
6:   for i to 16 do
7:     XOR index i of message with index i of shared_secret_key and store the result
   in index i of cipher_text
8:     Increment i
9:   end for
10:  Generate the message authentication code (9) using the last 32 bytes of
   shared_secret_key and cipher_text and store the result in tag
11: end procedure

```

3.7.5 Decryption

In order to decrypt a symmetric key for use with decrypting an SPDU, a call to *ECIES_DecryptSymmKey*(*BYTE *ephemeral_public_key*, *BYTE *cipher_text*, *BYTE *tag*, *BYTE *message*, *BYTE *recip_info*, *BYTE *recip_private_key*, *uECC_Curve curve*) must be made. This function is very similar to *ECIES_EncryptSymmKey*, however the variables that are input/output are swapped. The *ephemeral_public_key* is expected to be the initialized 64 byte array that was generated during the *ECIES_EncryptSymmKey* function.

The symmetric key to be decrypted is stored in an initialized 16 byte array known as *cipher_text*. The *tag* is expected to be a 16 byte array allocated to memory, which the recipient will compare to the tag received alongside the ephemeral public key and cipher text, in order to determine the integrity of the message. The decrypted *cipher_text* will be stored in *message*, which is expected to be a 16 byte array previously allocated to memory. The additional data parameter for use with decryption is stored in *recip_info*. The data this represents was previously defined in (3.7.2), however in this implementation the empty string is always assumed as certificates and encryption keys in the security header of a signed PDU are not supported. The private key corresponding to the recipients public key is stored in *recip_private_key* and is expected to be an initialized 32 byte array that is not equal to 0. The elliptic curve used in the decryption must be the same curve used for both the encryption and generation of all key pairs involved, and will be stored in *curve*. The decrypted symmetric key and tag, will be returned from *ECIES_DecryptSymmKey* with the Algorithm 11.

Algorithm 11 Elliptic curve integrated encryption scheme for symmetric key decryption

```

1: procedure ECIES_DECRYPTSYMMKEY
2:   Generate the shared secret value (7) with ephemeral_public_key and
   recip_private_key on the curve and store the result in ephemeral_secret_key
3:   Derive the shared secret key (8) using ephemeral_secret_key and recip_info and
   store the result in shared_secret_key
4:   Set i equal to 0
5:   for i to 16 do
6:     XOR index i of cipher_text with index i of shared_secret_key and store the result
     in index i of message
7:     Increment i
8:   end for
9:   Generate the message authentication code (9) using the last 32 bytes of
   shared_secret_key and cipher_text and store the result in tag
10: end procedure

```

3.8 Cryptomaterial Data Storage

In this implementation, cryptomaterial refers to any private keys and their corresponding public keys and certificates. In order to keep the private keys hidden, as well as ensuring their integrity by preventing modification to the keys, a data storage system was created for use by the SDS. There are two types of cryptomaterial to be stored, *Cryptomaterial* and *SymmCryptomaterial*. Each of the cryptomaterial types have their own respective storage array, capable of holding 65536 instances. In order to use the stored data, the SDEE must supply the SDS with a cryptomaterial handle (cmh or scmh), which is a randomized index in the array. The handle is simply an integer that refers to data stored within the SDS and provides no direct access or manipulation of the data. Each of the storages hold an internal state as well as an *algorithm*, referring to how the cryptomaterial was generated.

The *Cryptomaterial* has 4 states: *CMHS_uninitialized*, *CMHS_initialized*, *CMHS_key_pair* and *CMHS_key_certificate*. *CMHS_uninitialized* refers to an instance of *Cryptomaterial* that is currently unused by the SDS. *CMHS_initialized* indicates that a handle has been assigned that references this instance. *CMHS_key_pair* and *CMHS_key_certificate* refer to an in-use *Cryptomaterial* that contains an *algorithm* describing how the *Cryptomaterial* was created as well as a private key and it's associated public key for the former, and a private key and it's associated certificate for the latter. In this implementation certificates are not supported, however the storage system allows for certificates to be used in future enhancements of this implementation.

The *SymmCryptomaterial* is used to store symmetric keys for use with AES-CCM [15] encryption. The only supported *algorithm* for use with this standard [10] is *aes128Ccm*. There are only two states for this storage: *SCMHS_uninitialized* and *SCMHS_initialized*. The former refers to a handle that is not in use, while the latter refers to a handle that references a symmetric key.

The standard [10] requires primitives to be defined for handling cryptomaterial. If the SDEE wishes to obtain a cmh, a call must be made to *Sec_CryptomaterialHandle*. This

primitive returns a newly initialized cmh with Algorithm 12. A key pair can be initialized

Algorithm 12 Cryptomaterial handle generation

```

1: procedure SEC_CRYPTOMATERIALHANDLE
2:   Generate a random integer in the range [0, 65535] and store it in rnd
3:   Initialize i to rnd
4:   for i to 65536 do
5:     if Cryptomaterial referenced by i is uninitialized then
6:       Initialize it and return i
7:     end if
8:     Increment i
9:   end for
10:  Initialize i to rnd
11:  for i to 0 do
12:    if Cryptomaterial referenced by i is uninitialized then
13:      Initialize it and return i
14:    end if
15:    Decrement i
16:  end for
17:  Return -1
18: end procedure

```

by calling *Sec_CryptomaterialHandle_GenerateKeyPair(int cmh, PKAlgorithm algorithm)*. This primitive expects a valid *cmh* in the range [0, 65535] and will generate a key pair for the respective algorithm. On completion of this primitive, a public key will be returned with Algorithm 13. If one does not wish to generate a key pair, the caller may provide the SDS with a key pair of their choice by calling *Sec_CryptomaterialHandle_StoreKeyPair(int cmh, PKAlgorithm algorithm, EccP256CurvePoint public_key, EccP256CurvePoint private_key)*. This primitive will store the *public_key* and *private_key* at the specified *cmh* for the given *algorithm*. The *cmh* is expected to be a valid handle in the range [0, 65535]. The primitive will return the *public_key* upon completion of Algorithm 14.

Once an SDEE is finished with a given *Cryptomaterial*, it may delete all of the associated data for a given *cmh*. The *Sec_CryptomaterialHandle_Delete(int cmh)* primitive will clear all data referenced by the *cmh* and set the referenced *Cryptomaterial* back to uninitialized.

Algorithm 13 Cryptomaterial key pair generation

```
1: procedure SEC_CRYPTOMATERIALHANDLE_GENERATEKEYPAIR
2:   Determine the curve to be used based on the given algorithm
3:   if algorithm is eciesNistP256 or ecdsaNistP256WithSha256 then
4:     Set the curve to NIST P-256 [22]
5:   else if algorithm is eciesBrainpoolP256r1 or ecdsaBrainpoolP256r1WithSha256
   then
6:     Set the curve to brainpoolP256r1 [24]
7:   else
8:     Return an empty public_key
9:   end if
10:  Generate a key pair (3.4) with the given curve and store the public key in public
   and the private key in private
11:  Convert public into an EccP256CurvePoint, public_key
12:  Convert private into an EccP256CurvePoint, private_key
13:  if public_key is initialized and private_key is initialized then
14:    Set the Cryptomaterial algorithm to algorithm
15:    Set the Cryptomaterial state to CMHS_key_pair
16:    Store the public_key and private_key
17:  end if
18:  Return public_key
19: end procedure
```

Algorithm 14 Cryptomaterial key pair storage

```
1: procedure SEC_CRYPTOMATERIALHANDLE_STOREKEYPAIR
2:  Determine the validity of the given key pair (3.5) for the algorithm
3:  if key pair is valid then
4:    Set the Cryptomaterial algorithm to algorithm
5:    Set the Cryptomaterial state to CMHS_key_pair
6:    Store the public_key and private_key
7:    Return public_key
8:  end if
9:  Return public_key
10: end procedure
```

```

[ 27.538941] SPDU:
[ 27.558053] 03 81 00 40 03 80 41 C0 66 99 B3 21 D3 8E 83 98 54 82 0C 7F 37 3C 62 7A 12 DA 49 F0 63 47 1C 28
[ 27.603458] BD 10 6B DC E1 24 B3 74 2A DD D3 55 27 1A BC CF E6 3B 13 14 F4 E3 D0 71 AE BF 99 C9 58 69 91 92
[ 27.648848] 0F 44 4D C6 5B F8 F5 03 40 01 04 FF FF EB 9E 3D 2C DD C0 82 00 80 80 2D 73 05 0E 45 24 55 4A 8A
[ 27.694188] C2 0D 65 1F 45 44 0F 11 83 B5 7E 05 25 CC C6 17 90 59 84 6B 0D D2 32 27 40 13 8A D3 55 C1 22 F5
[ 27.739620] AD 6A AE 8F 7F 4A B1 B6 92 40 E6 FD 35 DE C3 FC 98 6C F8 BD 32 32 93

```

Figure 3.6: A self-signed data structure containing the signature generated from the included *Ieee1609Dot2Data* message

An SDEE may also choose to utilize symmetric keys and can store its *SymmCryptomaterial* data. The *Sec_SymmetricCryptomaterialHandle(SymmAlgorithm algorithm, boolean generate, Opaque key_bytes)* primitive allows the SDEE to either store a given *key_bytes* or generate a new symmetric key for the given *algorithm*. This primitive expects that the *algorithm* is *aes128Ccm*, otherwise the primitive returns an invalid *scmh*. When calling this primitive one may choose to either generate a new key by setting *generate* to true or one will initialize the *SymmCryptomaterial* with *key_bytes* which is expected to be 16 bytes long and initialized. Successful storage in either case will result in the primitive returning a valid *scmh* with Algorithm 15.

The standard [10] allows for a way to retrieve the *HashedId8* of a symmetric key for use within the SDS. The *HashedId8* is the 8 lower order bytes of the SHA-256 hash [32] of the respective symmetric key. The primitive defined for this is *Sec_SymmetricCryptomaterialHandle_HashedId8(int scmh)* and returns the *HashedId8* for the given *scmh* with Algorithm 16. The primitive expects that the *scmh* is in the range [0, 65535].

Much like a *Cryptomaterial*, a primitive is supplied to delete the *SymmCryptomaterial* referenced by an *scmh*. A call to the *Sec_SymmetricCryptomaterialHandle_Delete(int scmh)* will clear the data referenced by the *scmh* and set the *SymmCryptomaterial* state to uninitialized.

3.9 Signing Ieee1609Dot2Data

In order for a sender to prove that they are indeed who they say they are, the sender must sign their message in a way that allows anyone to verify the signature with the provided

Algorithm 15 Symmetric cryptomaterial handle generation

```
1: procedure SEC_SYMMETRICCRYPTOMATERIALHANDLE
2:   if algorithm is not aes128Ccm then
3:     Return -1
4:   end if
5:   if generate is true then
6:     Generate a new symmetric key and store it in bytes_to_store
7:   else if generate is false then
8:     if key_bytes is 16 bytes long then
9:       Store the result in bytes_to_store
10:    else
11:      Return -1
12:    end if
13:  end if
14:  Generate a random number and store it in rnd
15:  Initialize i to rnd
16:  for i to 65535 do
17:    if SymmCryptomaterial referenced by i is uninitialized then
18:      Initialize it
19:      Store the symmetric key, bytes_to_store
20:      Store the algorithm
21:      Return i
22:    end if
23:    Increment i
24:  end for
25:  Initialize i to rnd
26:  for i to 0 do
27:    if SymmCryptomaterial referenced by i is uninitialized then
28:      Initialize it
29:      Store the symmetric key, bytes_to_store
30:      Store the algorithm
31:      Return i
32:    end if
33:    Decrement i
34:  end for
35:  Return -1
36: end procedure
```

```
[ 30.200954] SPDU:
[ 30.219911] 03 81 00 40 03 80 44 F5 31 46 6E 45 0E 93 22 95 45 1C 03 12 3F 6F F6 81 36 9A 78 B4 7D 67 2A 56
[ 30.264558] 81 78 ED D2 E0 B1 97 8A 1B AE 68 9E 0F 8C B7 7D 45 1E 72 FE 4F DD CF DF 98 92 BB 9F D9 80 3B 9C
[ 30.309220] 05 8D C5 54 4B A2 9D 03 80 01 44 40 01 04 FF FF EB 9E 2F 95 DC C0 82 00 80 80 95 AF 89 C6 73 DC
[ 30.353918] 31 E5 ED 74 B5 7B C3 F7 5C 25 04 74 5B 67 6E 7A 40 2F 98 5D 64 86 69 35 A1 6A BB F1 CC 51 1B 00
[ 30.398687] EA 19 76 97 F0 0B FF FA 64 9B 32 8F 79 05 2D 53 3F D4 2C 85 32 2D 6D 6C F2 E8
```

Figure 3.7: A self-signed data structure containing the signature generated from the included raw message

Algorithm 16 Symmetric cryptomaterial ID generation

```
1: procedure SEC_SYMMETRICCRYPTOMATERIALHANDLE_HASHEDID8
2:   if algorithm of the SymmCryptomaterial referenced by scmh is not aes128Ccm
   then
3:     Return an empty ret_hash
4:   end if
5:   if SymmCryptomaterial referenced by scmh is initialized then
6:     Calculate the HashedId8 and store it in ret_hash
7:   end if
8:   Return ret_hash
9: end procedure
```

public key, while only allowing the sender to sign a message that can be verified with the key. This results in authentication of the sender, with the receiver being able to trust that the message is from the sender that it says its from, and that the message was not modified during transmission. As the signature generation and verification processes rely on recomputing the message hash, failed verifications are a result of a compromised message. Before a message is sent out, the SDEE may request that the SDS signs some data payload by calling the *Sec_SignedData(int cmh, Opaque data, DataType data_type, Opaque ext_data_hash, HashAlgorithm hash_algorithm, int psid, boolean set_gen_time, boolean set_gen_loc, Time64 expiry_time, SignerIdentifierType signer_id_type, int signer_id_cert_chain_length, int max_cert_chain_length, FastVerificationType verification_type, ECPointFormat ec_pt_format, boolean use_p2pcd, int sdee_id)* primitive. Upon a successful signature, the primitive will return an *Opaque* containing an encoded *Ieee1609Dot2Data*, see Figure 3.6 for an SPDU containing an already encoded *Ieee1609Dot2Data* or Figure 3.7 for an SPDU containing a raw payload. A failed signature operation will return an *Opaque* of length 0.

The *cmh* is expected to be in the range [0, 65535] and references a *Cryptomaterial* in the *CMHS_key_pair* state. The *data* to be signed is optional if *ext_data_hash* is present, otherwise this field is required and is expected to be an initialized byte array. If *data* is present, the *data_type* specifies the type of *data* that is passed in. If *data* is not present,

then *ext_data_hash* is required and is expected to be a 32 byte array containing the hash of some external data to be signed. A *hash_algorithm* is present corresponding the hash algorithm used when generating *ext_data_hash*, which in this implementation must be *sha256*. The SDEE provides the PSID for the associated data payload with *psid*. Flags for including a generation time and location are set with *set_gen_time* and *set_gen_loc* respectively. If *expiry_time* is provided, it must be later than the current generation time. In this implementation only a *signer_id_type* of *signer_self_type* is supported. In this implementation, the *signer_id_cert_chain_length* and *max_cert_chain_length* are always 0. Fast verification is unsupported in this implementation forcing *verification_type* to always be 0. All elliptic curve points are uncompressed, so *ec_pt_format* is set to *uncompressed_format*. Peer to peer cert distribution is unsupported forcing *use_p2pcd* to false. The *sdee_id* is only present if *use_p2pcd* is true, forcing the *sdee_id* to 0. For self signed PDUs, this implementation prepends the *data* with the public key to be used for verifying the signature. The algorithm for the *Sec_SignedData* primitive is Algorithm 17.

3.10 Verifying Ieee1609Dot2Data

Upon receipt of a signed *Ieee1609Dot2Data*, an SDEE may verify the signature to determine if the message has been tampered with and/or that the signatory is who they say they are. A primitive has been defined to determine the validity of the signature with a call to *Sec_SignedDataVerification(int sdee, int psid, ContentType type, Ieee1609Dot2Data signed_data, Opaque ext_hash, HashAlgorithm hash_algorithm, int max_cert_chain_length, boolean replay_relevance, boolean generation_past_relevance, ValidityPeriod validity_period, boolean generation_future_relevance, ValidityPeriod future_period, Time64 generation_time, boolean expiry_relevance, Time64 expiry_time, boolean generation_location_consistency, boolean generation_location_relevance, int distance, ThreeDLocation generation_location, ValidityPeriod overdue_tolerance, boolean expired_certificate_relevance)*. This primitive will return true or false depending on

Algorithm 17 Sec-SAP Signed data generation

```
1: procedure SEC_SIGNEDDATA
2:   Initialize an Opaque, ret.Opaque to 0
3:   Create an Ieee1609Dot2Data, ret_data
4:   if cmh is not in the range [0, 65535] then
5:     Return ret.Opaque
6:   end if
7:   if data and ext_data_hash are empty then
8:     Return ret.Opaque
9:   end if
10:  if sdee_id is not 0 then
11:    Return ret.Opaque
12:  end if
13:  if data_type is Raw_type then
14:    Attempt to generate an Ieee1609Dot2Data wrapper to house the data
15:    if generation failed then
16:      Return ret.Opaque
17:    end if
18:  else if data_type is not Ieee1609Dot2Data_type then
19:    Return ret.Opaque
20:  end if
21:  if signer_id_type is not signer_self_type then
22:    Return ret.Opaque
23:  else
24:    if Cryptomaterial referenced by cmh is in the key pair state then
25:      Convert the public key stored at cmh to a 64 byte array, public
26:      Copy public to key_data
27:      Append data to key_data
28:      Determine the curve to be used with the signature generation
29:      if algorithm referenced by the cmh is ecdsaBrainpoolP256r1WithSha256
then
30:        Set the curve to brainpoolP256r1 [24]
31:      else if algorithm referenced by the cmh is ecdsaNistP256WithSha256 then
32:        Set the curve to NIST P-256 [22]
33:      else
34:        Return ret.Opaque
35:      end if
36:      Set the to be signed data payload of ret_data to key_data
37:      Set a flag in ret_data indicating the type of data it contains (data,
ext_data_hash or both)
38:      Compute the hash of the to be signed data payload
39:      Fill in the security headers
40:      if set_gen_time is true then
41:        Store the current time
```

```

42:         end if
43:         if set_gen_loc is true then
44:             Store the current location
45:         end if
46:         Convert the private key stored at cmh to a 32 byte array, private
47:         Compute the signature for hash with the given private on the curve and
store it in signature
48:         Set the signature type of ret_data to the corresponding curve
49:         Copy the first 32 bytes of signature into the r component of ret_data
50:         Copy the last 32 bytes of signature into the s component of ret_data
51:         Encode ret_data into ret_Opaque
52:     else
53:         Return ret_Opaque
54:     end if
55: end if
56: Return ret_Opaque
57: end procedure

```

whether the signature could be verified. During the verification of the PDU, relevancy tests will be performed, see Figure 3.8.

This implementation is constrained to supporting only 1 SDEE, forcing *sdee* to be 0. The *psid* is the PSID that is derived from the context in which the message is received, as specified in [10]. This primitive is required to have a content *type* of the signed data, however no specification of what this is, or how it is used, is defined. It appears to be left over from [6] and was forgotten about in the newer revision, therefore in this implementation, the value passed in *type* is meaningless. The *signed_data* to verify, must be of a valid *Ieee1609Dot2Data* type, indicating that it is signed. If the *signed_data* was created using an external data hash, that same hash is to be provided in an initialized 32 byte array, *ext_hash*. The corresponding hash algorithm, *hash*, is set if *ext_hash* is not empty, with the only supported algorithm being *sha256*. As certificates are not supported, *max_cert_chain_length* and *replay_relevance* will be 0 and false, respectively.

To request the SDS to reject an old SPDU, the flag *generation_past_relevance* will be true and the *validity_period* specifying how old is too old, will be set. To request that the SDS rejects an SPDU generated in the future, the flag *generation_future_relevance* will be

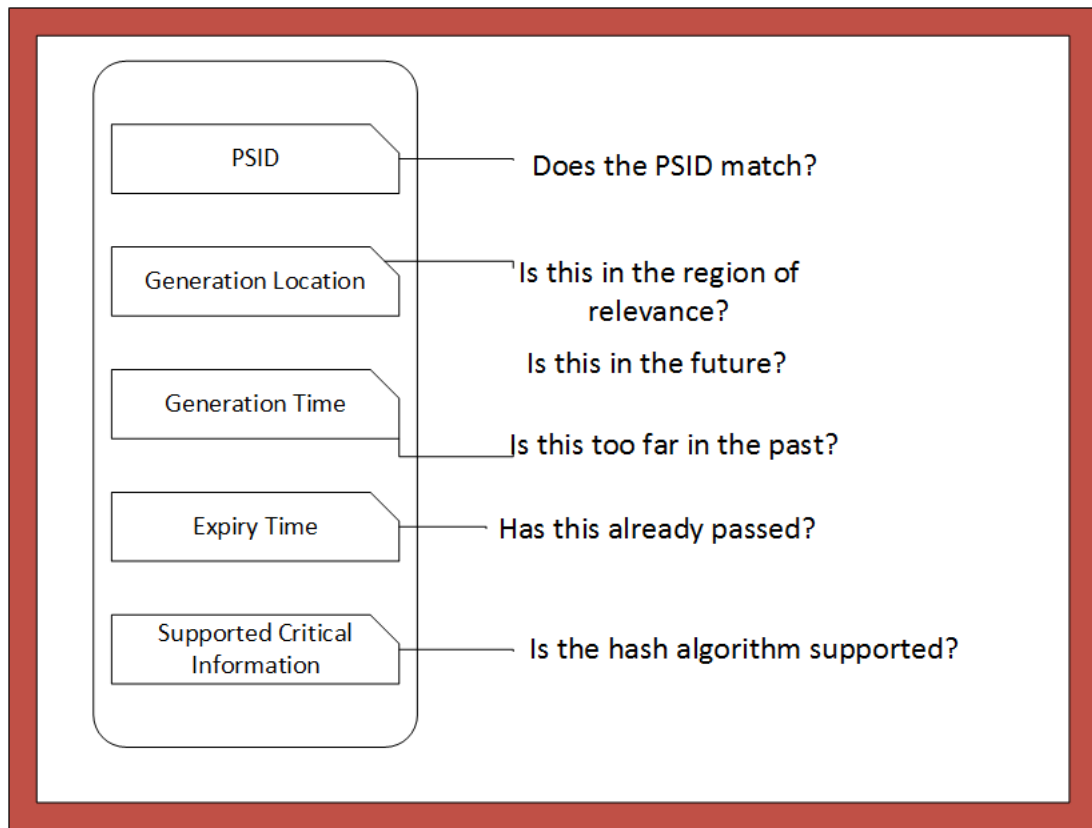


Figure 3.8: Relevance tests for signed data [10]

set along with a *future_period* defining how far out in the future to reject. If a generation time is not included within the security headers, one must be provided in *generation_time*. To request that the SDS rejects an expired SPDU, the *expiry_relevance* flag is set true. If the security headers do not contain an expiry time, one must be provided with *expiry_time*. To check if the SPDU was generated within the validity region of the certificate, the *generation_location_consistency* flag is true. Relevance checks will be performed if the *generation_location_relevance* flag is set along with a relevance *distance* defining the maximum distance between the recipient and generation location. If the security headers do not contain a generation location, one must be provided with *generation_location* initialized to a *ThreeDLocation*. In this implementation the *overdue_tolerance* and *expired_certificate_relevance* have no functional impact. The algorithm for the *Sec_SignedDataVerification* primitive is Algorithm 18.

3.11 Encrypting Ieee1609Dot2Data

If data is to be transmitted across an open medium, and only the intended recipient is supposed to be able to read the message, the message must be encrypted. Prior to sending out the message, an SDEE may request that the SDS encrypts a message using AES-CCM [15]. The primitive defined to encrypt an *Ieee1609Dot2Data* is *Sec_EncryptedData*(*Opaque data*, *DataType data_type*, *EncryptionType encryption_type*, *int *scmh*, *SequenceOfCertificate *certs*, *uint8_t *signed_recipient_info*, *HashedId8 pk_recip*, *ECPointFormat format*). Upon completion of this primitive call, either the encoded *Ieee1609Dot2Data* will be returned on a successful encryption or an empty *Opaque* will be returned.

The *data* parameter is the data to be encrypted, which is either an already encoded *Ieee1609Dot2Data* or a raw message. The type of message to encrypt is defined by *data_type*. Whether the message is to be encrypted using a static key or an ephemeral key, is defined by *encryption_type*. If *encryption_type* is static, then *scmh* will contain a single

Algorithm 18 Sec-SAP Signed data verification

```
1: procedure SEC_SIGNEDDATAVERIFICATION
2:   if sdee is not 0 then
3:     Return -1
4:   end if
5:   if signed_data is not of type signedData then
6:     Return invalid_input
7:   end if
8:   if signer type of signed_data is not signer_self_type then
9:     Return -1
10:  end if
11:  if hash algorithm used is not sha256 then
12:    Return unsupported_critical
13:  end if
14:  if security headers contain a generation time and generation_time is present then
15:    Return inconsisent_input
16:  end if
17:  if security headers do not contain a generation time and generation_time is not
    present then
18:    Return generation_time_not_available
19:  end if
20:  if expiry_relevance is true then
21:    if expiry_time is before the generation time then
22:      Return expiry_before_generation
23:    end if
24:    if security headers contain an expiry time and expiry_time is present then
25:      Return inconsitent_input
26:    end if
27:  end if
28:  if generation_location_relevance is true then
29:    if security headers contain a generation location and generation_location is
    present then
30:      Return inconsistent_input
31:    end if
32:    if security headers do not contain a generation location and generation_location
    is not present then
33:      Return generation_location_not_available
34:    end if
35:  end if
36:  if generation_location_consistency is true and no generation location is present in
    the security headers and generation_location is not present then
37:    Return generation_location_not_available
38:  end if
39:  if PSID in the security headers is not equal to psid then
```

```

40:      Return psid_no_match
41:  end if
42:  if to be signed data payload contains an external data hash then
43:      if ext_hash is present then
44:          if ext_hash does not match the external data hash then
45:              Return no_extdatahash_match
46:          end if
47:      end if
48:  else
49:      Return no_extdatahash_provided
50:  end if
51:  if external data hash is not present and ext_hash is present then
52:      Return no_extdatahash_present
53:  end if
54:  if no signed data payload is present then
55:      Return unsupported_critical
56:  end if
57:  Compute the SHA-256 hash [32] of the to be signed data payload and store it in
    hash
58:  Copy the first 64 bytes of the to be signed data payload into public
59:  Determine the curve to be used
60:  if signature type is ecdsaNistP256_type then
61:      Set curve to NIST P-256 [22]
62:  else if signature type is ecdsaBrainpoolP256r1_type then
63:      Set curve to brainpoolP256r1 [24]
64:  else
65:      Return unsupported_critical
66:  end if
67:  Store the r component of the signature into the first 32 bytes of sig
68:  Store the s component of the signature into the last 32 bytes of sig
69:  Verify the signature using the public key, public, the signature, sig, the hash and the
    curve
70:  if uECC_verify returns true then
71:      Return success
72:  end if
73:  Return -1
74: end procedure

```

SymmCryptomaterial handle, otherwise it may or may not be present. As certificates are unsupported in this implementation, *certs* will be expected to be *NULL*. The *signed_recipient_info* field is also expected to be *NULL* as this implementation does not support encryption keys in the security headers. The encryption key to be used can be derived from *pk_recip*, which is the *HashedId8* of the symmetric key. The only supported elliptic curve point format in this implementation is uncompressed, so it is assumed that all curve points will be of uncompressed. With this in mind, *format* can be of any value and has no functional impact on this primitive. The algorithm for *Sec_EncryptedData* is Algorithm 19.

3.12 Decrypting Ieee1609Dot2Data

Upon receipt of an encrypted *Ieee1609Dot2Data*, the SDEE must request the SDS to decrypt it, before any processing can be done. This implementation supports any number of recipients and will attempt to decrypt it using the recipients *Cryptomaterial*. The primitive defined for decryption is *Sec_EncryptedDataDecryption(Opaque data, int cmh, _octet1 signed_recip_info)* and upon successful decryption, will return an encoded, decrypted *Ieee1609Dot2Data* or an empty *Opaque* upon failure. The *Ieee1609Dot2Data* that is to be decrypted will be passed in as *data* and is expected to be properly encoded. The *cmh* is a handle to either a *Cryptomaterial* or *SymmCryptomaterial*, depending on whether the data was encrypted ephemeraly for the former, or statically for the latter. Retrieving a key from a signed PDU is unsupported in this standard, therefore *signed_recip_info* is optional and can be any value. The algorithm for this primitive is Algorithm 20.

Algorithm 19 Sec-SAP Signed data encryption

```
1: procedure SEC_ENCRYPTEDDATA
2:   Initialize an Opaque, ret_opaque to 0
3:   Initialize an Ieee1609Dot2Data, encrypted
4:   if data is empty then
5:     Return ret_opaque
6:   end if
7:   if data_type is Raw_type then
8:     Generate an Ieee1609Dot2Data wrapper to house the data and store it in
    working_data
9:   else if data_type is Ieee1609Dot2Data_type then
10:    Copy data into working_data
11:  else
12:    Return ret_opaque
13:  end if
14:  if encryption_type is static_encryption then
15:    if scmh contains more than one handle then
16:      Return ret_opaque
17:    end if
18:    if SymmCryptomaterial referenced by scmh is uninitialized then
19:      Return ret_opaque
20:    end if
21:    if SymmCryptomaterial referenced by scmh does not use aes128Ccm then
22:      Return ret_opaque
23:    else
24:      Copy the symmetric key to aes_cipher
25:    end if
26:    Generate a random nonce and store it in nonce
27:    Encrypt working_data using AES-CCM [15] with aes_cipher and the nonce and
    store the result in cipher_text
28:    if encryption failed then
29:      Return ret_opaque
30:    end if
31:    Copy the nonce, cipher_text and their respective lengths into encrypted
32:    Set the recipient type of encrypted to pskRecip_type
33:    Store the HashedId8 of the symmetric key in encrypted
34:  end if
35:  if encryption_type is ephemeral_encryption then
36:    if pk_recip is not 8 bytes long then
37:      Return ret_opaque
38:    end if
```

```

39:      Look up the handle associated with pk_recip and store it in cmh
40:      Store the public key of the Cryptomaterial referenced by cmh to
      public_response
41:      Determine the curve type
42:      if algorithm of the Cryptomaterial referenced by cmh is eciesNistP256 or
      ecdsaNistP256WithSha256 then
43:          Set curve to NIST P-256 [22]
44:          Set key_type to eciesNistP256_type
45:      else if algorithm of the Cryptomaterial referenced by cmh is
      eciesBrainpoolP256r1 or ecdsaBrainpoolP256r1WithSha256 then
46:          Set the curve to brainpoolP256r1 [24]
47:          Set the key_type to eciesBrainpoolP256r1_type
48:      else
49:          Return ret_opaque
50:      end if
51:      Generate a new SymmCryptomaterial handle and copy the symmetric key into
      aes_cipher
52:      Generate a random nonce and store it in nonce
53:      Encrypt working_data using AES-CCM [15] with aes_cipher and the nonce and
      store the result in cipher_text
54:      Copy the nonce, cipher_text and their respective lengths into encrypted
55:      Encrypt the symmetric key aes_cipher with the public_responses key on the
      curve using (3.7.4) and store the ephemeral key in rnd_public, the tag in tag1 and the
      encrypted key in encrypted_cipher
56:      Set the recipient type of encrypted to rekRecip_type
57:      Store the HashedId8 of the public_response key in encrypted
58:      Store rnd_public, encrypted_cipher and tag1 in encrypted
59:      Store key_type in encrypted
60:      end if
61:      Encode encrypted into ret_opaque
62:      Return ret_opaque
63: end procedure

```

Algorithm 20 Sec-SAP Signed data decryption

```
1: procedure SEC_ENCRYPTEDDATADECRYPTION
2:   Initialize an Opaque, ret_opaque to 0
3:   if data is empty then
4:     Return ret_opaque
5:   end if
6:   Decode data into an Ieee1609Dot2Data, data_struct
7:   if decode failed then
8:     Return ret_opaque
9:   end if
10:  Determine what the cmh handle references
11:  if cmh references a Cryptomaterial that is initialized then
12:    if recipient type of data_struct is not rekRecip_type then
13:      Return ret_opaque
14:    end if
15:    Store the public key referenced by cmh in public_key
16:    Compute the HashedId8 of public_key and store it in key_hash
17:    for each of the recipients in data_struct do
18:      if recipient id of the current recipient is equal to key_hash then
19:        Break the loop
20:      else
21:        Look at the next recipient
22:      end if
23:    end for
24:    if no recipient id is found that matches key_hash then
25:      Return ret_opaque
26:    end if
27:    Determine the curve type
28:    if recipient encryption key choice is eciesNistP256_type then
29:      Set the curve to NIST P-256 [22]
30:    else if recipient encryption key choice is eciesBrainpoolP256r1_type then
31:      Set the curve to brainpoolP256r1 [24]
32:    else Return ret_opaque
33:    end if
34:    Store the ephemeral public key, tag and encrypted key from data_struct into
    public_key, tag1 and encrypted respectively
35:    Copy the private key referenced by cmh into private_key
36:    Decrypt the symmetric key encrypted using (3.7.5) with public_key and
    private_key on curve and store the resulting decrypted key in decrypted and the tag
    in tag2
37:    if tag1 does not equal tag2 then
38:      Return ret_opaque
39:    end if
```

```

40:      Decrypt the cipher text stored in data_struct using AES-CCM [15] with the
      nonce stored in data_struct and the symmetric key decrypted and store the result in
      plaintext
41:      if decrypt failed then
42:          Return ret_opaque
43:      end if
44:  else if cmh references a SymmCryptomaterial that is initialized then
45:      if recipient type is not pskRecip_type then
46:          Return ret_opaque
47:      end if
48:      Store the HashedId8 of the symmetric key referenced at cmh into key_hash
49:      for each of the recipients in data_struct do
50:          if recipient id of the current recipient is equal to key_hash then
51:              Break the loop
52:          else
53:              Look at the next recipient
54:          end if
55:      end for
56:      if no recipient id is found that matches key_hash then
57:          Return ret_opaque
58:      end if
59:      Decrypt the cipher text stored in data_struct using AES-CCM [15] with the
      nonce stored in data_struct and the symmetric key referenced at cmh and store the result
      in plaintext
60:      if decrypt failed then
61:          Return ret_opaque
62:      end if
63:  else
64:      Return ret_opaque
65:  end if
66:  Copy the decrypted plaintext into ret_opaque
67:  Return ret_opaque
68: end procedure

```

CHAPTER 4

PROTOCOL CONFORMANCE

The research presented in this implementation aims to conform to the 1609.2 protocol, as outlined in Annex A of [10]. The standard provides a PICS (protocol implementation conformance statement) proforma, that outlines the features specified in this standard and assigns a status to each feature. A status may be one of four types:

1. **C** $\langle n \rangle$ - Indicates a mutual conditionality is expected where only one of the features labeled with the same n is mandatory.
2. **O** $\langle n \rangle$ - Indicates that a mutual conditionality is expected where at least one of the items labeled with the same n is mandatory.
3. **M** - Indicates that the feature is mandatory.
4. **O** - Indicates that the feature is optional.

A clause for exceptions to the PICS proforma has been created, specifying that any exception created, results in a non conforming implementation. The implementation presented does have valid exceptions to the PICS proforma, however they are created under the assumption that the status of some items are incorrect, as a result of contradictory items. The assumption was made that higher level items with optional status, should not be mandatory in a lower sub item if it does not apply. An overview of these contradictions is defined below:

1. Item S1.2.2.5 states "Determine that certificate used to sign data is valid (part of a consistent chain, valid at the current time and location, hasn't been revoked)" and is marked as mandatory in order to meet S1.2.2 ("Create Ieee1609Dot2Data containing valid SignedData"). However, S1.2.2.3 requests that a signer identifier type used to sign must be at least one of: digest (S1.2.2.3.1), certificate (S1.2.2.3.2) or self (S1.2.2.3.3). If it's optional to support a signer of type certificate, then it is not possible for S1.2.2.5 to be mandatory, as implementations, such as this, can choose to not support signing with certificates.
2. Following item S1.2.2.5 are several items of status **O**(**n**) required for S1.2.2.5. This also contradicts with the optional status of S1.2.2.3.2.
3. Item S1.2.2.6 is mandatory to support S1.2.2, however this once again contradicts with the optional requirement to support S1.2.2.3.2.
4. Item S1.2.3 states "Create Ieee1609Dot2Data containing EncryptedData." Item S1.2.3.2 is mandatory for S1.2.3 and states "Maximum number of recipients supported," however to meet the requirements of this item, at least one of the recipient info types must be supported (PreSharedKeyRecipientInfo, symmRecipientInfo, certRecipientInfo, signedDataRecipientInfo or rekRecipientInfo), S1.2.3.3.1 to S1.2.3.3.4 respectively. As S1.2.3.3.2 is optional for support, it contradicts S1.2.3.4.1.5 and S1.2.3.4.6 which specify support for encryption using keys included in explicit and implicit certificates respectively.
5. Item S1.3.2.5 is similar to item S1.2.2.5 in that they are mandatory for the support of the verification and signature generation, respectively, of a signed SPDU and must check the validity of the certificate. S1.3.2.5 should be optionally supported as the signer identifier for certificates is optional. It would be very difficult for an implementation to validate a certificate, if it does not recognize that an SPDU was signed with a certificate.

6. Following item S1.3.2.5 are several items of status **O** \langle **n** \rangle that is the same case as mentioned for items following S1.2.2.5.
7. Additionally as part of the validity and consistency checks for a signed SPDU, there are several items marked **M** that relate to certificates that must be met in order to support the verification of a signed SPDU. As many of these checks do not correlate with self signed SPDUs, they should be optional or only mandatory for SPDUs signed with a certificate.

As it stands, there are quite a few contradictions relating to the conformance of an IEEE 1609.2 implementation. It is suggested that PICS proforma be considered for reorganization and restructuring to allow for a more consistent conformance. Much of the standard is left in a way requiring implementation specific interpretation. If the PICS proforma is left as is, valid implementations choosing to not implement optional, higher level features will be considered non conforming.

CHAPTER 5

TESTING AND VERIFICATION

This chapter will outline all tests conducted throughout this research including:

1. Various performance metrics
2. Verification of key generation against test vectors
3. Verification of the encryption and decryption of SPDUs
4. Verification that SPDUs are properly encoded and decoded

For both the NIST P-256 and brainpoolP256r1 curves, test vectors have been used to verify the accuracy of this implementation, [3][18]. The method for testing was the same for both elliptic curves. The curve domain parameters will be printed, followed by a private and public key pair supplied by the test vectors. Using the supplied private key, a new public key will be generated to show equivalency to the supplied public key. A string will be hashed, following a signature generation and then a verification on that signature, see Figures 5.1 and 5.3. Finally 5 key pairs will be generated, each signing and verifying the same string hash, in order to demonstrate randomness and the ability to generate fresh keys, see Figures 5.2 and 5.4. One thing to note, the NIST P-256 **a** parameter is actually negative 3; in this implementation **a** is represented as positive 3 in order to save on resources. All calculations using the NIST P-256 **a** parameter will be reversed, subtracted instead of added and added instead of subtracted, to accommodate the reverse representation.


```
[ 87.787105] Curve parameters:
[ 87.808576] p:
[ 87.819884] FF FF FF FF 00 00 00 01 00 00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF FF FF
[ 87.857989] n:
[ 87.875593] FF FF FF FF 00 00 00 00 FF FF FF FF FF FF FF BC E6 FA AD A7 17 9E 84 F3 B9 CA C2 FC 63 25 51
[ 87.913552] G:
[ 87.930777] 4F E3 42 E2 FE 1A 7F 9B 8E E7 EB 4A 7C 0F 9E 16 2B CE 33 57 6B 31 5E CE CB B6 40 68 37 BF 51 F5
[ 87.974654] 6B D1 D7 F2 E1 2C 42 47 F8 BC E6 E5 63 A4 40 F2 77 03 7D 81 2D EB 33 A0 F4 A1 39 45 D8 98 C2 96
[ 88.012496] a:
[ 88.029338] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ 88.066959] b:
[ 88.083455] 5A C6 35 D8 AA 3A 93 E7 B3 EB BD 55 76 98 86 BC 65 1D 06 B0 CC 53 B0 F6 B3 CE 3C 3E 27 D2 60 4B
[ 88.120908] Beginning ECDSA test....
[ 88.120917]
[ 88.120923] Test will load example NIST Private and Public keys for NIST P-256 curve.
[ 88.120936] A sample message will be hashed with SHA-256 and a signature generated.
[ 88.120949] A public key will be generated based off the NIST Private key.
[ 88.120961] This should be identical to the NIST Public key.
[ 88.120972] 5 key pairs will then be generated and signed to demonstrate randomness.
[ 88.120985]
[ 88.313060] NIST Private key:
[ 88.328066] 70 A1 2C 2D B1 68 45 ED 56 FF 68 CF C2 1A 47 2B 3F 04 D7 D6 85 1B F6 34 9F 2D 7D 5B 34 52 B3 8A
[ 88.365608] NIST Public key:
[ 88.386315] 81 01 EC E4 74 64 A6 EA D7 0C FE 9A 6E 2B D3 D8 86 91 A3 26 2D CB A4 F7 63 5E AF F2 66 80 A8
[ 88.429798] D8 A1 2B A6 1D 59 92 35 F6 7D 9C BA D5 8F 17 83 D3 CA 43 E7 8F 0A 5A BA A6 24 07 99 36 C0 C3 A9
[ 88.467386] string to hash: "This is only a test message. It is 48 bytes long"
[ 88.500240] message value:
[ 88.520701] 54 68 69 73 20 69 73 20 6F 6E 6C 79 20 61 20 74 65 73 74 20 6D 65 73 73 61 67 65 2E 20 49 74 20
[ 88.564508] 69 73 20 34 38 20 62 79 74 65 73 20 6C 6F 6E 67
[ 88.582689] size of msg: 48
[ 88.596776] hash value:
[ 88.616514] 00 A2 C8 7D EF 71 FD 5B A6 D8 92 FC F4 FC 1E 42 D2 A6 70 BF BF A0 AD 6D 71 A7 EC 67 E9 AB 13 6E
[ 88.654262] size of hash: 32
[ 88.719558] signature value:
[ 88.739645] 7B F2 10 1A 53 EF 82 9C AC 55 D4 A5 1C CF 07 7D 82 8A 69 18 FB 47 8F 9E DD 3A B7 40 E9 9F 13 47
[ 88.782762] 9D 86 01 46 C3 EC C3 09 00 7E A1 1D 81 61 9E FA A5 53 64 E2 FA 27 CA 7D 64 88 46 D1 C8 9B 0C D2
[ 88.869059] Calculated public key:
[ 88.891364] 81 01 EC E4 74 64 A6 EA D7 0C FE 9A 6E 2B D3 D8 86 91 A3 26 2D CB A4 F7 63 5E AF F2 66 80 A8
[ 88.935038] D8 A1 2B A6 1D 59 92 35 F6 7D 9C BA D5 8F 17 83 D3 CA 43 E7 8F 0A 5A BA A6 24 07 99 36 C0 C3 A9
[ 89.031520] hash verified
```

Figure 5.1: Curve parameters and test vector verification for NIST P-256

89.045143]	Creating 5 private/public key pairs	
89.122988]	Private key 1:	89.122988]
89.142494]	3C 27 99 3C 86 CE 06 6D AF D7 51 8F D1 6A 84 B2 88 72 F1 88 64 56 3F CE 85 94 C8 0F DC 53 86 F7	89.142494]
89.179763]	Public key 1:	89.179763]
89.199005]	12 76 FB A7 0B 0F 81 1F 54 51 3A D2 BA A6 C3 B8 79 9A CA B8 B5 19 E6 AF 15 D0 02 46 63 4E 0D	89.199005]
89.241892]	2F BA 82 F5 89 1B F5 BA F4 6A 04 A4 64 45 13 C3 ED 9D 9A 08 DC F1 B8 13 BE 84 90 46 B1 CC EN 08	89.241892]
89.330381]	signature value:	89.330381]
89.350543]	B4 C7 FF D3 C3 8A FA 1A 54 63 83 C7 CE A3 84 AD B2 E2 2F 61 8F 5C 96 4 C4 59 00 9A 8B F5 78 B4 1B	89.350543]
89.393874]	6F 3E 40 69 DC 25 F3 5A 77 D0 0E C4 FD 2D 8D 04 DC 89 73 28 69 13 2A 30 5C C4 48 95 FA BB BB 5	89.393874]
89.487677]	hash verified	89.487677]
89.550095]	Private key 2:	89.550095]
89.570066]	66 08 F1 69 30 61 A9 8F 53 15 67 85 2A EC FE 15 8A B5 06 1B DF D8 6A 18 6F 43 6E BA 76 52 5F 3F	89.570066]
89.607515]	Public key 2:	89.607515]
89.627050]	55 8F BB CB 75 3F 57 FC DE 66 CC DF 8E 0B 1C 1B 0A D1 4A 6E 3C 64 C2 B9 48 ED 2A DE E0 AF 8A F9	89.627050]
89.678143]	80 F0 EE 31 CF A4 D2 48 29 BE 86 58 C2 61 8D 9F 86 E1 9D ED BC 22 7D 69 50 06 5E 41 40 78 42 1A	89.678143]
89.758631]	signature value:	89.758631]
89.778891]	70 8A AE 09 62 EC 06 EB 8B 0B EF 90 F5 15 C8 43 73 F6 93 7B 06 9A 22 8E 65 02 43 CE B9 1F 2B BD	89.778891]
89.822706]	54 A0 CA 56 57 3F FB 28 63 8D 01 3C 07 B1 E6 A6 53 57 20 1E D2 EF 8F 54 B0 08 3A 34 C1 26 9D 0D D3	89.822706]
89.917789]	hash verified	89.917789]
89.980359]	Private key 3:	89.980359]
90.000739]	4D DC C5 0B AD 42 ED 1A 40 1E 49 C6 4C 50 DC 13 C0 36 71 D0 1B AA 1F 9C F9 B3 81 55 2B 1A 28 12	90.000739]
90.038377]	Public key 3:	90.038377]
90.058103]	99 7E 5D 80 74 E1 91 E0 07 16 27 A6 AD 0E 1B E1 97 31 A2 E5 3F 05 B6 58 C9 4E E9 7E BE F9 A6	90.058103]
90.101675]	57 1B 87 9F 04 6B 2B 03 05 BC F0 A7 2D 53 7A A2 32 73 52 7D 50 AD 9A 54 8B 73 A5 36 F5 12 61 8E 22	90.101675]
90.190474]	signature value:	90.190474]
90.210689]	E8 FB 64 81 27 92 17 0E 11 0A 58 F7 EF 7C 2B 48 9B 02 56 50 6D 59 2B C1 56 AF AE CA F6 FD 2E 16	90.210689]
90.254430]	02 60 BF 4C 0B 68 E9 89 68 E7 D7 14 E7 91 21 8F 0E 09 4A FE DD 9A 32 65 CD B4 A5 DB 94 80 C2 65	90.254430]
90.349049]	hash verified	90.349049]
90.411472]	Private key 4:	90.411472]
90.431849]	AF 76 6D FA 9D 18 A5 24 D4 72 8A D8 C4 87 44 EE CD 39 06 AF FC 1B D1 91 00 91 77 36 F4 DF 49	90.431849]
90.469434]	Public key 4:	90.469434]
90.489163]	1C 0E 4E F6 ED 16 E5 CA 5A 2F 5C CC 09 85 16 BC 3D E9 44 BE 92 0B 28 F4 29 89 A7 6A 9F 71 2A E	90.489163]
90.532714]	9E D5 E6 CD 81 D5 E6 9D 9D 54 6B 9A A0 99 75 A4 7F D5 21 09 33 41 9B AC 97 39 69 7C A0 70 5D C0	90.532714]
90.621371]	signature value:	90.621371]
90.641655]	F7 FA B1 40 FC 32 93 01 24 FC A1 44 07 DC A9 6E 4C 33 35 19 EB AE B6 AE 49 6A 08 74 C2 0B B3 44	90.641655]
90.685423]	96 96 D4 70 86 B5 56 DF BA 06 C2 31 F0 0B B5 74 C4 8A 5E 68 51 6B C2 C8 F6 0C D9 9F FF D8 B6 AC	90.685423]
90.780211]	hash verified	90.780211]
90.842643]	Private key 5:	90.842643]
90.863048]	DB 32 59 57 04 80 9C 88 EA 3B A0 65 19 59 64 B3 22 8A 80 85 AA BB 64 77 26 09 7A E0 AB 5E 9F 2E	90.863048]
90.900765]	Public key 5:	90.900765]
90.920514]	83 AB 97 CB 14 1C 10 34 62 3B 02 F4 76 06 25 82 84 52 BC 5F 67 BE 4C 65 9F C2 80 27 90 77 E2 1D	90.920514]
90.964081]	EE CA 8B 15 F4 CA E8 19 46 78 B3 62 3C 93 A0 2C DF 98 77 54 F2 69 AC B7 CF 22 10 25 EC 81 59 81	90.964081]
91.052781]	signature value:	91.052781]
91.073039]	1A 21 29 A7 48 51 AD B7 61 9A E2 03 07 64 A2 C4 A3 AE CC 7E 06 CA 10 F7 89 C6 01 D0 06 1F 9B 45	91.073039]
91.116774]	01 A7 77 32 89 67 99 B7 B7 2C 71 90 3B B5 AA 46 E4 AF A9 3D 08 F2 71 4A 55 75 13 F3 C9 A9 EC 26	91.116774]
91.210423]	hash verified	91.210423]

Figure 5.2: 5 freshly generated key pairs for NIST P-256

```

[ 47.012260] Curve parameters:
[ 47.033762] p:
[ 47.045093] A9 FB 57 D8 A1 EE A9 BC 3E 66 0A 90 9D 83 8D 72 6E 3B F6 23 D5 26 20 28 20 13 48 1D 1F 6E 53 77
[ 47.083194] n:
[ 47.100808] A9 FB 57 D8 A1 EE A9 BC 3E 66 0A 90 9D 83 8D 71 8C 39 7A A3 B5 61 A6 F7 90 1E 0E 82 97 48 56 A7
[ 47.138783] G:
[ 47.156050] 54 7E F8 35 C3 DA C4 FD 97 F8 46 1A 14 61 1D C9 C2 77 45 13 2D ED 8E 54 5C 1D 54 C7 2F 04 69 97
[ 47.199877] 8B D2 AE B9 CB 7E 57 CB 2C 4B 48 2F FC 81 B7 AF B9 DE 27 E1 E3 BD 23 C2 3A 44 53 BD 9A CE 32 62
[ 47.237675] a:
[ 47.254558] 7D 5A 09 75 FC 2C 30 57 EE F6 75 30 41 7A FF E7 FB 80 55 C1 26 DC 5C 6C E9 4A 4B 44 F3 30 B5 D9
[ 47.292220] b:
[ 47.308762] 26 DC 5C 6C E9 4A 4B 44 F3 30 B5 D9 BB D7 7C BF 95 84 16 29 5C F7 E1 CE 6B CC DC 18 FF 8C 07 B6
[ 47.346215] Beginning brainpool test....
[ 47.346224]
[ 47.346229] Test will calculate Private and Public keys for the brainpoolP256r1 curve.
[ 47.346243] A sample message will be hashed with SHA-256 and a signature generated.
[ 47.346256]
[ 47.346262] 5 key pairs will then be generated and signed to demonstrate randomness.
[ 47.346275]
[ 47.499883] string to hash: "This is only a test message. It is 48 bytes long"
[ 47.526626] message value:
[ 47.546524] 54 68 69 73 20 69 73 20 6F 6E 6C 79 20 61 20 74 65 73 74 20 6D 65 73 73 61 67 65 2E 20 49 74 20
[ 47.590031] 69 73 20 34 38 20 62 79 74 65 73 20 6C 6F 6E 67
[ 47.608204] size of msg: 48
[ 47.622010] hash value:
[ 47.640784] 00 A2 C8 7D EF 71 FD 5B A6 D8 92 FC F4 FC 1E 42 D2 A6 70 BF BF A0 AD 6D 71 A7 EC 67 E9 AB 13 6E
[ 47.678165] size of hash: 32
[ 47.692107] Brainpool Private key:
[ 47.713606] 04 1E B8 B1 E2 BC 68 1B CE 8E 39 96 3B 2E 9F C4 15 B0 52 83 31 3D D1 A8 BC C0 55 F1 1A E4 96 99
[ 47.750904] Brainpool Public key:
[ 47.772109] 78 02 84 96 B5 EC AA B3 C8 B6 C1 2E 45 DB 1E 02 C9 E4 D2 6B 41 13 BC 4F 01 5F 60 C5 CC C0 D2 06
[ 47.815163] A2 AE 17 62 A3 83 1C 1D 20 F0 3F 8D 1E 3C 0C 39 AF E6 F0 9B 4D 44 BB E8 0C D1 00 98 7B 05 F9 2B
[ 49.255433] Calculated public key:
[ 49.277088] 78 02 84 96 B5 EC AA B3 C8 B6 C1 2E 45 DB 1E 02 C9 E4 D2 6B 41 13 BC 4F 01 5F 60 C5 CC C0 D2 06
[ 49.320276] A2 AE 17 62 A3 83 1C 1D 20 F0 3F 8D 1E 3C 0C 39 AF E6 F0 9B 4D 44 BB E8 0C D1 00 98 7B 05 F9 2B
[ 52.165910] signature value:
[ 52.186308] 97 F2 87 2B E1 6E 96 27 F0 F7 AF BE E7 C2 24 CE 46 C9 56 96 41 F5 8A CC 10 F3 45 20 07 11 CC B3
[ 52.229866] 7D B6 86 4C 68 A8 5A D3 6F 0E 11 F7 B4 64 9A 1D C8 E4 1C 4B D6 4F 17 7C 7A 42 F9 90 5C 2C F8 3F
[ 54.086397] hash verified

```

Figure 5.3: Curve parameters and test vector verification for brainpoolP256r1

```

[ 54.099903] Creating 5 private/public key pairs
[ 55.531701] Private key 1:
[ 55.551647] 67 2E 7B D8 00 F1 3E 65 84 B9 3E CD 0E D4 A0 62 E3 B0 5C 84 C9 AC B2 AF B8 BB 77 7E CD 03 8B EC
[ 55.589101] Public key 1:
[ 55.608949] 19 A0 66 AF BB CE 7D 8D B0 83 09 3A 92 1A 4A C3 42 43 41 6D 54 05 04 7F FB E8 A1 36 35 A4 13 0C
[ 55.652516] A9 9B A3 EB F7 26 1C A9 E8 79 F0 04 BE F3 2A BB DE 21 5E 86 AC DD C3 70 AD BD 04 57 63 AB 57 98
[ 57.095438] signature value:
[ 57.116073] 4A FE A4 6C 4E 98 3B A6 55 21 9F 06 F6 75 D1 65 BE 96 E7 B8 A4 D1 63 55 B3 5D 26 66 96 3D D3 2A
[ 57.159612] 99 2F 8D 4B E8 20 AE 8E 63 0A 74 B6 20 52 5F 0C 2E A2 EA BE 78 5D 8E 33 18 08 B0 BD B3 3F 3D D8
[ 59.010259] hash verified
[ 60.426671] Private key 2:
[ 60.446614] 3D 48 A6 CB A8 F1 CC AB DF B3 C7 AD E1 D1 B5 D5 4A 7D 3E 53 9B 18 91 D5 7B 9F 42 3A 35 0D 10 03
[ 60.484122] Public key 2:
[ 60.503779] 26 16 52 E2 B7 DE B7 8E 7B 30 70 E5 BF 99 F9 05 46 41 19 B1 54 0A 38 43 FB 57 26 FB A8 48 A6 6E
[ 60.546875] 01 09 B4 5F 3D B4 6C 57 1E D0 45 D6 01 E3 38 69 CB 81 05 0D 8F 0A 2C 4A 39 DE F4 E5 16 CA 51 19
[ 61.989525] signature value:
[ 62.009843] 53 C6 27 2A 7C 80 78 B3 6F D5 6E 4F 0A EB 49 44 AB CA AE 45 70 FF 92 5D 4B D9 AB B9 35 17 33 5A
[ 62.053691] 4A 08 76 34 13 9A 84 27 FD 3A 79 41 F6 09 9E 7F EE ED 71 A1 51 20 F7 11 49 C3 57 22 56 CF 38 0C
[ 63.900592] hash verified
[ 65.317216] Private key 3:
[ 65.337730] 08 B1 6D 0C EA AC AD 3F 27 19 1D E2 1B B1 74 3A 35 4B D3 2D AB 0B 96 0F B6 CB 2C D8 48 23 B3 8E
[ 65.375406] Public key 3:
[ 65.395223] 22 9A 3F 13 18 A1 E6 C8 88 EA E5 87 23 03 45 82 6C 69 0C 89 A4 22 EB D5 61 08 22 BC 91 63 EF E9
[ 65.438843] 2D 7B BF B2 53 59 61 EA 7A 16 FB 46 A2 C2 DF 90 03 EA E4 D9 27 BD 8A BE 4B AA 87 C1 33 B7 2B C1
[ 66.881625] signature value:
[ 66.902023] 01 72 B9 71 1F 62 41 63 23 68 49 35 1F 4B DE F8 DE 19 06 32 E1 34 B6 80 76 A1 AA C5 0A AB EE BB
[ 66.945856] 8D 1D 44 8A EB 1A 8D E9 BA CA 61 43 78 3B B4 A6 69 69 D7 66 6F 41 A8 B1 0F 65 DC 5A 97 E1 6C 0B
[ 68.758172] hash verified
[ 70.174809] Private key 4:
[ 70.195315] 61 66 50 33 09 ED E8 76 3F 5E 1A 40 71 C7 EB 24 DC B1 FC 0D 70 A9 1F B9 BB B8 FF 86 67 BB 51 1B
[ 70.232990] Public key 4:
[ 70.252789] 41 18 AC 6F D0 4D 06 03 44 2B F0 78 A2 38 AA 70 C9 F7 C4 51 50 C7 E0 F7 3A BB 83 94 A8 D9 57 9A
[ 70.296425] 23 A5 96 22 2A 77 CB 28 DF C9 7B 68 26 E3 B5 8B 89 D9 68 6A 4D 66 2E 36 9D 33 0E 3B 50 04 7F 37
[ 71.739112] signature value:
[ 71.759417] 3A D9 0C AE 93 C0 25 8B 50 A6 DA BE 6C F9 39 5D 0F 7B 5B 11 01 BF 1A ED 22 7C 60 52 2D A1 F6 24
[ 71.803123] 3B B7 2B 2B 79 C0 21 BD 2A 13 22 90 C3 F6 94 B1 F9 EE 28 02 C1 B5 46 5E 05 E6 86 26 62 5B 96 F4
[ 73.632882] hash verified
[ 75.048970] Private key 5:
[ 75.069408] 29 CA 78 02 43 5E C9 70 94 E3 6A F5 82 24 7B AB 01 84 F3 65 37 30 5F A9 83 4B BA 5D C2 B3 A9 02
[ 75.107112] Public key 5:
[ 75.126931] 00 D8 14 7E 4F C2 00 FB 6D 9B 60 A8 58 09 96 4D 38 9C 48 72 A6 FE 75 94 12 4B B8 46 EB 32 5A FC
[ 75.170545] 06 87 E0 D2 47 E5 6D DE 56 E2 7A 1E 77 B0 D2 B2 27 8C 60 E2 22 82 74 65 8C BF A9 89 7E 75 C2 31
[ 76.612238] signature value:
[ 76.632618] 36 2F 36 FC 97 56 28 03 4E 46 BD 3B 12 AB F6 54 AC D7 46 B7 73 69 90 20 F0 C4 5C 80 E9 4A CC B7
[ 76.676371] 6B 12 08 C4 28 F2 D5 14 9C 76 B7 CC 80 8A EE 3D 9B DD 1F B5 3C 32 84 0B D4 BC E2 23 EB 82 BF 69
[ 78.525459] hash verified#

```

Figure 5.4: 5 freshly generated key pairs for brainpoolP256r1

```

curve = uECC_secp256r1();
uECC_make_key(receiver_public, receiver_private, curve);
printk(KERN_INFO "Beginning ECIES Encryption with NIST P-256...");
ECIES_EncryptSymmKey(public, encrypted, tag1, cipher, empty, receiver_public, curve);
printk(KERN_INFO "String to encrypt: ");
vli_print(cipher, 16);
printk(KERN_INFO "Encrypted string: ");
vli_print(encrypted, 16);
ECIES_DecryptSymmKey(public, encrypted, tag2, decrypted, empty, receiver_private, curve);
printk(KERN_INFO "Decrypted string: ");
vli_print(decrypted, 16);
printk(KERN_INFO "MAC from encryption: ");
vli_print(tag1, 16);
printk(KERN_INFO "MAC from decryption: ");
vli_print(tag2, 16);

```

Figure 5.5: The sample code used to verify encryption and decryption of a symmetric key using the NIST P-256 curve.

```

[ 70.260980] Beginning ECIES Encryption with NIST P-256...
[ 70.374686] String to encrypt:
[ 70.394373] E9 44 41 12 F6 E5 63 A6 03 9D 54 52 A5 22 16 5A
[ 70.412552] Encrypted string:
[ 70.431638] 19 A2 18 BF 5B 1F FD CC 92 53 12 2E BA 1C D5 2C
[ 70.498659] Decrypted string:
[ 70.517715] E9 44 41 12 F6 E5 63 A6 03 9D 54 52 A5 22 16 5A
[ 70.535859] MAC from encryption:
[ 70.555935] 9C 49 91 88 F9 DE A4 DC F1 7F 80 AF E8 21 9D 5E
[ 70.574102] MAC from decryption:
[ 70.594222] 9C 49 91 88 F9 DE A4 DC F1 7F 80 AF E8 21 9D 5E

```

Figure 5.6: The results of the encryption and decryption of a symmetric key using NIST P-256

The key pairs shown in this chapter shall not be used for any cryptographic operation, as the private key is exposed. The ECIES encryption and decryption of a symmetric key is able to be performed for both the NIST P-256 curve and the brainpoolP256r1 curve. The verification method and verification results are provided for each, see Figures 5.5, 5.6 and Figures 5.7, 5.8 for each curve respectively.

The other form of encryption provided in this implementation is symmetric encryption with AES-CCM, both ephemerally and statically. For purposes of testing, the static encryption approach assumes that an SDEE already has access to a symmetric key via a symmetric cryptomaterial handle (SCMH), as the standard does not specify a method for the exchanging of keys. In this test, the SDEE encrypting and decrypting is the same entity. Additionally, the SDEE already has access to a freshly signed SPDU. The SDEE

```

curve = uECC_brainpoolP256r1();
uECC_make_key(receiver_public, receiver_private, curve);
printk(KERN_INFO "Beginning ECIES Encryption with brainpoolP256r1...");
ECIES_EncryptSymmKey(public, encrypted, tag1, cipher, empty, receiver_public, curve);
printk(KERN_INFO "String to encrypt: ");
vli_print(cipher, 16);
printk(KERN_INFO "Encrypted string: ");
vli_print(encrypted, 16);
ECIES_DecryptSymmKey(public, encrypted, tag2, decrypted, empty, receiver_private, curve);
printk(KERN_INFO "Decrypted string: ");
vli_print(decrypted, 16);
printk(KERN_INFO "MAC from encryption: ");
vli_print(tag1, 16);
printk(KERN_INFO "MAC from decryption: ");
vli_print(tag2, 16);

```

Figure 5.7: The sample code used to verify encryption and decryption of a symmetric key using the brainpoolP256r1 curve.

```

[ 72.015252] Beginning ECIES Encryption with brainpoolP256r1...
[ 74.843567] String to encrypt:
[ 74.863395] E9 44 41 12 F6 E5 63 A6 03 9D 54 52 A5 22 16 5A
[ 74.881551] Encrypted string:
[ 74.901301] 67 3F 0E D7 9C 39 99 09 03 43 FA 1D 11 F0 0B 70
[ 76.322372] Decrypted string:
[ 76.342221] E9 44 41 12 F6 E5 63 A6 03 9D 54 52 A5 22 16 5A
[ 76.360381] MAC from encryption:
[ 76.380751] 5E 1A 65 C6 A4 0E 76 40 3B 9E C9 D2 A2 54 5B 66
[ 76.398881] MAC from decryption:
[ 76.418868] 5E 1A 65 C6 A4 0E 76 40 3B 9E C9 D2 A2 54 5B 66 #

```

Figure 5.8: The results of the encryption and decryption of a symmetric key using brainpoolP256r1

```

28.200518] SPDU value:
28.220414] 03 81 00 40 03 80 44 43 C4 9D C0 B9 5D 53 5D 74 38 6D 02 7D 4F FA F2 1B DE 23 6C 91 D3 E2 B1 2F
28.264490] B3 2E BF 66 D9 E6 AA E3 4F 58 22 E5 02 F9 9A 90 91 17 86 8F D9 F4 89 45 E3 FC 52 DF 45 C5 FE A6
28.308771] B6 C2 7A 75 9D 70 F8 03 80 01 44 40 01 04 FF FF EB 9F 12 52 4B C0 82 00 80 80 65 5A 54 C9 91 ED
28.352997] 67 A3 D5 FC DD DF 59 2B 30 71 6A CE A9 67 B2 BD 71 D7 C4 45 83 B5 95 38 9A 74 AE 98 31 4A 61 1E
28.397195] 30 5F DE 87 CB 4A B2 8B B1 85 42 33 9C 5A 24 2E D5 06 01 DA F3 6D 13 89 B5 4E
28.737333] Encrypted Ephemeral Ieee1609Dot2Data:
28.765184] 03 82 01 01 08 5F B5 2C 72 74 C0 75 00 00 08 FE 5B 20 99 75 8E 62 9B 72 77 F9 12 64 68 10 E9 BF
28.810515] B4 FB C2 1A 9A 4B 2F 40 9A 86 BB 89 21 F2 FB B3 66 36 67 95 51 5C 57 6B F6 DD 4E EA C6 08 B4 4A
28.855909] C2 1E 02 79 60 75 7E C0 53 AE 16 05 45 3E 9E 4F 0B 36 77 51 91 0B C6 41 17 03 BF 6B F9 1F 83 98
28.901305] D3 B4 7E ED B1 4F 3E 3E B6 5B 91 82 79 7F F5 80 B4 59 62 0C 5D 6A 69 C1 3F 01 8E 53 81 AA A3 2F
28.946860] 98 2E DE 68 66 FA AD D5 B3 69 B9 BB A5 16 04 34 0F 36 28 D7 45 6E C2 1E FB CA 49 A4 44 ED BA 59
28.993065] 46 95 5E CE D5 01 E9 22 2F 8F EC FD EA 92 97 91 23 98 DC BB 99 E3 7A 5F 9D 2F 41 42 3B 7C 13 7C
29.039987] 1E 0E 37 E7 E6 F4 CF D1 44 12 DA 0C 24 4D 36 A5 26 B7 A3 E0 3C 93 96 1E 40 9A FD 11 DA 46 27 83
29.087514] 55 CB 8F 29 DE 52 1E F0 7D 25 5A F1 AC 1D A5 90 B6 38 26 3D EE A4 14 D0 6D 9B 6E 90 0F CD 9A 77
29.135086] 22 4D 9F 82 40 5C 13 37 5F FB E2 6C 66 72 59 BE A8 7B 1F 87 A7 6F 07 E9 B2 C7 B4 C5 C8 D8 9F 1D
29.182629] D1 BC 1D D4 1A 62 95 E2
29.243363] Decrypted Ephemeral Ieee1609Dot2Data:
29.272770] 03 81 00 40 03 80 44 43 C4 9D C0 B9 5D 53 5D 74 38 6D 02 7D 4F FA F2 1B DE 23 6C 91 D3 E2 B1 2F
29.319738] B3 2E BF 66 D9 E6 AA E3 4F 58 22 E5 02 F9 9A 90 91 17 86 8F D9 F4 89 45 E3 FC 52 DF 45 C5 FE A6
29.366783] B6 C2 7A 75 9D 70 F8 03 80 01 44 40 01 04 FF FF EB 9F 12 52 4B C0 82 00 80 80 65 5A 54 C9 91 ED
29.414024] 67 A3 D5 FC DD DF 59 2B 30 71 6A CE A9 67 B2 BD 71 D7 C4 45 83 B5 95 38 9A 74 AE 98 31 4A 61 1E
29.461258] 30 5F DE 87 CB 4A B2 8B B1 85 42 33 9C 5A 24 2E D5 06 01 DA F3 6D 13 89 B5 4E

```

Figure 5.9: The results of a sample ephemeral encryption and decryption

first makes a call to the *Sec_EncryptedData* primitive specifying a static encryption and passing in the signed SPDU and the SCMh for it's symmetric key. The SDS will return a statically encrypted SPDU containing the previously signed SPDU. Upon receipt of the encrypted SPDU, the SDEE makes a call to the *Sec_EncryptedDataDecryption* primitive, passing in the received SPDU and the associated SCMh. The signed SPDU, encrypted SPDU and decrypted SPDU are all displayed in Figure 5.10.

Similarly the ephemeral encryption test uses the same SDEE, for encryption and decryption, for the purpose of demonstration. In this test, the SDEE generates a freshly signed SPDU and a new cryptomaterial handle (CMH). The SDEE then generates a public and private key pair using NIST P-256 and stores the *HashedId8* of the public key. The SDEE then instructs the SDS to ephemerally encrypt its data, with a call to the *Sec_EncryptedData* primitive. The signed SPDU, *HashedId8* of the public key, and the specification of ephemeral are passed in the primitive. The SDS returns an ephemerally encrypted SPDU containing the signed SPDU as requested. The SDEE then makes a call to *Sec_EncryptedDataDecryption* with the encrypted SPDU and its CMH as parameters. The SDS then returns the decrypted SPDU, which contains the now visible, signed SPDU. The signed SPDU, encrypted SPDU and decrypted SPDU are all displayed in Figure 5.9. IEEE 1609.2 [10] includes an examples section (Annex D), which contains a sample


```

[ 22.712926] SPDU value:
[ 22.733075] 03 81 00 40 03 80 44 C5 CC 5F AD A2 63 87 FC 6B 14 F6 72 BF 50 F4 75 CE 6D 29 4B F5 03 17 6A 9E
[ 22.777288] B6 FF 68 B5 7C F4 6B A0 90 51 3B 67 78 62 07 F1 0E 09 27 9B 42 20 91 B9 DC 35 4A 41 85 03 A9 6F
[ 22.821496] 1C 41 10 7B D9 BC E4 03 80 01 44 40 01 04 FF FF EB 9F 08 3F 8F 80 82 00 80 80 BF 25 B8 75 F9 5B
[ 22.865774] 39 6D 42 6F 98 52 08 71 D9 62 1E E8 8A 9C 16 18 C6 5C 51 BA 15 4B A0 F9 CE A0 9B 9E A0 CC B8 FB
[ 22.910135] 4A 91 0E 30 07 5D 0F 9B DD C1 AD 80 0B BA 74 ED B4 34 84 BB 94 61 14 2B 46 2C
[ 23.099329] Encrypted Static Ieee1609Dot2Data:
[ 23.125386] 03 82 01 01 00 1F BC 33 9D 95 23 DB 45 80 63 E8 26 B3 3D A8 75 7F 64 63 17 95 81 AA 0A 25 4E 16
[ 23.169400] BA 82 47 6F 7C 65 49 71 71 53 E2 20 91 FE 7E FF FC ED 5D 5B 2E 41 05 DD 82 09 E4 01 23 39 83 97
[ 23.213420] 86 F6 C8 D0 82 BE 98 B7 27 98 6B F6 CC DB 1B A3 6E 43 D4 7D 59 49 F7 58 9B E6 A5 98 DC 62 AC 99
[ 23.257434] AE BD 0A 18 1B C3 B0 DA 1B 77 1F 95 47 2E A3 17 A0 F7 32 64 63 EA 2C 28 9F DA FD 52 A1 7C 48 05
[ 23.301560] 50 11 A8 1F A9 7C 4E B2 92 35 B8 41 CA BD 7D DE B9 6B 41 3E 16 FA FB 6D 5F 21 5C 01 2A 2F C1 DE
[ 23.346357] 15 26 B2 3A 27 42 85 00 46 C7 86 39 D4 51 92 20 1F CF 4B AA BD 0D E4 EC FF A5 6B 14 84 9D 82 42
[ 23.391775] D1 60 60 40 20 2C
[ 23.401668] Decrypted Static Ieee1609Dot2Data:
[ 23.429043] 03 81 00 40 03 80 44 C5 CC 5F AD A2 63 87 FC 6B 14 F6 72 BF 50 F4 75 CE 6D 29 4B F5 03 17 6A 9E
[ 23.474424] B6 FF 68 B5 7C F4 6B A0 90 51 3B 67 78 62 07 F1 0E 09 27 9B 42 20 91 B9 DC 35 4A 41 85 03 A9 6F
[ 23.519814] 1C 41 10 7B D9 BC E4 03 80 01 44 40 01 04 FF FF EB 9F 08 3F 8F 80 82 00 80 80 BF 25 B8 75 F9 5B
[ 23.565209] 39 6D 42 6F 98 52 08 71 D9 62 1E E8 8A 9C 16 18 C6 5C 51 BA 15 4B A0 F9 CE A0 9B 9E A0 CC B8 FB
[ 23.610656] 4A 91 0E 30 07 5D 0F 9B DD C1 AD 80 0B BA 74 ED B4 34 84 BB 94 61 14 2B 46 2C

```

Figure 5.10: The results of a sample static encryption and decryption

```

[ 18.722778] Encoded structure:
[ 18.738026] 03 81 00 40 03 80 0F 54 68 69 73 20 69 73 20 61 20 42 53 4D 0D 0A 40 01 20 11 12 13 14 15 16 17
[ 18.781288] 18 80 21 22 23 24 25 26 27 28 80 82 31 32 33 34 35 36 37 38 31 32 33 34 35 36 37 38 31 32 33 34
[ 18.824530] 35 36 37 38 31 32 33 34 35 36 37 38 41 42 43 44 45 46 47 48 41 42 43 44 45 46 47 48 41 42 43 44
[ 18.867820] 45 46 47 48 41 42 43 44 45 46 47 48

```

Figure 5.11: The encoded structure of the decoded, encoded SPDU provided by [10]

COER (canonical octet encoding rules) encoding of a signed SPDU. In order to verify the ability for this implementation to encode and decode an SPDU, the given example *Ieee1609Dot2Data* is created, encoded, decoded and then encoded again. This can be seen in Figure 5.11.

NIST P-256	brainpoolP256r1
48.8	1402.7

Table 5.1: Time required to generate a key pair in milliseconds

NIST P-256	brainpoolP256r1
51.1	1404.9

Table 5.2: Time required to generate a signature in milliseconds

NIST P-256	brainpoolP256r1
57.1	1817.4

Table 5.3: Time required to verify a signature in milliseconds

For each of the performance metrics given, a timer was used to capture the time it takes for each operation to succeed, with an average value being calculated after 100

NIST P-256	brainpoolP256r1
97.7	2805.7

Table 5.4: Time required to encrypt a symmetric key using ECIES in milliseconds

NIST P-256	brainpoolP256r1
48.9	1402.9

Table 5.5: Time required to decrypt a symmetric key using ECIES in milliseconds

Encode	Decode
8	6

Table 5.6: Time required to encode and decode sample *Ieee1609Dot2Data* in microseconds

Raw	Ieee1609Dot2Data
52.2	51.3

Table 5.7: Time required to generate self signed SPDUs of either raw data or an already encoded *Ieee1609Dot2Data* in milliseconds

Static	Ephemeral
49.1	265.3

Table 5.8: Time required to both statically and ephemerally encrypt an SPDU in milliseconds

Static	Ephemeral
1.1	49.5

Table 5.9: Time required to both statically and ephemerally decrypt an SPDU in milliseconds

operations on the VMCD [11]. As shown by the Tables 5.1, 5.2, 5.3, 5.4, 5.5, the NIST P-256 curve operations are significantly faster than the brainpoolP256r1 curve. This is primarily attributed to the optimized modulus functions for NIST P-256 provided by [3], while the brainpoolP256r1 curve uses standardized elliptic curve modulus functions. The brainpoolP256r1, while significantly slower than NIST P-256, still has relatively low speeds for some operations. It is important to remember that these calculations were performed on an older platform with very low performance specifications. More modern hardware will be able to execute these operations with significant speed increases.

Performance for encoding and decoding was determined by calculating the total run time required to encode and decode the sample *Ieee1609Dot2Data* provided by [10]. As the results of these operations are very fast, see Table 5.6, the additional cost of performance is neglected for primitives who rely on this functionality. The performance associated with generating self signed SPDUs can be seen in Table 5.7. The performance associated with encrypting and decrypting static or ephemeral SPDUs can be seen in Tables 5.8 and 5.9. The SPDU that was passed into the encryption primitives, was an already created self signed SPDU. The increased time for ephemeral encryption is a result of the additional time required to search the *Cryptomaterial* storage for the *HashedId8* of the public key. One thing to note is that the 1609.2 standard allows for messages to be placed inside other messages. In rare cases, the time to complete one of these operations can continue to increase, as a result of the ability to recursively sign and/or encrypt messages inside other messages.

As messages in a VANET are constrained to a real time transmission rate, it is important for all messages to be sent, received and processed, as fast as possible. Having fast cryptographic algorithms will allow for a stronger communication link with a lower chance of missed messages. Even on the older platform, in which these performance metrics were obtained, a majority of the operations were under 100 milliseconds, with several being even faster than that. This allows for the security aspect of WAVE to have as

little impact as possible on transmission. As the hardware continues to advance, the operating time of these operations will continue to decrease, resulting in a more real time cryptographic processing of messages.

CHAPTER 6

CONCLUSION

As the concerns for security and privacy increase for our mobile devices, these same concerns will eventually come to being with our vehicles. It is important that before vehicles begin to publicly communicate with one another, these concerns must be addressed. The IEEE 1609.2 standard [10] aims to take care of this, by providing mechanisms to ensure the integrity, authentication, and security of messages. Vehicular communication by way of VANET will soon become an integral part of our society. In the U.S. law makers are currently drafting mandates for vehicle to vehicle communication (V2V) that could be finished as early as 2017 [29].

This thesis aims to augment the work of [23] for the Smart Drive VANET testbed [11]. An implementation of the IEEE 1609.2 standard is presented and serves to allow for additional future testing with the Smart Drive Initiative. The ability to have a standalone implementation of 1609.2 allows for development of more advanced VMCD prototypes, while still maintaining development on the WAVE protocol stack. Secure communication is now able to be conducted between VMCDs, allowing for additional performance tests of the IEEE 1609.4 [8] protocol as a result of the potential for increased packet length. This implementation provides the ability to sign and verify datagrams to demonstrate authenticity and message integrity. Mechanisms for the encryption and decryption, adhering to the current version of the standard have been implemented. Included in this implementation are various test cases and helper functionality to enhance development and prototyping of future revisions.

Test cases that address performance concerns for rapid communication have been given to alleviate worry and/or to demonstrate the need for more advanced hardware. The use of test vectors in the verification of key generation allows this implementation to show that it has been correctly implemented. The ability to correctly encrypt and decrypt a message, along with the capability of packing said message into an octet string allow for the reliability of secure data. The work presented in this research allows for the continual improvement and research of [11].

6.1 Future Work

As this implementation includes a subset of the optional features of the IEEE 1609.2 standard, there are many ways in which this implementation can be used to propel future research:

- Implementation of cryptographic operations utilizing certificates (signing, verifying, encrypting decrypting).
- Implementation of the certificate revocation list verification entity.
- Implementation of the peer to peer certificate distribution entity.
- Implementation of more efficient cryptographic operations.
- Implementation of a more secure cryptomaterial data storage, in which all data is encrypted as opposed to hidden.
- Implementation of compressed curve points for keys.
- Implementation of 1609.2 security profiles.
- Implementation of the SSME (security services management entity).
- Support for identification of multiple SDEEs (secure data exchange entities).
- Support for preprocessing of data and replay detection.

BIBLIOGRAPHY

- [1] ITU-T Study Group 17. Information technology specification of octet encoding rules (oer), august 2015. ITU-T Recommendations. URL: <http://handle.itu.int/11.1002/1000/12487>.
- [2] Information Assurance Directorate at the National Security Agency. Mathematical routines for the nist prime elliptic curves, 2010. URL: <https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/mathematical-routines-for-the-nist-prime-elliptic-curves.cfm>.
- [3] Information Assurance Directorate at the National Security Agency. Suite b implementers guide to fips 186-3, 2010. URL: <https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/suite-b-implementers-guide-to-fips-186-3-ecdsa.cfm>.
- [4] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls). RFC 4492, RFC Editor, May 2006. <http://www.rfc-editor.org/rfc/rfc4492.txt>. URL: <http://www.rfc-editor.org/rfc/rfc4492.txt>.
- [5] Buildroot. Making embedded linux easy, 2016. URL: <https://buildroot.org/>.
- [6] Intelligent Transportation Systems Committee. Ieee standard for wireless access in vehicular environments security services for applications and management messages. *IEEE Std 1609.2-2013 (Revision of IEEE Std 1609.2-2006)*, pages 1–289, April 2013.

- [7] Intelligent Transportation Systems Committee. Ieee guide for wireless access in vehicular environments (wave) - architecture. *IEEE Std 1609.0-2013*, pages 1–78, March 2014.
- [8] Intelligent Transportation Systems Committee. Ieee standard for wireless access in vehicular environments (wave) – multi-channel operation corrigendum 1: Miscellaneous corrections. *IEEE Std 1609.4-2010/Cor 1-2014 (Corrigendum to IEEE Std 1609.4-2010)*, pages 1–30, Dec 2014.
- [9] Intelligent Transportation Systems Committee. Ieee standard for wireless access in vehicular environments (wave) – network systems corrigendum 2: Miscellaneous corrections. *IEEE Std 1609.3-2010/Cor 2-2014 (Corrigendum to IEEE Std 1609.3-2010)*, pages 1–33, Dec 2014.
- [10] Intelligent Transportation Systems Committee. Ieee standard for wireless access in vehicular environments–security services for applications and management messages. *IEEE Std 1609.2-2016 (Revision of IEEE Std 1609.2-2013)*, pages 1–240, March 2016.
- [11] Smart Mobile Computing. Nsf mri grant, 2016. URL: <http://smc.eng.fau.edu/#/nsf-mri-grant/>.
- [12] Brad Conte. crypto-algorithms, 2015. URL: <https://github.com/B-Con/crypto-algorithms>.
- [13] Arzoo Dahiya and R. K. Chauhan. A comparative study of manet and vanet environment. *Journal of Computing*, pages 87–92, 2010.
- [14] Christophe Doche and Laurent Imbert. The double-base number system in elliptic curve cryptography, 2008. URL: http://www.lirmm.fr/~imbert/talks/laurent_Asilomar_08.pdf.

- [15] Morris J. Dworkin. Sp 800-38c. recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2004.
- [16] E. C. Eze, S. Zhang, and E. Liu. Vehicular ad hoc networks (vanets): Current state, challenges, potentials and way forward. In *Automation and Computing (ICAC), 2014 20th International Conference on*, pages 176–181, Sept 2014.
- [17] Carlos Gutierrez, James Turner, and Cita Furlani. Fips pub 198-1 federal information processing standards publication the keyed-hash message authentication code (hmac), July 2008. U.S.Department of Commerce/National Institute of Standards and Technology.
- [18] D. Harkins. Brainpool elliptic curves for the internet key exchange (ike) group description registry. RFC 6932, RFC Editor, May 2013.
- [19] H. Hasrouny, C. Bassil, A. E. Samhat, and A. Laouiti. Group-based authentication in v2v communications. In *Digital Information and Communication Technology and its Applications (DICTAP), 2015 Fifth International Conference on*, pages 173–177, April 2015.
- [20] Chanbok Jeong. Cryptography engine design for ieee 1609.2 wave secure vehicle communication using fpga. Master’s thesis, Ulsan National Institute of Science and Technology, december 2014.
- [21] Jung-Ha Kang, Sung-Jin Ok, Jae-Young Kim, and Eun-Gi Kim. Software implementation of wave security algorithms. *Journal of the Korea Academia-Industrial cooperation Society*, pages 1691–1699, 2014.
- [22] Cameron Kerry, Patrick Gallagher, and Charles Romine. Fips pub 186-4 federal information processing standards publication digital signature standard (dss), July 2013. U.S.Department of Commerce/National Institute of Standards and Technology.

- [23] Kyle Kuffermann. An implementation of the ieee 1609.4 wave standard for use in a vehicular networking testbed. Master's thesis, Florida Atlantic University, december 2014.
- [24] M. Lochter and J. Merkle. Elliptic curve cryptography (ecc) brainpool standard curves and curve generation. RFC 5639, RFC Editor, March 2010.
- [25] Kenneth MacKay. micro-ecc, 2014. URL: <https://github.com/kmackay/micro-ecc/>.
- [26] DI Management. Key derivation functions: How many kdfs are there?, 2016. URL: <http://www.di-mgt.com.au/cryptoKDFs.html>.
- [27] Microprocessor and Microcomputer Standards Committee. Ieee standard specifications for public-key cryptography - amendment 1: Additional techniques. *IEEE Std 1363a-2004 (Amendment to IEEE Std 1363-2000)*, pages 1–167, Sept 2004.
- [28] Rashmi Mishra, Akhilesh Singh, and Rakesh Kumar. Vanet security: Issues, challenges and solutions, 2016. International Conference on Electrical, Electronics, and Optimization Techniques. URL: <http://iceeot.org/papers/OR0389.pdf>.
- [29] Elvina Nawaguna. U.s. may mandate 'talking' cars by early 2017, 2014. Reuters. URL: <http://www.reuters.com/article/us-autos-technology-rules-idUSBREA1218M20140203>.
- [30] OpenSSL. Cryptography and ssl/tls toolkit, 2016. URL: <https://www.openssl.org>.
- [31] B. Pooja, M. M. M. Pai, R. M. Pai, N. Ajam, and J. Mouzna. Mitigation of insider and outsider dos attack against signature based authentication in vanets. In *Computer Aided System Engineering (APCASE), 2014 Asia-Pacific Conference on*, pages 152–157, Feb 2014.

- [32] Penny Pritzker, Willie E. May, and Charles H. Romine. Fips pub 198-1 federal information processing standards publication secure hash standard (shs), August 2015. U.S.Department of Commerce/National Institute of Standards and Technology.