CRYPTOGRAPHIC SECURITY AND KEY MANAGEMENT IN MULTI-PLATFORM IOT SYSTEMS: IMPLEMENTATION AND PERFORMANCE ASPECTS

by
ALPEREN DOĞAN

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
July 2025

CRYPTOGRAPHIC SECURITY AND KEY MANAGEMENT IN MULTI-PLATFORM IOT SYSTEMS: IMPLEMENTATION AND PERFORMANCE ASPECTS

Approved by:

Prof. ALBERT LEVİ ....
(Thesis Supervisor)

Asst. Prof. ORÇUN ÇETİN ....

Prof. VEDAT COŞKUN ....

Date of Approval: July 23, 2025

ABSTRACT

CRYPTOGRAPHIC SECURITY AND KEY MANAGEMENT IN MULTI-PLATFORM IOT SYSTEMS: IMPLEMENTATION AND PERFORMANCE ASPECTS

ALPEREN DOĞAN

Computer Science and Engineering, M.S. Thesis, May 2025

Thesis Supervisor: Prof. Albert Levi

Keywords: key management, elliptic curve cryptography, internet of things, trusted platform module, performance measurement

Smart-city services rely on many small sensors that send data about traffic, energy, and air quality. Each message must stay private and unaltered, yet the devices that send them have little memory, slow CPUs, and limited power. This thesis builds and tests a light but strong key-management system that fits these limits. The protocol uses elliptic curve certificates and elliptic curve Diffie–Hellman key exchange to set up a shared secret. Then, for every message between client and server, a symmetric key is derived from the shared secret. The shared secret, and elliptic curve keys on the server are renewed periodically to decrease the damage in case of compromise. The proposed protocol is formally verified using Tamarin Prover. Long-term keys on the server sit inside a TPM 2.0 chip, which adds an extra layer of security for storage of the keys. The protocol is deployed in a web-API, and performance tests on the protocol and individual cryptographic operations used by the protocol are carried out on a set of single-board computers representing generic IoT devices. The results show that by the proposed protocol, the key exchange operations complete in reasonable amounts of time where majority of the time is spent by the TPM.

ÖZET

ÇOK PLATFORMLU IOT SİSTEMLERİNDE KRİPTOGRAFİK GÜVENLİK VE ANAHTAR YÖNETİMİ: GERÇEKLEŞTİRME VE PERFORMANS HUSUSLARI

ALPEREN DOĞAN

Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans Tezi, Mayıs 2025

Tez Danışmanı: Prof. Dr. Albert Levi

Anahtar Kelimeler: anahtar yönetimi, eliptik eğri kriptografisi, nesnelerin interneti, güvenilir platform modülü, performans ölçümü

Akıllı-şehir hizmetleri, trafik, enerji ve hava kalitesi hakkında veri gönderen çok sayıda küçük sensöre dayanır. Gönderilen her mesaj gizli kalmalı ve değişikliğe uğramamalıdır; ancak bu cihazların belleği az, işlemcileri yavaş ve güçleri sınırlıdır. Bu tez, bu kısıtlamalara uyan hafif ama güçlü bir anahtar-yönetim sistemi tasarlar ve test eder. Protokol, ortak bir ana anahtar oluşturmak için eliptik eğri sertifikaları ve eliptik eğri Diffie–Hellman anahtar değişimini kullanır. Daha sonra, istemci ile sunucu arasındaki her mesaj için bu ortak sırdan bir simetrik anahtar türetilir. Ortak sır ile sunucudaki eliptik eğri anahtarları, olası bir ihlal durumunda zararı azaltmak amacıyla periyodik olarak yenilenir. Önerilen bu protokol, Tamarin kanıtlayıcısı kullanılarak biçimsel olarak doğrulanmıştır. Sunucudaki uzun vadeli anahtarlar, anahtar depolamasına ek bir güvenlik katmanı sağlayan TPM 2.0 yongası içinde tutulur. Protokol bir web API'sine yerleştirilmiştir. Protokolün kendisi ve kullandığı kriptografik işlemler, jenerik IoT cihazlarını temsil eden çeşitli tek kartlı bilgisayarlar üzerinde performans testlerine tabi tutulmuştur. Sonuçlar, önerilen protokolde, anahtar değişimi işlemlerinin makul bir sürede tamamlandığını, ve bu sürenin çoğunluğunun TPM tarafından harcandığını göstermektedir.

ACKNOWLEDGEMENTS

I want to start with thanking my supervisor Prof. Albert Levi, for his never ending support and guidance during my master's education. He did not only give me irreplaceable advice, but he also kept me motivated when I needed it most.

I would also like to thank my respected thesis jury members, Asst. Prof. Orçun Çetin and Prof. Vedat Coşkun for their time and their comments, advices and feedbacks.

I want to thank my colleagues and friends in the laboratory, for their support and encouragement all the way through my masters education. Their friendship has created a more comfortable working environment. I am also grateful for the friendship of many friends as they made me enjoy this journey, and kept me motivated.

Most of all, I want to thank my mother, Gülsen, and my siblings, Hüseyin, Lokman, Nursen, and Sultan, who has guided me and gave me all their love my entire life. Their emotional support during my masters education meant everything for me. I want to thank my father, Abdulvahap, for always believing in me, boosting my confidence, and making me want to do better.

This work was supported by Turkish Scientific and Technology Research Institution - Technology and Innovation Funding Programmes Directorate (TUBITAK - TEYDEB) under grant 3219500.

to my beloved father

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATONS

AES	Advanced Encryption Standard	xi, xii, xiii, 4, 5, 6, 15, 16, 17, 18, 23, 24, 25, 26, 31, 32, 33, 55
API	Application Programming Interface	3, 53
CBC	Cipher Block Chaining	xi, xii, xiii, 4, 7, 15, 16, 23, 24, 31, 55
CCM	Counter with Cipher Block Chaining Message Authentication Code	xi, xii, xiii, 5, 6, 15, 16, 17, 23, 25, 32
CSR	Certificate Signing Request	49
CSRF	Cross-Site Request Forgery	58
DER	Distinguished Encoding Rules	59
ECC	Elliptic Curve Cryptography	2, 67, 68
ECDH	Elliptic-curve Diffie-Hellman	2, 3, 7, 9, 14, 17, 25, 34, 39, 54, 60, 62
ECDSA	Elliptic-curve Digital Signature Algorithm	xi, xii, xiii, 3, 7, 8, 9, 18, 20, 25, 26, 28, 34, 36, 39, 44
ESAPI	Enhanced System Application Programming Interface	48
GCM	Galois Counter Mode	xi, xii, xiii, 5, 16, 18, 23, 26, 33
HKDF	Hash Based Key Derivation Function	10, 40
HMAC	Hash-based Message Authentication Code	xi, xii, xiii, xiv, 4, 5, 14, 19, 21, 22, 26, 29, 30, 34, 37, 38, 40, 41, 48, 49, 50, 54, 57, 58, 59, 60, 62
HTTP	Hypertext Transfer Protocol	57, 58, 59
IoT	Internet of Things	1, 2, 58, 67, 68
IV	Initialization Vector	4, 13, 14, 48, 55

MAC Message Authentication Code 4, 5, 7, 12, 14, 39

NIST National Institute of Standards and Technology 2, 11, 12, 13

PEM Privacy Enhanced Mail 59

RSA Rivest-Shamir-Adleman . . . . xiv, 48, 49, 50, 52, 55

SHA Secure Hash Algorithm xi, xii, xiii, xiv, 14, 19, 21, 22, 26, 27, 29, 30, 34, 35, 37, 38

TLS Transport Layer Security 42

TPM Trusted Platform Module xiv, 2, 3, 10, 39, 44, 45, 46, 47, 48, 49, 50, 51, 52, 55, 61, 63, 64, 65, 67, 68

U.S. FDA United States Food and Drug Administration 1

1. INTRODUCTION

Smart cities use connected sensors and devices, such as traffic cameras, smart meters, and air quality monitors, to improve public services, use resources more efficiently and lower operational costs (Zanella, Bui, Castellani, Vangelista & Zorzi, 2012). But, many IoT systems have been subject to cyber attacks. In 2021, attackers obtained administrator credentials for Verkada's cloud-managed CCTV platform, exposing live video from more than 150,000 cameras deployed in hospitals, prisons and factories (Randolph & Hunt, 2021). In 2015, researchers remotely took control of a Jeep Cherokee by exploiting a unit that accepted unsigned over-the-air firmware updates, underscoring the risks of insecure automotive IoT (Miller & Valasek, 2015). In 2017, a U.S. FDA (United States Food and Drug Administration) advisory revealed that Abbott (formerly St. Jude) implantable pacemakers accepted unauthenticated radio commands capable of altering therapy parameters or draining batteries (U.S. Food and Drug Administration, 2017). Therefore, security surveys warn that any smart city system must guarantee confidentiality, integrity and authenticity of the data all the way from field device to cloud, even when the devices are small and resource-constrained (Alaba, Othman, Hashem & Alotaibi, 2017). Mostly due to performance concerns, different forms of symmetric cryptosystems are used for such protection, which requires both parties, the client and the server, possessing the same key. Thus, distributing these keys, and periodically renewing them are also needed to be performed in a secure way. To this end, the most practical approach involves using long-term private keys in a protocol employing public-key cryptography to derive short-term session keys.

If an attacker later extracts the server's long-term private key, every past data flow secured with its derived session keys can be decrypted offline. The solution to this problem is referred to as forward secrecy (or perfect forward secrecy): exposure of long-term credentials must not reveal earlier session keys (Boyd & Gellert, 2019). Modern transport security protocols such as TLS 1.3 achieve this property by combining an authenticated certificate with an ephemeral Diffie–Hellman exchange; once both sides discard the per-session secret, recordings of the handshake cannot be brute-forced even if the certificate is stolen years later (Springall, Durumeric & Halderman, 2016).

Given storage and bandwidth limitations of IoT (Internet of Things) devices, the most prominent public-key cryptosystem seems to be ECC (Elliptic Curve Cryptography) and its key agreement scheme, namely, ECDH (Elliptic Curve Diffie-Hellmann) key exchange scheme (Barker, Chen, Roginsky, Vassilev & Davis, 2018).

In this thesis, in order to address the abovementioned security issues of smart city applications and IoT systems, we developed an end-to-end key agreement protocol based on ECC. Our protocol computes a shared secret between client and the server using ephemeral elliptic curve key pairs. This shared secret is used to derive single-use symmetric keys that provide confidentiality, integrity, and authentication of messages sent between the client and the server. As a result, forward secrecy is achieved since if a key encrypting transmitted data gets compromised, the rest of the communication remains confidential. Similarly, if a master key or its corresponding elliptic curve private key is compromised, only the session associated with that master key is affected. In addition to these, cryptoperiods[1] are assigned to each key based on the recommendations of NIST (National Institute of Standards and Technology) (Barker, 2020). Storage of the secrets on the server utilize TPM (Trusted Platform Module) which is a chip on the processor (Trusted Computing Group, 2008). The TPM encrypts the secrets while they are not used, and decrypts them only when they are to be used in the application which makes the keys being compromised more challenging.

We complement implementation testing with symbolic analysis of our key-exchange protocol using the Tamarin prover. In the symbolic Dolev–Yao attacker model, cryptography is abstracted as perfect operations and the network is controlled by an active adversary who can intercept, replay, and synthesize messages from known components. Tamarin supports unbounded sessions, equational theories for Diffie–Hellman, signatures, and symmetric encryption, and proof obligations expressed as temporal lemmas (Basin, Cremers, Dreier & Sasse, 2025) (The Tamarin Team, 2024). Using this framework we prove secrecy of keys, confidentiality and authenticity of data, server and client authentication, agreement on both master and derived keys, and perfect forward secrecy.

This thesis also presents the performance measurements of some cryptographic operations on three different devices, a Raspberry Pi 3, a Raspberry Pi 4, and a laptop computer. These operations include symmetric encryption, message authentication,

[1] A *cryptoperiod* of a key is a limitation on the duration in which it can be used.

authenticated symmetric encryption, ECDH, and ECDSA (Elliptic Curve Digital Signature Algorithm). Each of these operations are benchmarked as standalone operations with respect to different key sizes and data sizes (except for ECDH for which key size is the only variable), using *pycryptodome* (Anonymous, 2025d) and *cryptography* (Anonymous, 2025c) libraries on Python programming language. The performance of the TPM operations are measured on the server by the time it takes to take the ownership of the TPM, generating encryption keys, and encrypting/decrypting data using the generated keys. Finally, a web API (Application Programming Interface) implementation is built and the durations of request-response cycles to complete key exchange operations are decomposed into client computations, network latency and server processing.

Each component of the implementation has been analyzed using static code analyzers. The client library that is implemented in python is analyzed using *bandit* (Anonymous, 2025b), *ruff* (Astral, 2025), and *pylint* (Anonymous, 2025a) tools. The server library that is implemented in C is analyzed using the *cppcheck* tool (Marjamäki, 2025). Finally, the application layer that is implemented in *.NET* is analyzed using *.NET*'s built-in *RoslynSecurityGuard* package (Arteau, 2025), and *CodeQL* tool (GitHub Inc., 2025). According to the results, no valid security issues were detected.

The rest of this thesis is organized as follows: Chapter 2 provides an overview of the specifications for cryptographic algorithms used, definitions of different types of keys used in these algorithms, and the frequency at which the keys are regenerated. Chapter 3 describes the performance test setups for the cryptographic operations on different platforms, and presents the results of these tests for each device. Chapter 4 discusses the design and implementation of the proposed key management protocol, including the issues related to key distribution and storage of the keys in the server and formal verification of the protocol. Chapter 5 details the integration of the key exchange protocol to a server application and evaluates end-to-end performance of the web API. Finally, Chapter 6 concludes the thesis.

2. BACKGROUND

This chapter provides theoretical knowledge about cryptographic primitives, operations, technologies, and platforms that are used. First, a list of definitions about different types of keys used by cryptographic algorithms along with definitions of some operations and technologies. Then, the algorithms to be tested and used along with the test platforms are specified. Also, algebraic definitions and operational details of these algorithms are provided. Lastly, usage limits of the mentioned keys, and suggestions about their lifetimes are made.

2.1 Cryptographic Operations and Test Platforms

Cryptographic primitives, which will be used for secure storage and transmission of data, are divided into three groups. First group is the pure symmetric encryption and MAC (Message Authentication Code) operations that consists of AES - CBC (Advanced Encryption Standard - Cipher Block Chaining) mode of operation and HMAC (Hash-based Message Authentication Code). AES - CBC is a confidentiality mode of operation, that requires an IV (initialization vector), for which encryption and decryption operations are defined in equations 2.1 and 2.2 respectively. For encryption, first an exclusive or operation is applied to the IV and the first block of the plaintext. The cipher function is applied to the result of the exclusive or operation and the output is the first block of the ciphertext. For all the following blocks of the plaintext, the exclusive or operation is applied to the plaintext block and the ciphertext of the previous block. The result is given to the cipher function to obtain the corresponding ciphertext block. To decrypt the first ciphertext block, an exclusive or operation is applied to the IV and output of the inverse cipher function on the first ciphertext block, and the result is the first block of the plaintext. For the remaining ciphertext blocks, the exclusive or operation is applied to the previous ciphertext and the output of the inverse cipher function on the ciphertext block itself to obtain the corresponding plaintext block (Dworkin, 2001).

$$\begin{aligned} C_1 &= CIPH_K(P_1 \oplus IV) \\ C_j &= CIPH_K(P_j \oplus C_{j-1}) \quad \text{for} \quad j = 2...n \end{aligned} \tag{2.1}$$

$$\begin{aligned} P_1 &= CIPH_K^{-1}(C_1) \oplus IV \\ P_j &= CIPH_K^{-1}(P_j) \oplus C_{j-1} \quad \text{for} \quad j = 2...n \end{aligned} \tag{2.2}$$

HMAC is used to check the integrity of a message using a key K and a hash function H as specified in equation 2.3. The inputs to the HMAC function are K and *text*, where K is the key used for generating the MAC and *text* is the data for which the MAC is generated. K_0 is obtained from K, and has the same length as the block size. The pads, *ipad* and *opad*, are the inner and outer pads that have the bytes *x'36'* and *x'5c'* repeated block size times respectively (National Institute of Standards and Technology, 2008).

$$MAC(text) = HMAC(K, text) = H((K_0 \oplus opad)||H((K_0 \oplus ipad)||text) \tag{2.3}$$

The second group is the authenticated symmetric encryption operations that includes AES - GCM (Galois Counter Mode) and AES - CCM (Counter with Cipher Block Chaining Message Authentication Code) mode. The authenticated encryption function, $GCM - AE_K$, of GCM mode of operation generates a ciphertext C, and a tag T of length t as shown in 2.4 where IV is the initialization vector, P is the plaintext, and A is the additional authenticated data. The authenticated decryption function, $GCM - AD_K$, returns the plaintext if authentication succeeds, and $FAIL$ otherwise, which is represented by equation 2.5 (Dworkin, 2007).

$$\begin{aligned} C &= GCTR_K(inc_{32}(J_0), P) \\ T &= MSB_t(GCTR_K(J_0, S)) \\ GCM\_AE_K(IV, P, A) &= (C, T) \end{aligned} \tag{2.4}$$

$$\begin{aligned} T' &= MSB_t(GCTR_K(J_0, S)) \\ P &= GCTR_K(inc_{32}(J_0), C) \\ GCM\_AD_K(IV, C, A, T) &= \begin{cases} P, & \text{if} \quad T = T' \\ FAIL, & \text{else} \end{cases} \end{aligned} \tag{2.5}$$

The definitions of additional variables, J_0 and S, that are used to generate C and T in the encryption function or P and *T'* in the decryption function are shown in 2.6. The functions $GCTR_K$ and $GHASH_H$ are defined in equations 2.7 and 2.8 respectively. For a bit string X, the result of incrementing its rightmost s bits regarded as a binary integer is defined as $inc_s(X)$, and its leftmost s bits is defined as $MSB_s(X)$ (Dworkin, 2007).

$$\begin{aligned} H &= CIPH_K(0^{128}) \\ s &= 128 \cdot \lceil len(IV)/128 \rceil - len(IV) \\ J_0 &= \begin{cases} IV || 0^3 1 || 1, & \text{if} \quad len(IV) = 96 \\ GHASH_K(IV || 0^{s+64} || [len(IV)]_{64}), & \text{if} \quad len(IV) \neq 96 \end{cases} \\ u &= 128 \cdot \lceil len(C)/128 \rceil - len(C) \\ v &= 128 \cdot \lceil len(A)/128 \rceil - len(A) \\ S &= GHASH_K(A || 0^v || C || 0^u || [len(A)]_{64} || [len(C)]_{64}) \end{aligned} \tag{2.6}$$

$$\begin{aligned} n &= \lceil len(X)/128 \rceil \\ X &= X_1 || X_2 || ... || X_n^* \\ CB_1 &= ICB \\ CB_i &= inc_{32}(CB_{i-1}) \quad \text{for} \quad i = 2...n \\ Y_i &= X_i \oplus CIPH_K(CB_i) \quad \text{for} \quad i = 1...n \\ Y_n^* &== X_n^* \oplus MSB_{len(X_n^*)}(CIPH_K(CB_n)) \\ Y &= Y_1 || Y_2 || ... || Y_n^* \\ GCTR_K(ICB, X) &= Y \end{aligned} \tag{2.7}$$

$$\begin{aligned} Y_0 &= o^{128} \\ Y_i &= (Y_{i-1} \oplus X_i) \cdot H \\ GHASH_H(X_1 || X_2 || ... || X_m) &= Y_m \end{aligned} \tag{2.8}$$

The generation and encryption function along with the verification and decryption function of the other authenticated symmetric encryption mode, AES - CCM, are defined in equations 2.9 and 2.10 where N is the nonce, A is the associated data, P is the plaintext, and C is the ciphertext. In the generation and encryption function, the CBC-MAC mechanism generates a MAC on the formatted data, and counter mode encryption is applied to the plaintext and the MAC individually to result in the ciphertext. In the decryption function, the plaintext and the MAC are obtained by applying counter mode decryption on the ciphertext, and CBC-MAC mechanism verifies the MAC on the formatted data. The decryption and verification function returns the plaintext if the verification succeeds, otherwise it returns $INVALID$ (Dworkin, 2004).

$$\begin{gathered} T = MSB_{Tlen}(Y_r) \\ C = (P \oplus MSB_{Plen}(S))||(T \oplus MSB_{Tlen}(S_0)) \\ CCM\_GE_K(N,P,A) = C \end{gathered} \tag{2.9}$$

$$\begin{gathered} P = MSB_{Clen-Tlen}(C) \oplus MSB_{Clen-Tlen}(S) \\ T = LSB_{Tlen}(C) \oplus MSB_{Tlen}(S_0) \\ CCM\_VD_K(N,A,C) = \begin{cases} P, & \text{if} \quad T = MSB_{Tlen}(Y_r) \\ INVALID, & \text{else} \end{cases} \end{gathered} \tag{2.10}$$

The extra variables that are computed within these functions, Y and S, are defined in equation 2.11. The formatted data blocks from B_0 up to B_r are produced by the formatting function applied to the inputs; and the counter blocks from Ctr_0 to Ctr_m where $m = \lceil Plen/128 \rceil$ are generated by the counter generation function. Rightmost s bits of a bit string X is defined as $LSB_s(X)$ (Dworkin, 2004).

$$\begin{gathered} S_j = CIPH_K(Ctr_j) \quad \text{for} \quad j = 0...m \\ S = S_1||S_2||...||S_m \\ Y_0 = CIPH_K(B_0) \\ Y_j = CIPH_K(B_i \oplus Y_{i-1}) \quad \text{for} \quad i = 1...r \end{gathered} \tag{2.11}$$

The last group is public-key infrastructure using elliptic curves, and it contains ECDH key exchange scheme (Barker et al., 2018) and ECDSA operations (National Institute of Standards and Technology, 2023). The inputs to the $ECDH$ function that is defined in equation 2.12 are, the domain parameters $q, FR, a, b, SEED, G, n, h$ for the curve that is used, the private key d_A of the party that computes the shared secret, and the public key Q_B of the other party; and it returns the shared secret Z which is a byte string. The *element-to-bytes* function used within the $ECDH$

function converts the field element to byte string. Before returning, either an error or Z, all intermediate calculation results such as P and z and the corresponding intermediate results while computing them are destroyed (Barker et al., 2018).

$$\begin{gathered}
P = hd_A Q_b \\
z = x_P \\
Z = element\_to\_bytes(z) \\
ECDH((q,FR,a,b,SEED,G,n,h),d_A,Q_b) = \begin{cases} Z, & \text{if} \quad P \neq \emptyset \\ ERROR, & \text{else} \end{cases}
\end{gathered} \tag{2.12}$$

The ECDSA signature generation function is defined in equation 2.13, and it takes the message M to be signed, the private key d of the signing party, and a hash function $Hash$ as inputs, and generates an integer pair (r,s) as the output. Equation 2.14 defines the ECDSA signature verification function, which takes a message M, a pair of integers (r,s) which is the signature on M, and a public key Q belonging to the private key that signed the message. The intermediate functions, *bytes-to-int* and *element-to-int* are used to represent a byte string as an integer and converting a field element to an integer respectively. The integer n is a domain parameter that belongs to the curve used in the operation (National Institute of Standards and Technology, 2023).

$$\begin{gathered}
H = Hash(M) \\
e = \begin{cases} bytes\_to\_int(H), & \text{if} \quad len(n) \geq hashlen \\ bytes\_to\_int(MSB_{\lceil log_2(n) \rceil}(H)), & \text{else} \end{cases} \\
R = [k]G \\
r_1 = element\_to\_int(x_R) \\
r = r_1 \bmod n \\
s = ((k^{-1} \bmod n) \cdot (e + r \cdot d)) \bmod n \\
ECDSA\_sign(M,d,Hash) = (r,s)
\end{gathered} \tag{2.13}$$

$$\begin{aligned} H &= Hash(M) \\ e &= \begin{cases} bytes\_to\_int(H), & \text{if } len(n) \geq hashlen \\ bytes\_to\_int(MSB_{\lceil log_2(n) \rceil}(H)), & \text{else} \end{cases} \\ u &= (e \cdot (s^{-1} \bmod n)) \bmod n \\ v &= (r \cdot (s^{-1} \bmod n)) \bmod n \\ R &= [u]G + [v]Q \\ r_1 &= element\_to\_int(x_R) \\ ECDSA\_verify(M,(r,s),Q) &= \begin{cases} reject, & \text{if } r \notin [1, n-1] \; or \; s \notin [1, n-1] \\ reject, & \text{if R is the identity element} \\ reject, & \text{if } r \neq r_1 \\ accept, & \text{if } r = r_1 \end{cases} \end{aligned} \tag{2.14}$$

ECDH and ECDSA operations are applied on the curves *P-256*, *P-384*, and *P-521*. The domain parameters of these curves consist of the values $p, a, b, G, n,$ and h where the curve equation is $y^2 = x^3 + ax + b \quad modp$ with p being a prime number that specifies the finite field. G is the base point of the curve, h is the cofactor and n is the order of G (Chen, Moody, Randall, Regenscheid & Robinson, 2023) (Standards for Efficient Cryptography Group, 2010).

2.2 Definitions

The following list contains the definitions of some key types, algorithms, or components that will be referred to in the following chapters.

- **Private Signature Key:** The private key of the key pair used by public-key cryptosystems to generate digital signatures for authentication purposes.
- **Public Signature-Verification Key:** The public key of the key pair used by public-key cryptosystems to verify digital signatures for authentication purposes.

- **Private Ephemeral Key-agreement Key:** The private key of the temporary key pair used by public-key cryptosystems to generate a symmetric key.
- **Public Ephemeral Key-agreement Key:** The public key of the temporary key pair used by public-key cryptosystems to generate a symmetric key.
- **Symmetric Master Key / Key-derivation Key:** The key responsible for deriving other symmetric keys that will be used for purposes such as encrypting data.
- **Symmetric Data Encryption Key - Data Transport:** A key used by symmetric-encryption algorithms to encrypt and decrypt the data transmitted over the network to protect its confidentiality.
- **Symmetric Data Encryption Key - Data Storage:** A key used by symmetric-encryption algorithms to encrypt and decrypt the data stored on the server to protect its confidentiality.
- **HKDF (Hash Based Key Derivation Function):** A function used with the symmetric master key to generate symmetric keys that will be employed for purposes such as encrypting data.
- **TPM:** A computer chip (microcontroller) on which passwords, certificates, and encryption keys that can be used for platform security can be stored securely.

2.3 Key Types and Cryptoperiods

To secure the communication between the client and the server, various cryptographic primitives and their associated keys will be employed. For the elliptic curve based operations of the public-key infrastructure that will be used in key distribution, the server side will make use of a private signature key, a public signature-verification key, a private ephemeral key-agreement key, and a public ephemeral key-agreement key. On the client side, only a private ephemeral key-agreement key and a public ephemeral key-agreement key will be used.

As a result of the elliptic curve based key exchange, both the server and the client will generate the same symmetric master key. A key derivation function will use this master key to produce symmetric data encryption keys. These symmetric data encryption keys will be used by the client and the server to encrypt the messages to be sent, and decrypt the received messages. In addition, a different symmetric data encryption key will be used by the server to store data received from clients securely.

The usage periods of the different key types that will be employed on the server and the clients are shown in Table 2.1. The periods of use were chosen on the basis of the key management recommendations published by NIST (Barker, 2020).

Table 2.1 Cryptoperiods of Key Types Used in Client and Server

Key Type	Originator-Usage Period	Recipient-Usage Period
Private Signature Key	1-3 Years	-
Public Signature-Verification Key	-	Many Years
Private Ephemeral Key-agreement Key	One Key-agreement	
Public Ephemeral Key-agreement Key	One Key-agreement	
Symmetric Master Key	1 day - 1 week	-
Symmetric Data Encryption Key - Data Transport	Connection period	Originator-Usage period + 3 years
Symmetric Data Encryption Key - Data Storage	2 years	5 years

3. TESTING FOR CRYPTOGRAPHIC UNITS

Some cryptographic primitives such as pure symmetric encryption operations, MAC operations, authenticated encryption operations, and elliptic curve based operations are implemented and tested on the following platforms with the given specifications:

1. Laptop: Lenovo ThinkPad E14 - Intel Core i5 processor @ 2.40 GHz
2. Raspberry Pi 4: Arm Cortex-A72 processor @ 1.80 GHz (Raspberry Pi Ltd, 2025b)
3. Raspberry Pi 3: Arm Cortex-A53 processor @ 1.40 GHz (Raspberry Pi Ltd, 2025a)

These tests consist of correctness and performance tests on the operations of the algorithm. To test the correctness, the outputs to the NIST test vectors (National Institute of Standards and Technology, 2016) (National Institute of Standards and Technology, nda) (National Institute of Standards and Technology, ndb) (National Institute of Standards and Technology, ndc) as inputs are examined. Then, using pairwise connection between each device pair is established and the compatibility of implementations on different platforms and libraries is checked. Upon successful completion of correctness tests, the performance of the libraries on different platforms were recorded with respect to changes in size of input data making use of the pairwise connections between devices.

3.1 Test Setup

All tests for all algorithms were developed on all devices in the Python programming language using two different libraries, *pycryptodome* (Anonymous, 2025d) and *cryp-*

tography (Anonymous, 2025c). The operations whose performance was measured are:

- encryption and decryption for pure symmetric-encryption primitives
- code generation and code verification for pure message authentication code primitives
- encryption and tagging (generation of the authentication code), decryption and tag verification for authenticated symmetric-encryption primitives, and
- key generation, key exchange, message signing, and message verification for public-key infrastructure primitives.

Before testing operational correctness on different platforms using the same or different libraries, the correctness of the libraries themselves was verified with the test vectors published by NIST (National Institute of Standards and Technology, 2016) (National Institute of Standards and Technology, nda) (National Institute of Standards and Technology, ndb) (National Institute of Standards and Technology, ndc). These vectors contain predetermined inputs together with the outputs that the algorithms were expected to produce for those inputs. The libraries used produced the expected outputs for the inputs in these test vectors.

To test operational correctness when the same or different libraries were used on different platforms, data had to be transferred between two platforms. For this purpose, the Python's *socket* library was used. By means of this library, one platform assumes the role of server and the other of client. After the client connects to the server, the following test methods are applied for the respective building blocks:

- **Pure symmetric-encryption operations:** The server generates three random keys, one 128 bits, one 192 bits, and one 256 bits, and a 128 bits IV to send to the client. Then it generates a random plaintext and shares it with the client. Both platforms encrypt this plaintext with the library in use. The client sends its ciphertext to the server; if the ciphertexts produced on both platforms are identical, the encryption operation is considered correct. After that, each platform sends its ciphertext to the other, both decrypt the received ciphertexts, and return the resulting plaintexts. If the decrypted plaintext matches the original predefined plaintext, the decryption operation is also accepted as correct.
- **Pure message authentication code operations:** he server generates a random 512-bit key and shares it with the client. The server and the client each generate a random message and, using the previously generated key,

compute a MAC for the message. In turn, the server and the client send the message to each other by appending the MAC. The recipient attempts to verify the received message and the MAC with the predefined key; If the verification succeeds, the generation and verification of the MAC are considered successful.

- **Authenticated encryption modes:** The server generates three random keys, one 128 bits, one 192 bits, and one 256 bits, a 96 bits IV and 160 bits of associated data, and shares them with the client. Then it produces a random plaintext and shares it as well. Both platforms perform encryption and tagging on this plaintext. The client sends the ciphertext and authentication tag it produced to the server; If the ciphertexts and tags generated on both platforms are identical, the encryption and tagging operations are accepted as correct. Each platform then sends its ciphertext and authentication tag to the other. Decryption and verification are performed on these data; if the decrypted plaintext matches the original predefined plaintext and the tag is verified, the decryption and verification operations are considered successful.

- **Elliptic curve based operations:** On both the server and the client, a private key and a public key are generated using the selected elliptic curve. The public keys are exchanged between the server and the client. Using its private key and the other party's public key, each side derives a 256-bit key using the ECDH key exchange scheme (Barker et al., 2018). Then, the server and the client each generate a random message and, using the derived key, compute an HMAC using the SHA3-512 (Secure Hash Algorithm) hash function. Each side sends its message together with the HMAC to the other. If both parties can verify the received code with the derived key, the key-exchange operation is deemed successful.

  To test digital-signature generation and verification (National Institute of Standards and Technology, 2023), both platforms generate a random message, sign this message with their own private key, and send the message and the digital signature to the other side. If the received message and signature can be verified with the sender's public key, digital-signature generation and verification are accepted as correct.

During the performance analysis, the time elapsed while executing the operations was measured with Python's built-in *time* library. Because the duration of a single operation was very short, the same operations were repeated 1 000 times; the elapsed time was measured in seconds, multiplied by 1 000 to express it in milliseconds, and then divided by the repetition count of 1 000, so that the time spent for a single operation was calculated in milliseconds.

In addition, to observe the time each operation spends working on different data sizes, the same operations were carried out on data whose sizes are multiples of 1 000 bytes and fall between 5 000 bytes and 50 000 bytes. These different data sizes were not used for the key-generation and key-exchange operations present in elliptic curve based procedures, because in those operations the size of the key produced depends only on the size of the curve.

3.2 Test Results

The performance tests were run on multiple devices; a laptop, a Raspberry Pi 4, and a Raspberry Pi 3. The results for each device is presented separately. For each device, the performance of each operation using different libraries are compared to each other. The change in performance with respect to input size and with respect to key size are also evaluated.

3.2.1 Test Results on Laptop

The performance results of the AES-CBC mode used for pure symmetric-encryption operations on the laptop can be seen in Figure 3.1. The graphs at the top left and bottom left show the encryption and decryption performance, respectively, obtained with the *pycryptodome* library, while the graphs at the top right and bottom right show the corresponding performance obtained with the *cryptography* library.From these graphs it can be observed that the *pycryptodome* library performs better in both encryption and decryption, and that in both libraries the decryption operation is faster. However, because the measured times fall in the 0.0002 – 0.05 millisecond range owing to the speed of the device used, they are highly sensitive to noise. For this reason, a clear linear increase in processing time with increasing data size cannot be observed, although a general linear rise is still visible within certain error margins.

The performance results of the AES-CCM mode used for authenticated-encryption operations on the laptop are presented in Figure 3.2. The graphs at the top left and bottom left show, respectively, the encryption-plus-tagging and the decryption-

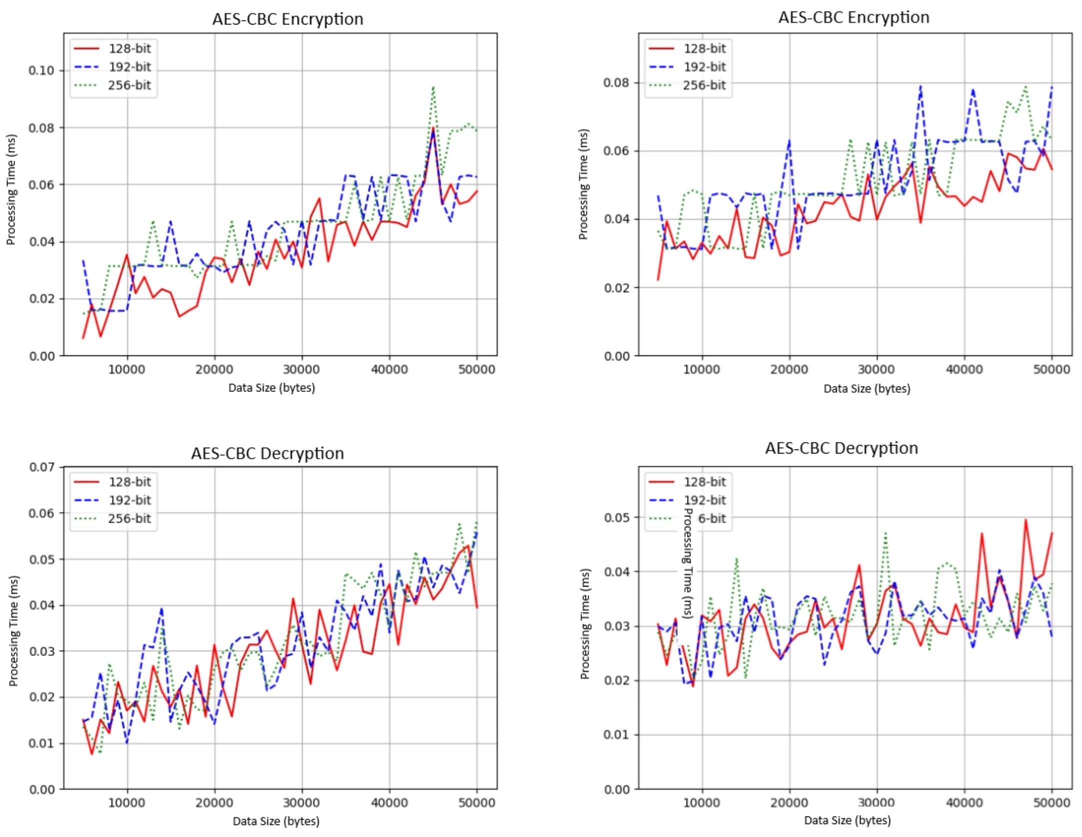


Figure 3.1 Processing time of cryptographic operations using AES-CBC mode with respect to data size (results obtained with *pycryptodome* library: charts on top left and bottom left; results obtained with *cryptography* library: charts on top right and bottom right)

plus-tag-verification performance obtained with *pycryptodome*, while the graphs at the top right and bottom right show the same operations obtained with *cryptography*. According to these graphs, for both encryption-plus-tagging and decryption-plus-tag-verification the *cryptography* library exhibits better performance than *pycryptodome*, and in both libraries decryption-plus-tag-verification takes less time. As in CBC mode, the processing times in CCM mode fall in the 0.005 – 0.1 millisecond range and are therefore very sensitive to noise. Nevertheless, a general linear increase in processing time with increasing data size can be mentioned.

The performance results of the other authenticated-encryption mode, AES-GCM, on the laptop are shown in Figure 3.3. The graphs at the top left and bottom left display, respectively, the encryption-plus-tagging and the decryption-plus-tag-verification performance obtained with *pycryptodome*, while the graphs at the top right and bottom right present the corresponding results obtained with *cryptography*. As with CCM mode, the cryptography library shows better performance in both operations with GCM mode. Although encryption takes less time compared to

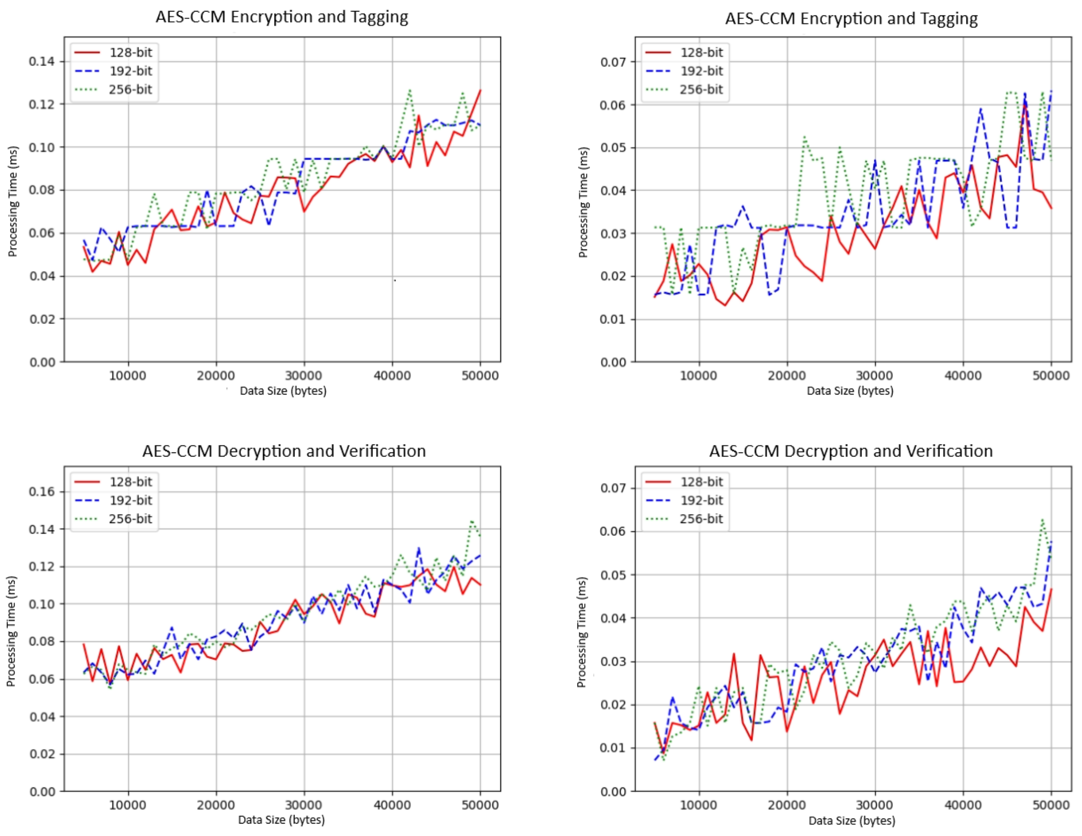


Figure 3.2 Processing time of cryptographic operations using AES-CCM mode with respect to data size (results obtained with *pycryptodome* library: charts on top left and bottom left; results obtained with *cryptography* library: charts on top right and bottom right)

decryption when *pycryptodome* is used and decryption takes less time compared to encryption when *cryptography* is used, the difference is not significant.

In all three AES modes, because the processing times are extremely short, no significant effect of key size on performance has been observed.

The performance findings obtained for elliptic curve based operations on the laptop with different curves can be seen in Figures 3.4 and 3.5. Figure 3.4 shows, on the left the times obtained with *pycryptodome* and on the right those obtained with *cryptography* for generating private and public key pairs for each curve and for deriving a symmetric key using the ECDH protocol. As the size of the elliptic curve increases, the time taken to generate private and public key pairs increases with both libraries, but this increase is more clear with *cryptography*. Similarly, the time required to derive the symmetric key also rises with curve size, an increase that is evident in both libraries. Although *cryptography* runs faster for key generation with the P-256 curve, the difference is not significant; for the other curves, however, *py-*

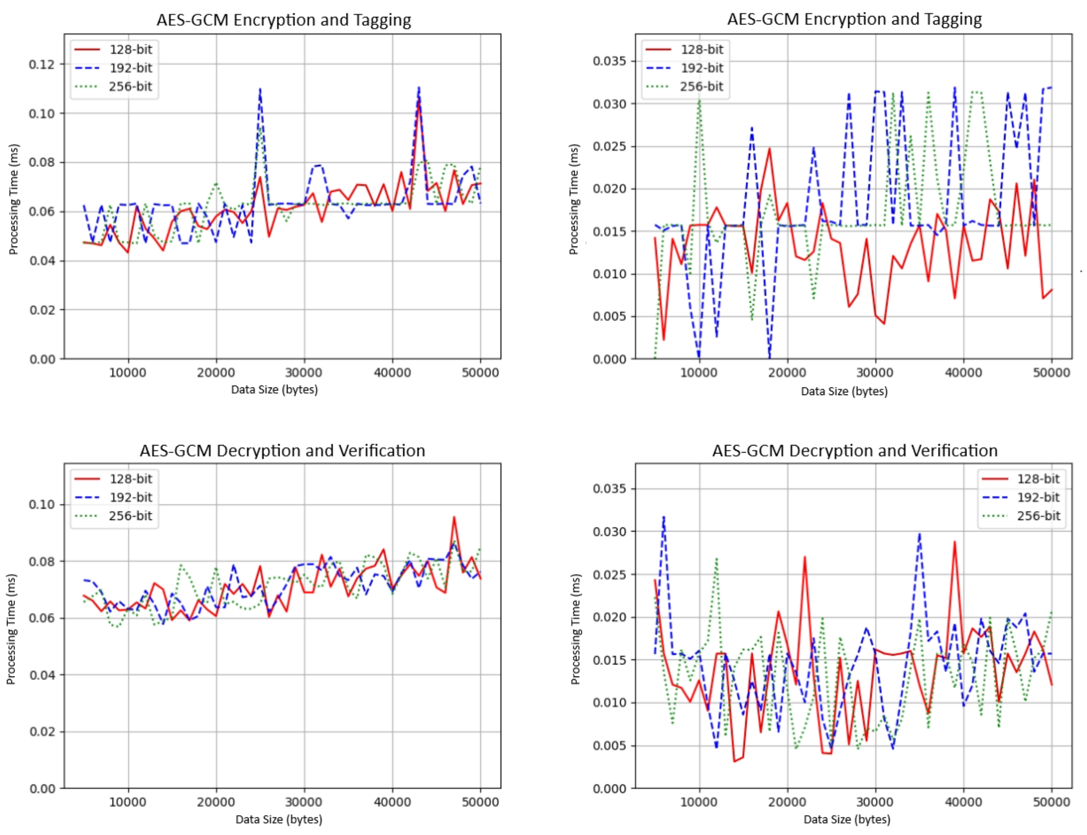


Figure 3.3 Processing time of cryptographic operations using AES-GCM mode with respect to data size (results obtained with *pycryptodome* library: charts on top left and bottom left; results obtained with *cryptography* library: charts on top right and bottom right)

cryptodome completes the operation in a shorter time. In symmetric-key derivation, the *cryptography* library spends less time for every curve.

Figure 3.5 contains graphs that show the ECDSA operation performance obtained with *pycryptodome* (left-hand side) and *cryptography* (right-hand side) for the P-256, P-384 and P-521 curves (top, middle and bottom rows, respectively). As the curve size increases, the time required for both signature generation and verification also increases. For operations on the same curve, the *cryptography* library completes the task in less time for every curve. Because ECDSA operations involve time-consuming steps that are unrelated to data size, a linear increase in processing time with increasing data size is not expected at small data sizes. Nevertheless, with the smallest curve tested (P-256) a linear rise in processing time with increasing data size can be observed, and this rise is greater when *cryptography* is used, because the total time is shorter and the data-size-dependent increase in processing time thus becomes more prominent. In four of the graphs—those at the top left, middle left, bottom left and top right—the time taken to verify the signature is, as expected, longer than the time taken to generate it. In the middle right and bottom right graphs, however, contrary to expectations, signature generation takes longer than signature verification.

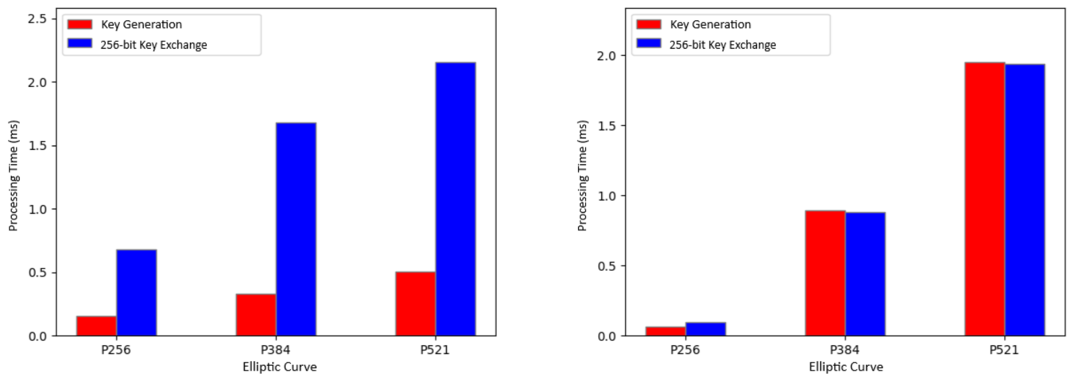


Figure 3.4 Processing time of elliptic curve key pair generation and 256-bit symmetric-key exchange for each curve (results obtained with *pycryptodome* library: left; results obtained with *cryptography* library: right)

Figures 3.6, 3.7, 3.8, 3.9 and 3.10 contain the graphs showing the performance of HMAC generation and verification on the laptop using the SHA3-256, SHA-384, SHA3-384, SHA-512 and SHA3-512 hash functions, respectively. In each figure the graph on the left presents the findings obtained with *pycryptodome*, and the graph on the right presents those obtained with *cryptography*. When the same hash algorithm is used, operations complete in a shorter time with the *cryptography* library. In addition, SHA-3 hash algorithms take more time than SHA-2 hash algorithms, and within the same category the processing time increases as the output length of the hash function grows. This increase in processing time becomes more pronounced as the data size increases; that is, the ratio of the linear rise in processing time to the growth in data size rises with the hash-output length. When the same library and the same hash function are used, no significant performance difference is observed between HMAC generation and HMAC verification.

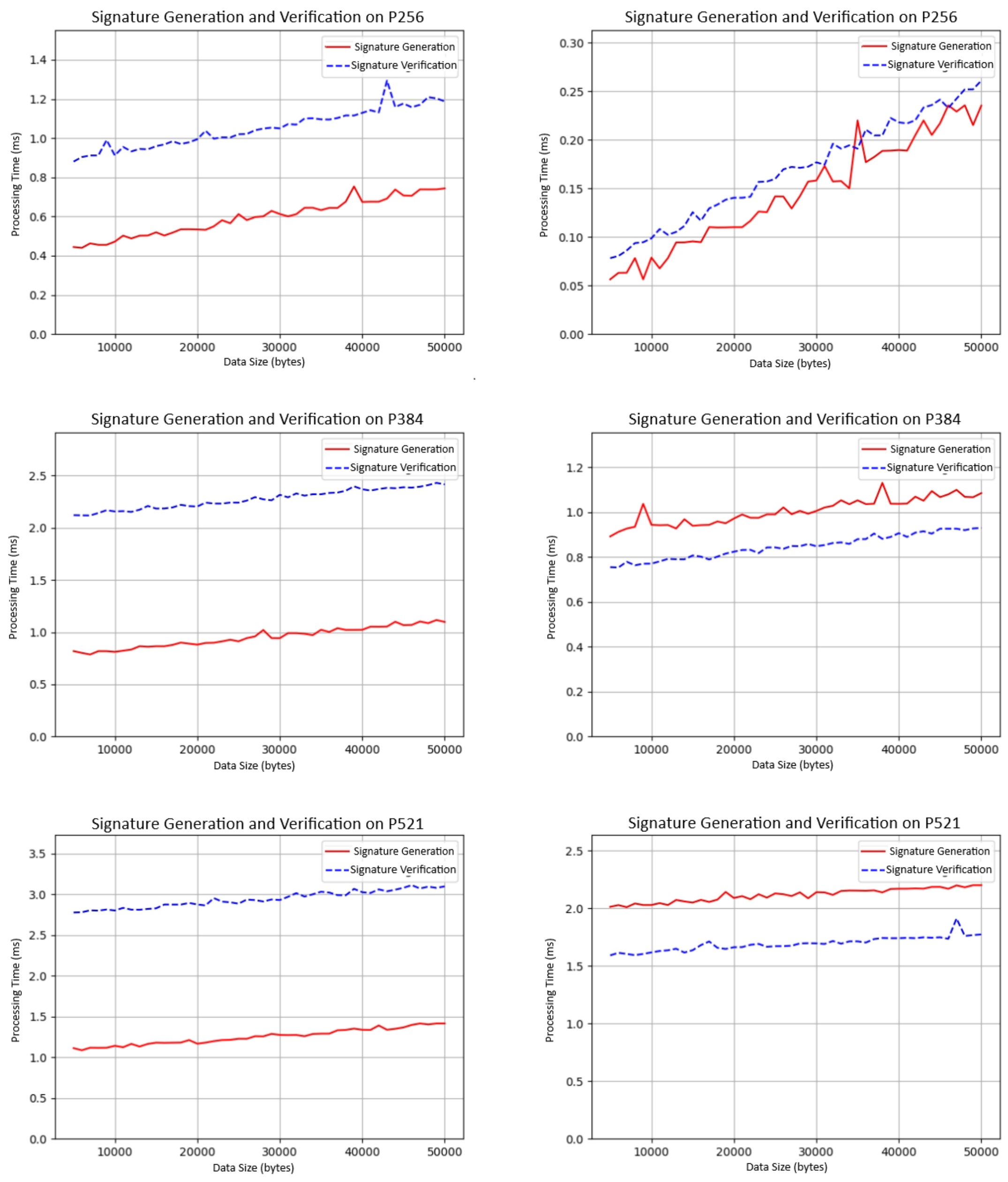


Figure 3.5 Processing time of ECDSA operations on different curves with respect to data size (results obtained with *pycryptodome* library: charts on top-left, middle-left and bottom-left; results obtained with *cryptography* library: charts on top-right, middle-right and bottom-right)

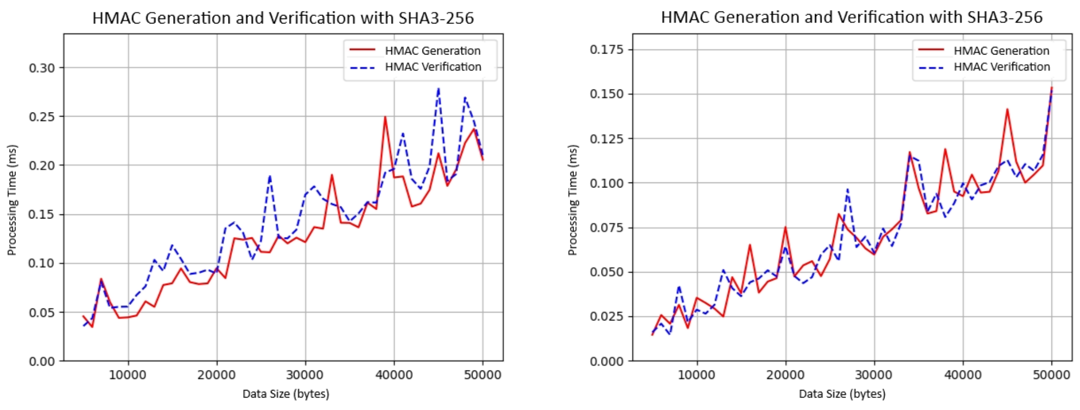


Figure 3.6 Processing time for HMAC generation and verification using SHA3-256 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

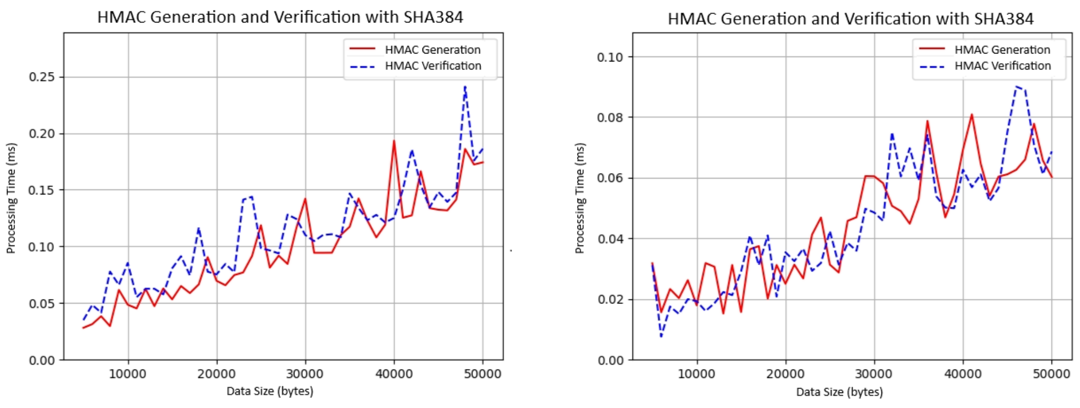


Figure 3.7 Processing time for HMAC generation and verification using SHA384 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

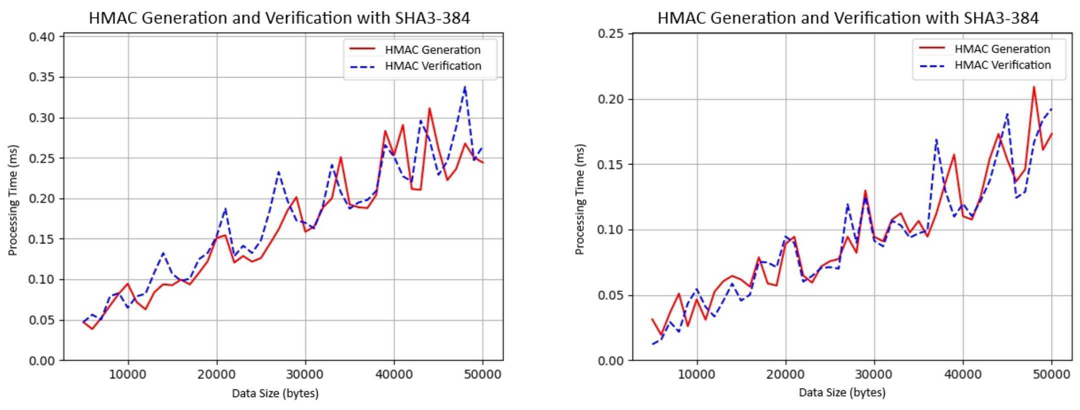


Figure 3.8 Processing time for HMAC generation and verification using SHA3-384 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

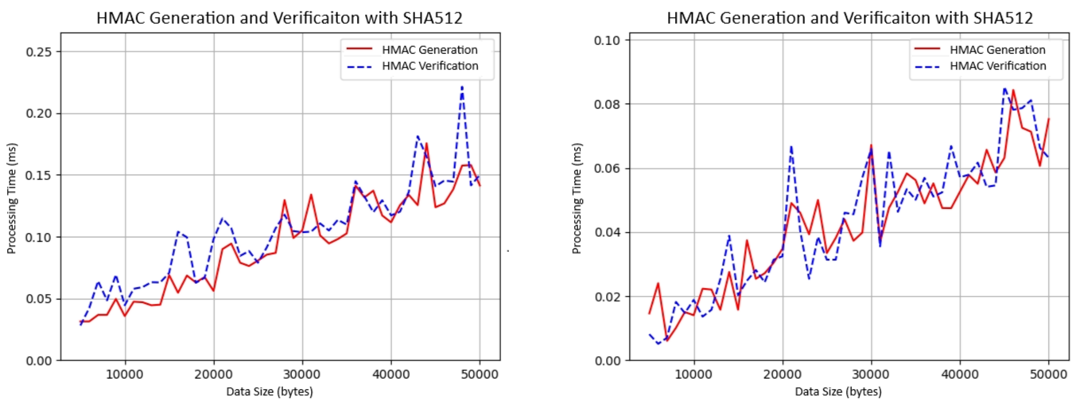


Figure 3.9 Processing time for HMAC generation and verification using SHA512 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

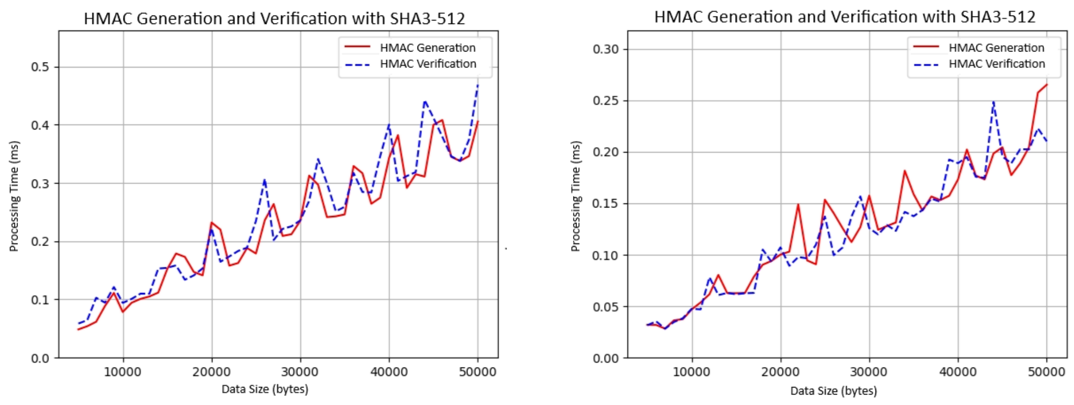


Figure 3.10 Processing time for HMAC generation and verification using SHA3-512 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

3.2.2 Test Results on Raspberry Pi 4

The performance results of the AES-CBC mode used for pure symmetric-encryption operations on the Raspberry Pi 4 device can be seen in Figure 3.11. The graphs at the top left and bottom left show, respectively, the encryption and decryption performance obtained with the *pycryptodome* library, while the graphs at the top right and bottom right show the encryption and decryption performance obtained with the *cryptography* library. In these operations the *pycryptodome* library exhibits better performance and spends similar times on both encryption and decryption. On the other hand, with the *cryptography* library the decryption operation is completed faster than the encryption operation. Because processing times on the Raspberry Pi 4 are longer than on the laptop, the influence of the noise factor is smaller; as a result, a linear increase in processing time with increasing data size can be observed. For small data sizes, no significant effect of key size on processing time is observed. However, as data size increases, the rate of the linear rise in processing time grows as the key size increases, and the impact of key size on processing time becomes stronger.

The performance results of the AES-CCM mode used for authenticated-encryption operations on the Raspberry Pi 4 device are shown in Figure 3.12. The graphs at the top left and bottom left present, respectively, the encryption-plus-tagging and the decryption-plus-tag-verification performance obtained with *pycryptodome*, while the graphs at the top right and bottom right present the corresponding performance obtained with *cryptography*. In these operations the *pycryptodome* library displays better performance. There is no notable difference between the time spent on encryption-plus-tagging and the time spent on decryption-plus-verification, and this holds for both libraries. Since processing times on the Raspberry Pi 4 are longer than on the laptop, the effect of the noise factor is smaller and, as a result, a linear increase in processing time with increasing data size can be observed. For small data sizes no significant effect of key size on processing time is observed. However, as data size increases, the rate of the linear rise in processing time increases with key length, and the influence of key size on processing time becomes more clear as in CBC.

The performance results of the other authenticated-encryption mode, AES-GCM, on the Raspberry Pi 4 device are presented in Figure 3.13. The graphs at the top left and bottom left show, respectively, the encryption-plus-tagging and the decryption-plus-tag-verification performance obtained with *pycryptodome*, while the graphs at the top right and bottom right show the corresponding performance obtained with

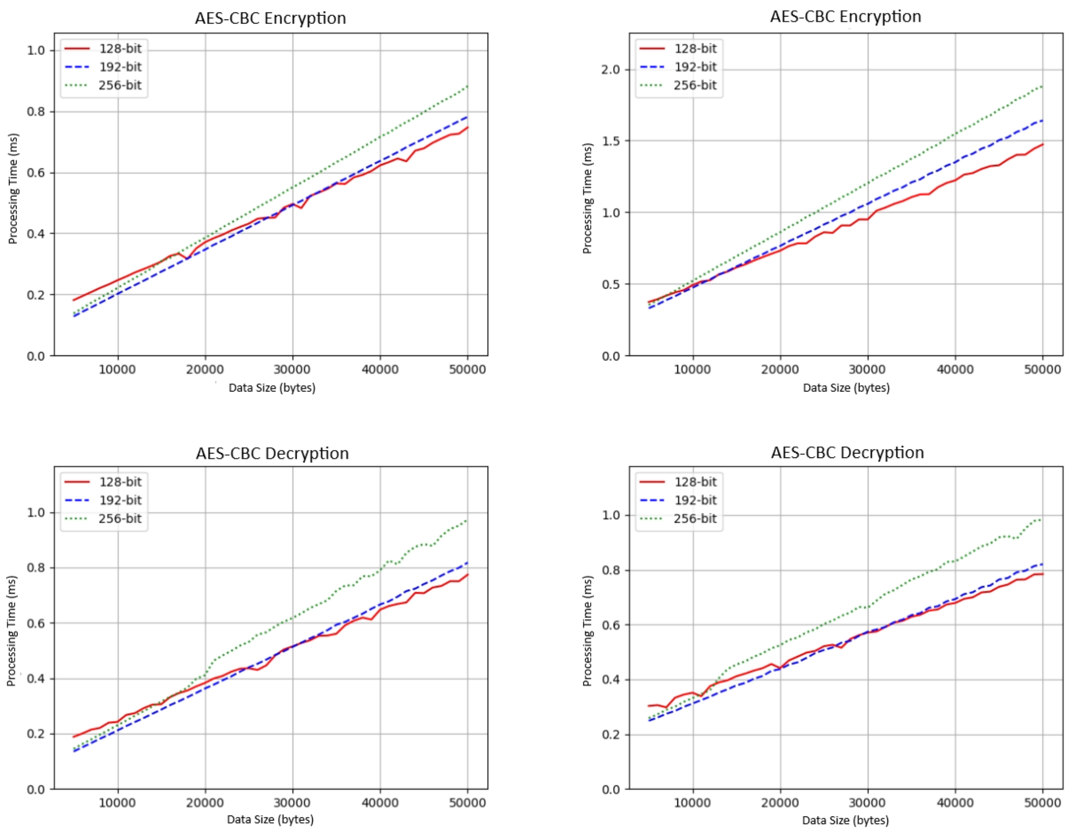


Figure 3.11 Processing time of cryptographic operations using AES-CBC mode with respect to data size (results obtained with *pycryptodome* library: charts on top left and bottom left; results obtained with *cryptography* library: charts on top right and bottom right)

cryptography. In these operations the *cryptography* library achieves better performance. There is no major difference between the time spent on encryption-plus-tagging and the time spent on decryption-plus-verification, and this applies to both libraries. Since processing times on the Raspberry Pi 4 are longer than on the laptop, the impact of the noise factor is smaller and, consequently, a linear rise in processing time with increasing data size can be observed. For small data sizes no significant effect of key size on processing time is observed. However, as data size increases, the effect of key size on the rate of the linear increase in processing time is not as pronounced in this mode as in the other symmetric-encryption modes: using 128-bit and 192-bit keys does not have a noteworthy impact on processing time, whereas the use of a 256-bit key consumes more time than the other key lengths but the difference is negligible.

The performance findings obtained on the Raspberry Pi 4 device for elliptic curve based operations with different curves are shown in Figures 3.14 and 3.15. Figure 3.14 depicts, on the left using *pycryptodome* and on the right using *cryptography*, the times spent for generating private and public key pairs and deriving a symmetric

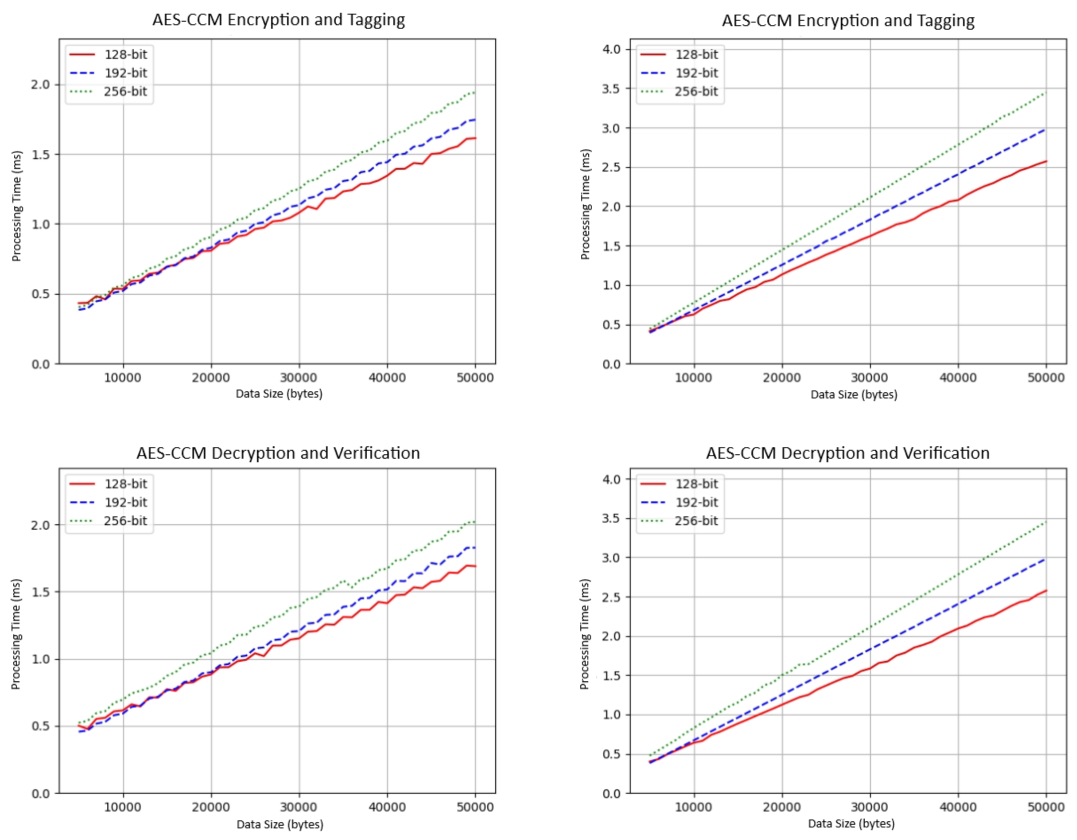


Figure 3.12 Processing time of cryptographic operations using AES-CCM mode with respect to data size (results obtained with *pycryptodome* library: charts on top left and bottom left; results obtained with *cryptography* library: charts on top right and bottom right)

key using the ECDH key-exchange protocol for each curve. Except for the private and public key pair generation times with the P-384 curve, *pycryptodome* spends more time on all operations, and the difference between the private and public key pair generation times with the P-384 curve is not significant. As the elliptic curve size increases, the processing times with *pycryptodome* also increase. However, with *cryptography*, contrary to expectations, the P-384 curve has longer processing times than the other two curves.

Figure 3.15 contains graphs showing the performance of ECDSA operations on the Raspberry Pi 4, where in the top, middle and bottom rows, the P-256, P-384 and P-521 curves are used respectively; the graphs on the left use *pycryptodome* and those on the right use *cryptography*. As the curve size grows, the time required for signature generation and verification with *pycryptodome* increases, whereas, when *cryptography* is used, the P-256 curve completes both operations in the shortest time; however, while the signature-verification performance of the P-384 and P-521 curves is similar, the P-521 curve has a shorter signature-generation time. For operations on the same curve, the *cryptography* library completes the task in less

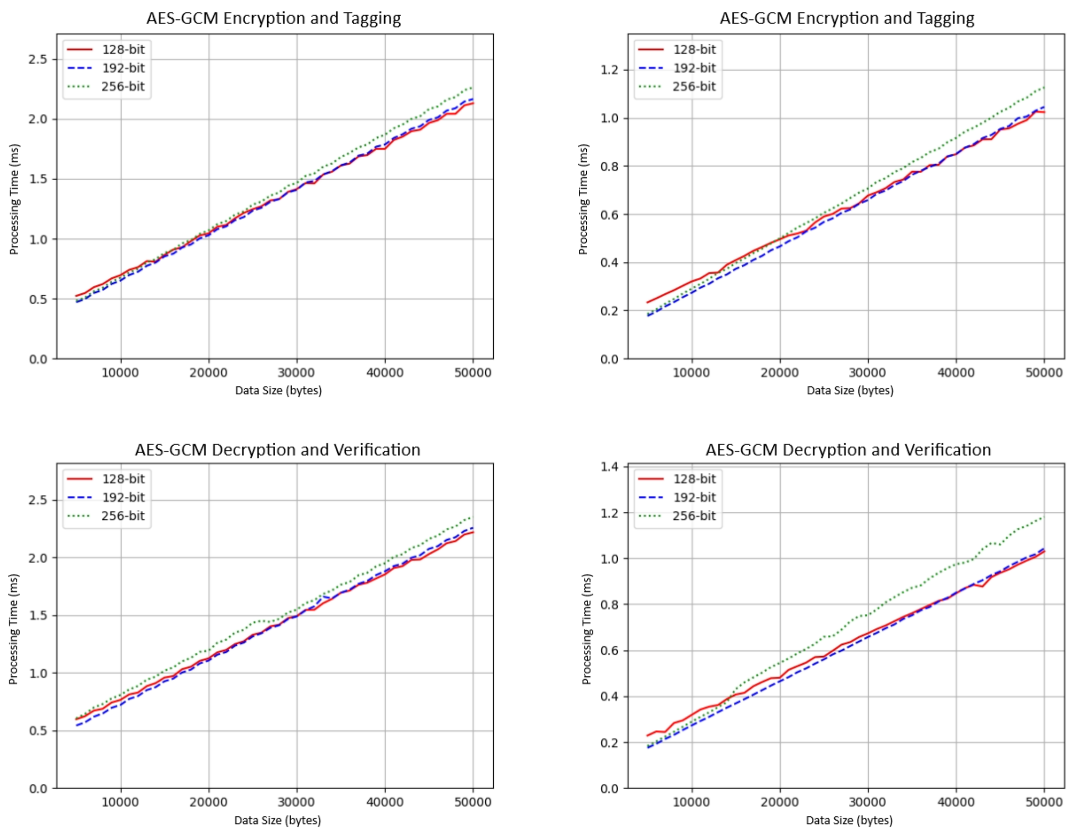


Figure 3.13 Processing time of cryptographic operations using AES-GCM mode with respect to data size (results obtained with *pycryptodome* library: charts on top left and bottom left; results obtained with *cryptography* library: charts on top right and bottom right)

time for every curve. In ECDSA operations, because majority of the time is spent on steps unrelated to data size, a linear increase in processing time with increasing data size is not expected at small data sizes. Nevertheless, when *cryptography* is used, a linear increase in processing time with increasing data size can be observed for the smallest curve tested, P-256, because the total time is sufficiently small and the processing time increase dependent on data size becomes more noticeable. In the top right and bottom right graphs of this figure, as expected, the time taken for signature verification is greater than that for signature generation. However, in the top left, middle left, bottom left and middle right graphs, contrary to expectations, the signature-generation time exceeds the signature verification time.

Figures 3.16, 3.17, 3.18, 3.19 and 3.20 contain the graphs showing the performance of HMAC generation and verification on the Raspberry Pi 4 device using the SHA3-256, SHA-384, SHA3-384, SHA-512 and SHA3-512 hash functions, respectively. In each figure the graph on the left presents the results obtained with *pycryptodome*, and the graph on the right presents those obtained with *cryptography*. When the same hash algorithm is used, operations finish in a shorter time with the *cryptography* library,

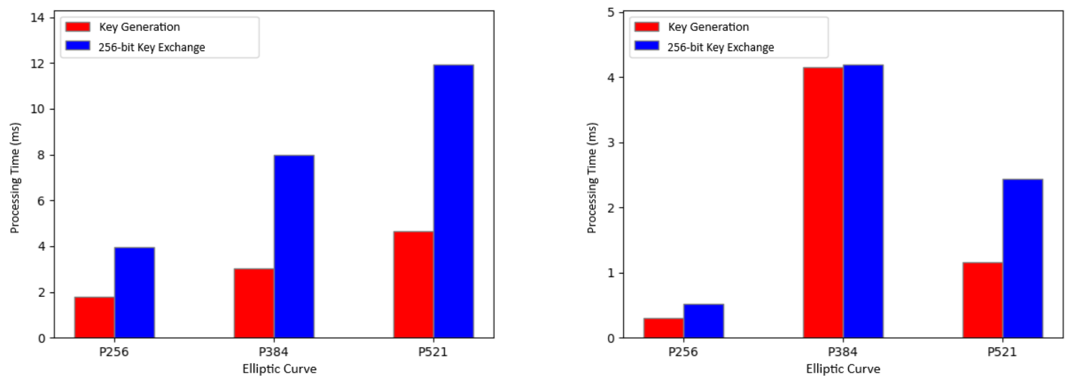


Figure 3.14 Processing time of elliptic curve key pair generation and 256-bit symmetric-key exchange for each curve (results obtained with *pycryptodome* library: left; results obtained with *cryptography* library: right)

and this difference is more notable when hash functions from the SHA-2 category are used. Moreover, with the *cryptography* library, the processing time when algorithms from the SHA-2 category are used is lower than when algorithms from the SHA-3 category are used, whereas no similar difference exists for *pycryptodome*. With both libraries, no distinct performance difference is observed among the algorithms in the SHA-2 category, while in the SHA-3 category the processing time increases as the hash-output length grows.

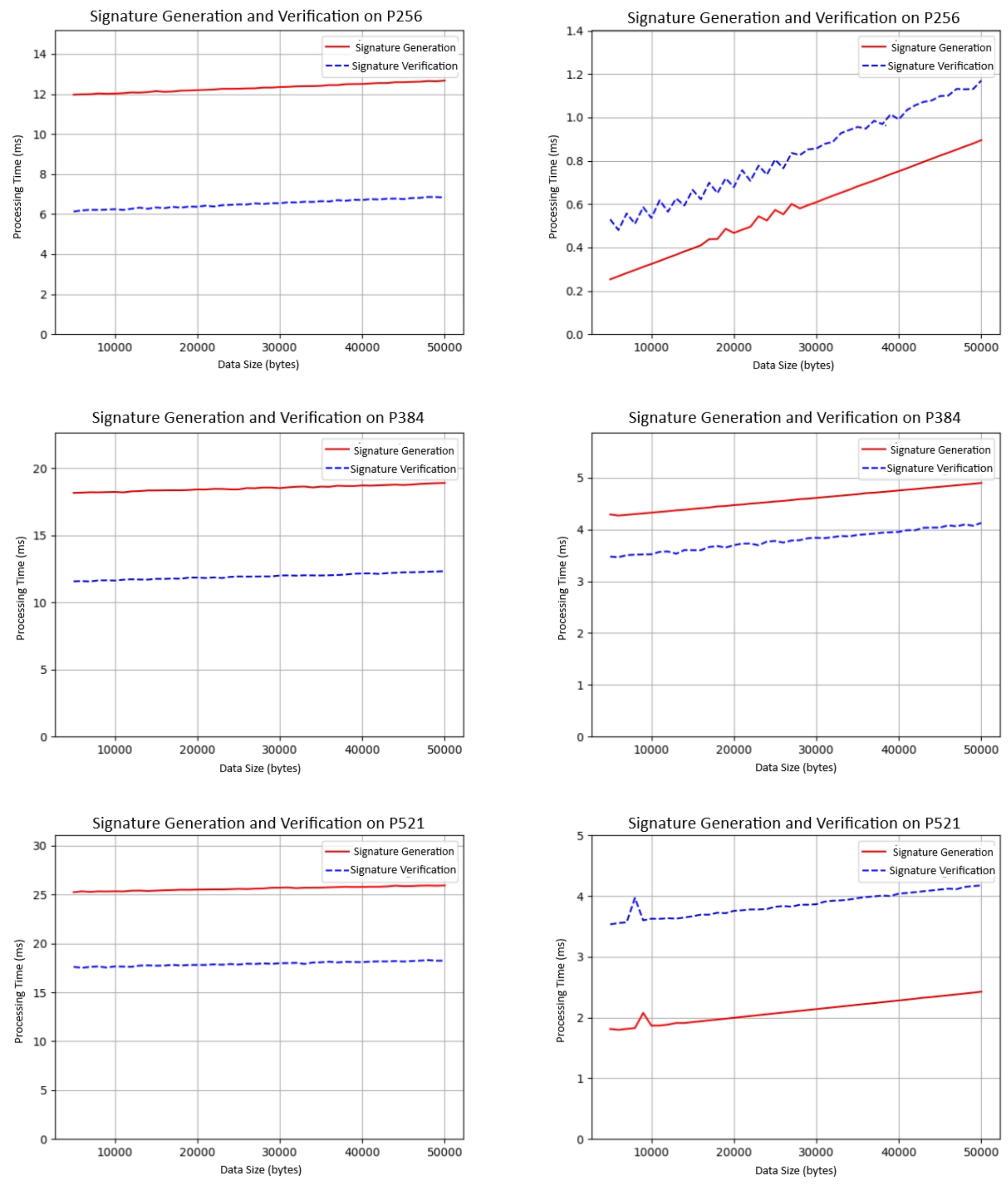


Figure 3.15 Processing time of ECDSA operations on different curves with respect to data size (results obtained with *pycryptodome* library: charts on top-left, middle-left and bottom-left; results obtained with *cryptography* library: charts on top-right, middle-right and bottom-right)

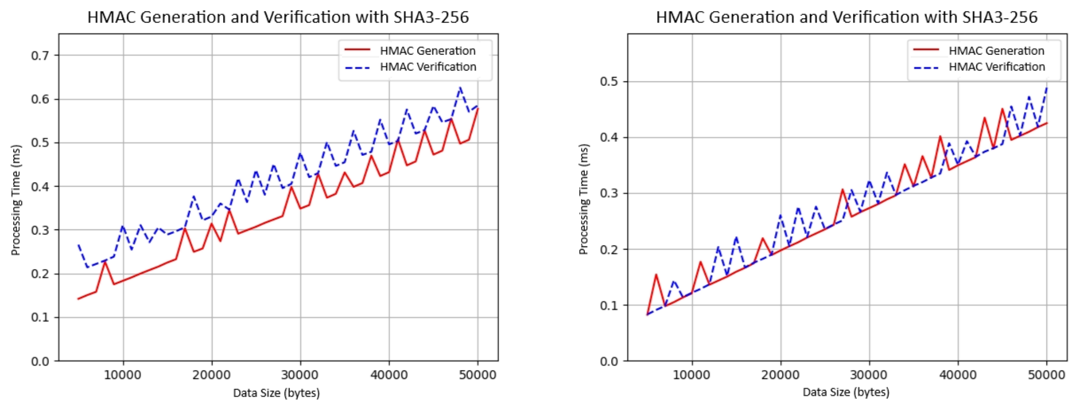


Figure 3.16 Processing time for HMAC generation and verification using SHA3-256 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

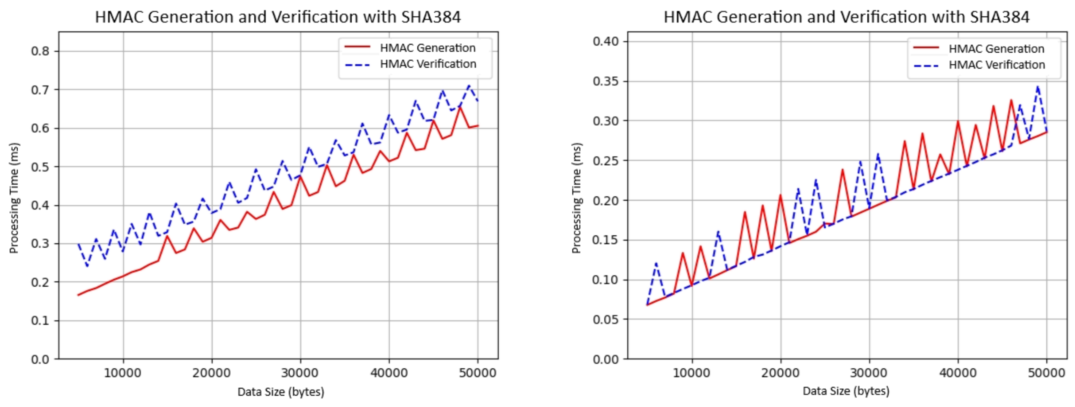


Figure 3.17 Processing time for HMAC generation and verification using SHA384 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

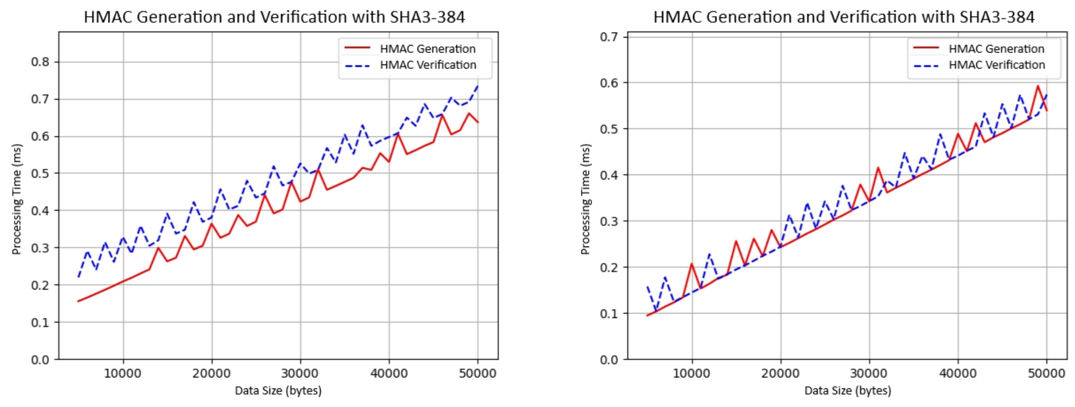


Figure 3.18 Processing time for HMAC generation and verification using SHA3-384 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

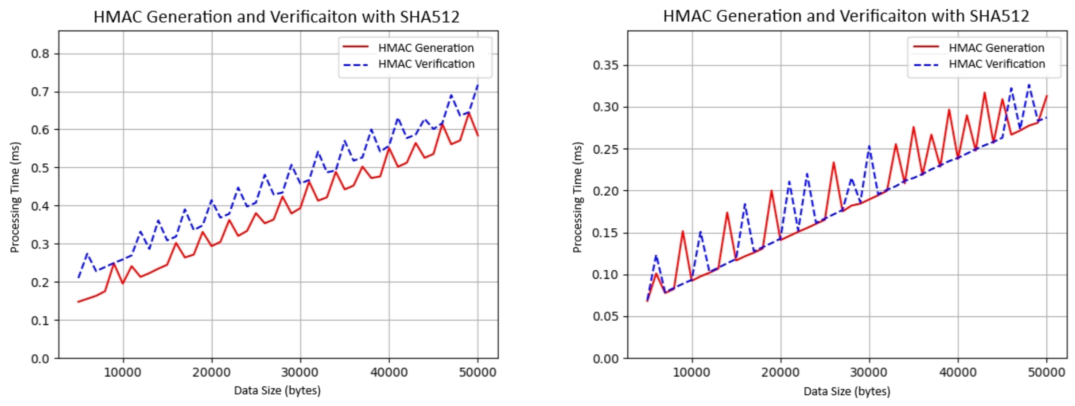


Figure 3.19 Processing time for HMAC generation and verification using SHA512 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

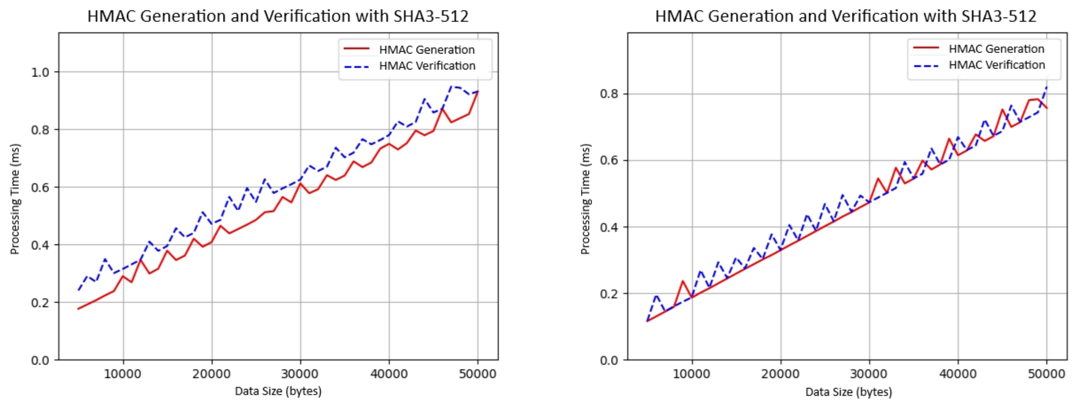


Figure 3.20 Processing time for HMAC generation and verification using SHA3-512 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

3.2.3 Test Results on Raspberry Pi 3

The performance results of the AES-CBC mode used for pure symmetric-encryption operations on the Raspberry Pi 3 device can be seen in Figure 2.21. The graphs at the top left and bottom left show, respectively, the encryption and decryption performance obtained with the *pycryptodome* library, while the graphs at the top right and bottom right show, respectively, the encryption and decryption performance obtained with the *cryptography* library. In these operations, the *pycryptodome* library exhibits better performance. There is no obvious difference between the time spent on encryption and the time spent on decryption, and this holds for both libraries. Because processing times on the Raspberry Pi 3 are longer than on the laptop, the influence of the noise factor is smaller; consequently, a linear increase in processing time with increasing data size can be observed. For small data sizes no significant effect of key size on processing time is observed. However, as data size increases, the rate of the linear rise in processing time grows as the key size increases, and the impact of key size on processing time becomes more evident.

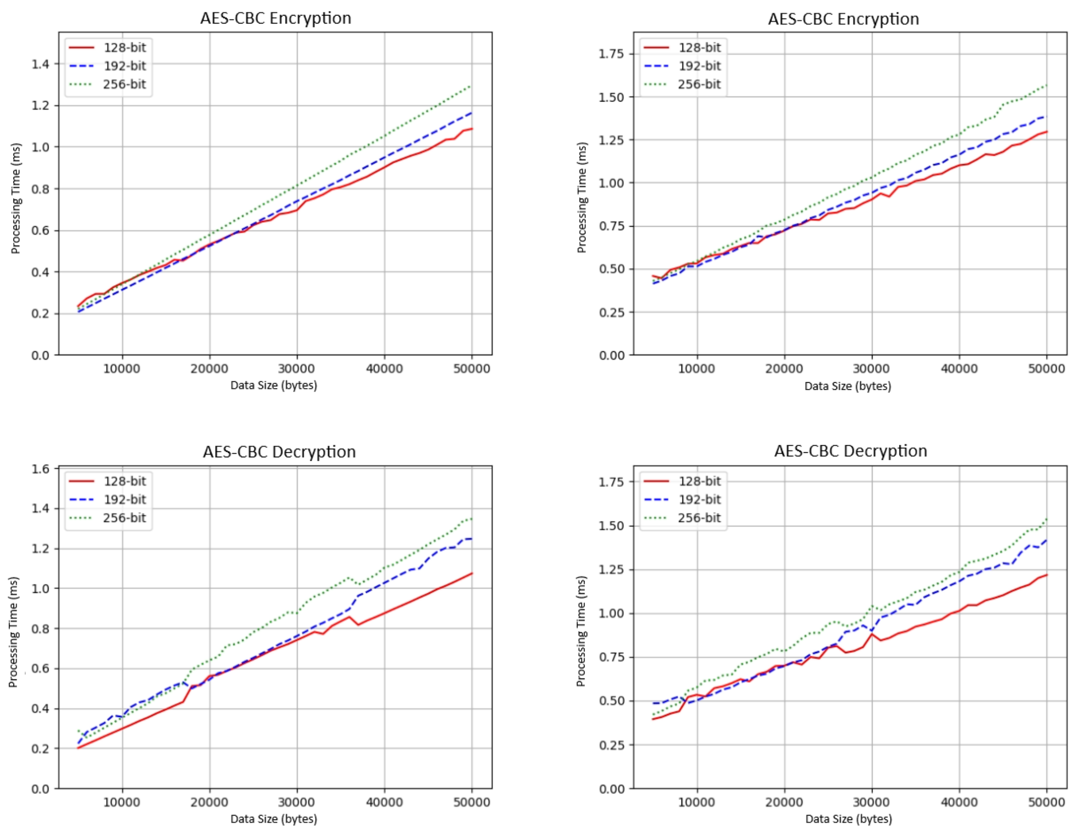


Figure 3.21 Processing time of cryptographic operations using AES-CBC mode with respect to data size (results obtained with *pycryptodome* library: charts on top left and bottom left; results obtained with *cryptography* library: charts on top right and bottom right)

The performance results of the AES-CCM mode used for authenticated-encryption operations on the Raspberry Pi 3 device can be seen in Figure 2.22. The graphs at the top left and bottom left show, respectively, the encryption-plus-tagging and the decryption-plus-tag-verification performance obtained with *pycryptodome*, while the graphs at the top right and bottom right show, respectively, the encryption-plus-tagging and the decryption-plus-tag-verification performance obtained with *cryptography*. In these operations the *cryptography* library exhibits better performance. There is no significant difference between the time spent on encryption-plus-tagging and the time spent on decryption-plus-verification, and this is true for both libraries. Because processing times on the Raspberry Pi 3 are longer than on the laptop, the effect of the noise factor is smaller and, as a result, a linear increase in processing time with increasing data size can be observed. For small data sizes no significant effect of key size on processing time is observed. However, as data size increases, the rate of the linear rise in processing time grows as the key size lengthens, and the influence of key size on processing time becomes more evident.

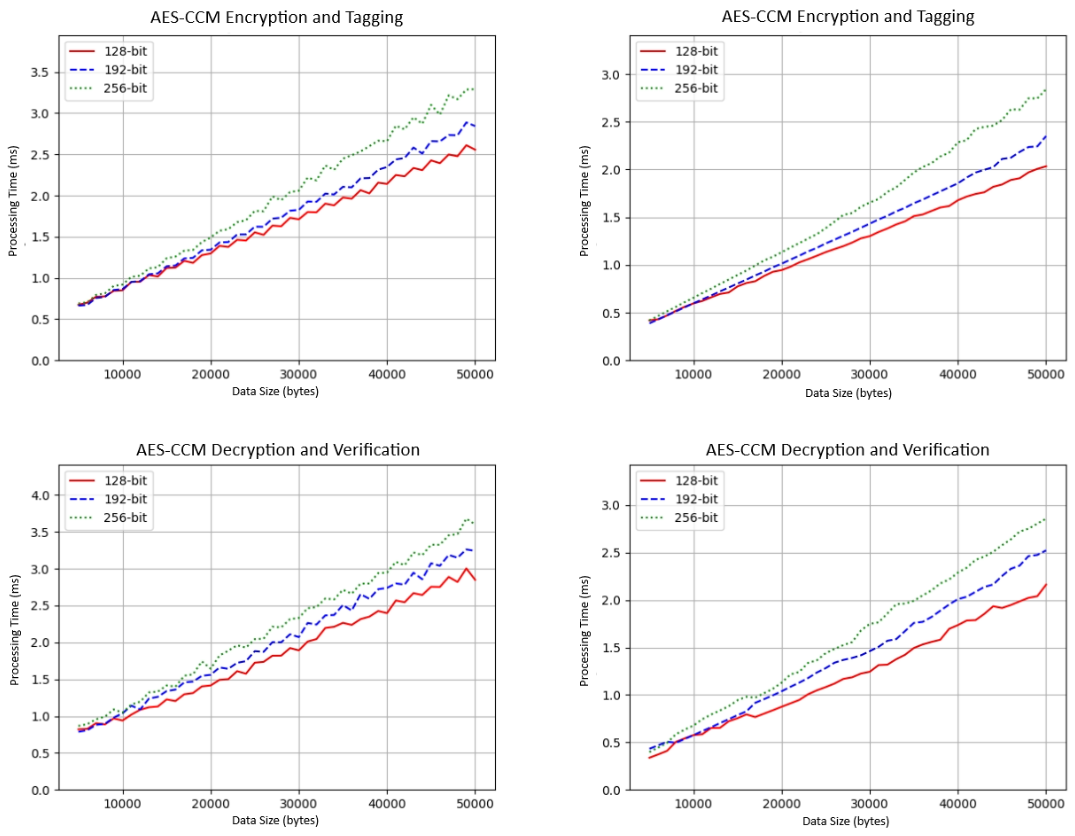


Figure 3.22 Processing time of cryptographic operations using AES-CCM mode with respect to data size (results obtained with *pycryptodome* library: charts on top left and bottom left; results obtained with *cryptography* library: charts on top right and bottom right)

The performance results of the other authenticated-encryption mode, AES-GCM, on the Raspberry Pi 3 device can be seen in Figure 2.23. The graphs at the top left and bottom left show, respectively, the encryption-plus-tagging and the decryption-plus-tag-verification performance obtained with *pycryptodome*, while the graphs at the top right and bottom right show, respectively, the encryption-plus-tagging and the decryption-plus-tag-verification performance obtained with *cryptography*. In these operations the *cryptography* library exhibits better performance. The difference between the time spent on encryption-plus-tagging and the time spent on decryption-plus-verification is negligible, and this situation applies to both libraries. Because processing times on the Raspberry Pi 3 are longer than on the laptop, the impact of the noise factor is smaller and, consequently, a linear increase in processing time with increasing data size can be observed. For small data sizes no significant effect of key size on processing time is observed. However, as data size increases, the rate of the linear rise in processing time grows as the key size lengthens, and the impact of key size on processing time becomes stronger.

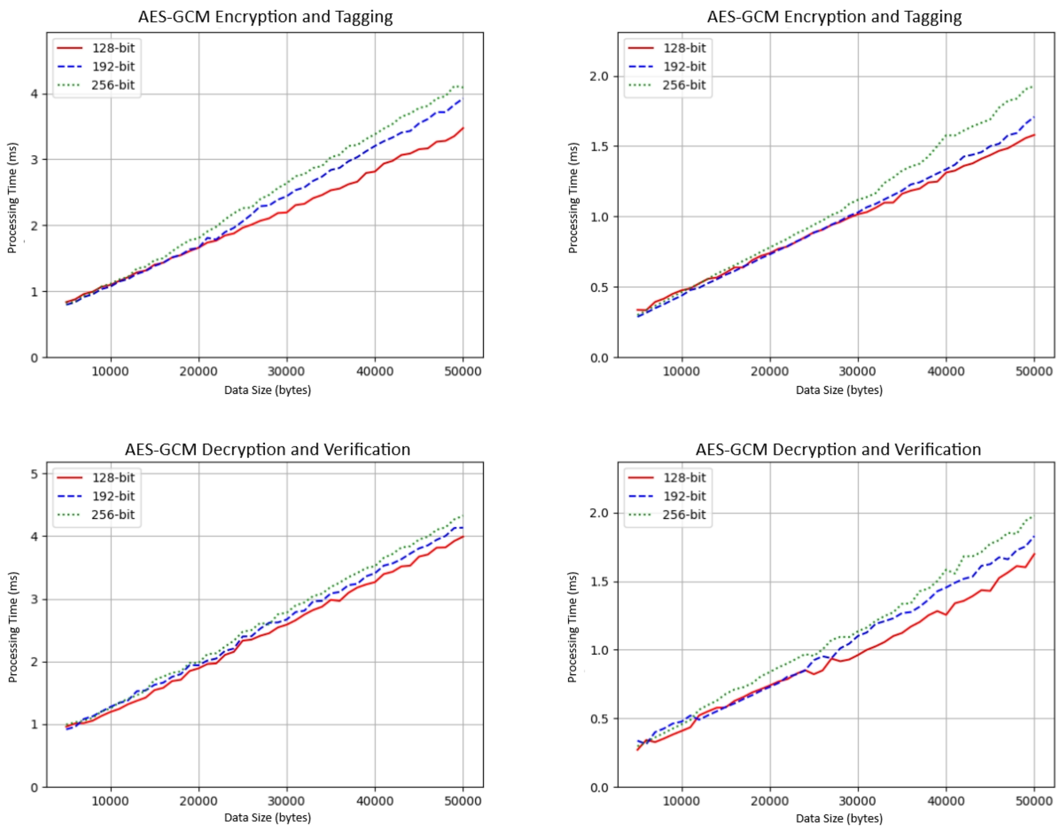


Figure 3.23 Processing time of cryptographic operations using AES-GCM mode with respect to data size (results obtained with *pycryptodome* library: charts on top left and bottom left; results obtained with *cryptography* library: charts on top right and bottom right)

The performance findings obtained on the Raspberry Pi 3 device for elliptic curve based operations with different curves can be seen in Figures 3.24 and 3.25. Figure 3.24 shows, on the left using *pycryptodome* and on the right using *cryptography*, the times taken for generating private and public key pairs and deriving a symmetric key using the ECDH key-exchange protocol for each curve. Except for the private and public key pair generation times with the P-384 curve, *pycryptodome* spends more time on all other operations, and the difference between the private and public key pair generation times with the P-384 curve is not significant. As the elliptic curve size increases, the processing times with *pycryptodome* also increase. However, with *cryptography*, contrary to expectations, the P-384 curve has longer processing times than the other two curves.

Figure 3.25 contains graphs showing the performance of ECDSA operations on the Raspberry Pi 3, where in the top, middle and bottom rows, the P-256, P-384 and P-521 curves are used respectively; the graphs on the left use *pycryptodome* and those on the right use *cryptography.* As the curve size grows, the time required for signature generation and verification with *pycryptodome* increases; however, when *cryptography* is used, both operations are completed in the shortest time on the P-256 curve, but the operations performed on the P-521 curve finish in less time than those performed on the P-384 curve. For operations on the same curve, the *cryptography* library completes the task in less time for every curve. In ECDSA operations, because considerable time is spent on steps unrelated to data size, a linear increase in processing time with increasing data size is not expected at small data sizes. Nevertheless, when *cryptography* is used, a linear increase in processing time with increasing data size can be observed for the smallest curve tested, P-256, because the total time is sufficiently small and the processing-time increase dependent on data size becomes more noticeable. In the top right and bottom right graphs of this figure, as expected, the time taken for signature verification is greater than that for signature generation. However, in the top left, middle left, bottom left and middle right graphs, contrary to expectations, the signature-generation time exceeds the signature-verification time.

Figures 3.26, 3.27, 3.28, 3.29 and 3.30 contain the graphs showing the performance of HMAC generation and verification on the Raspberry Pi 3 device using the SHA3-256, SHA-384, SHA3-384, SHA-512 and SHA3-512 hash functions, respectively. In each figure the graph on the left presents the results obtained with *pycryptodome*, and the graph on the right presents those obtained with *cryptography.* When the same hash algorithm is used, operations finish in a shorter time with the *cryptography* library, and this difference is more pronounced when hash functions from the SHA-2 category are used. Moreover, with the *cryptography* library, the processing

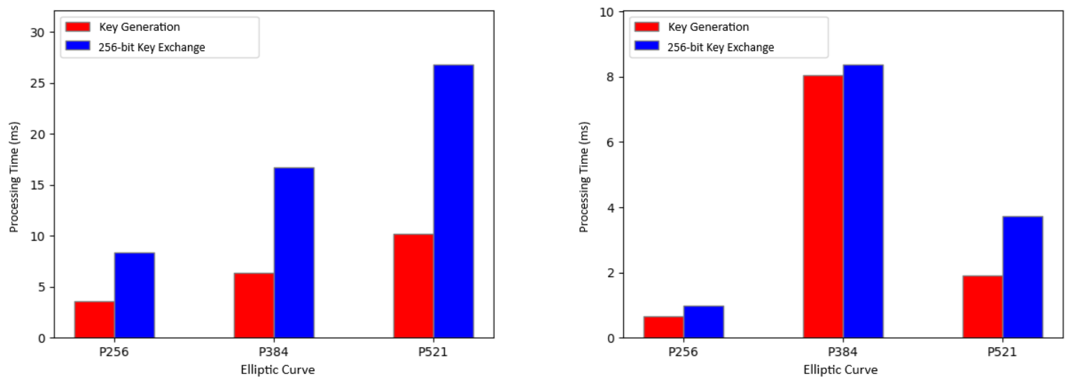


Figure 3.24 Processing time of elliptic curve key pair generation and 256-bit symmetric-key exchange for each curve (results obtained with *pycryptodome* library: left; results obtained with *cryptography* library: right)

time when algorithms from the SHA-2 category are used is lower than when algorithms from the SHA-3 category are used, whereas no similar difference exists for *pycryptodome*. With both libraries, no distinct performance difference is observed among the algorithms in the SHA-2 category, while in the SHA-3 category the processing time increases as the hash-output length grows.

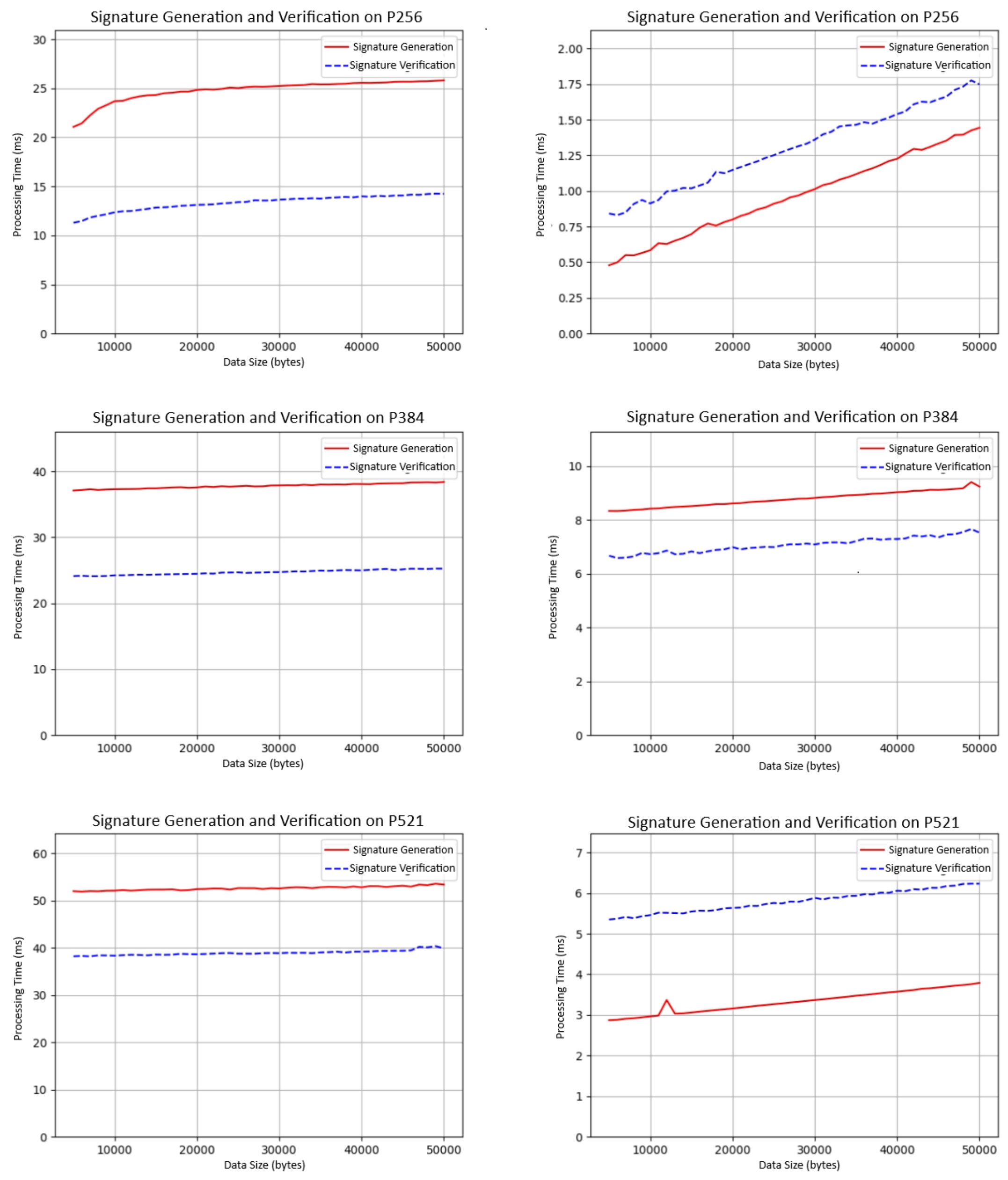


Figure 3.25 Processing time of ECDSA operations on different curves with respect to data size (results obtained with *pycryptodome* library: charts on top-left, middle-left and bottom-left; results obtained with *cryptography* library: charts on top-right, middle-right and bottom-right)

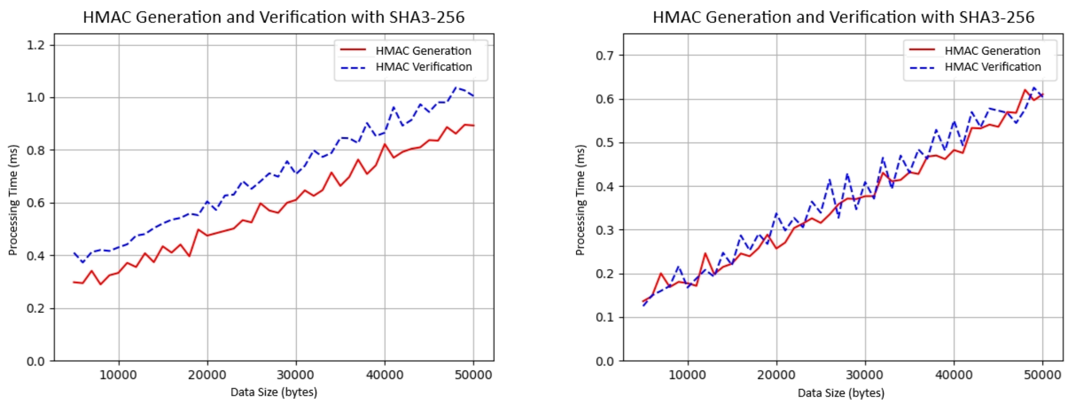


Figure 3.26 Processing time for HMAC generation and verification using SHA3-256 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

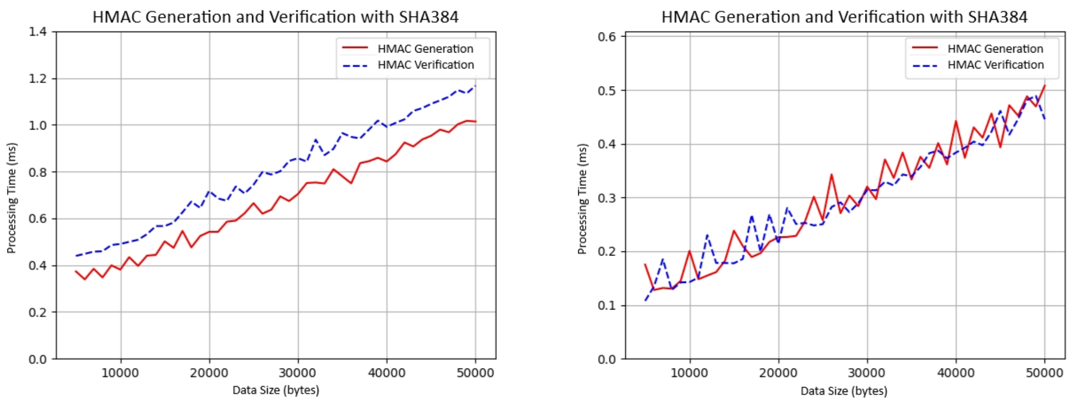


Figure 3.27 Processing time for HMAC generation and verification using SHA384 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

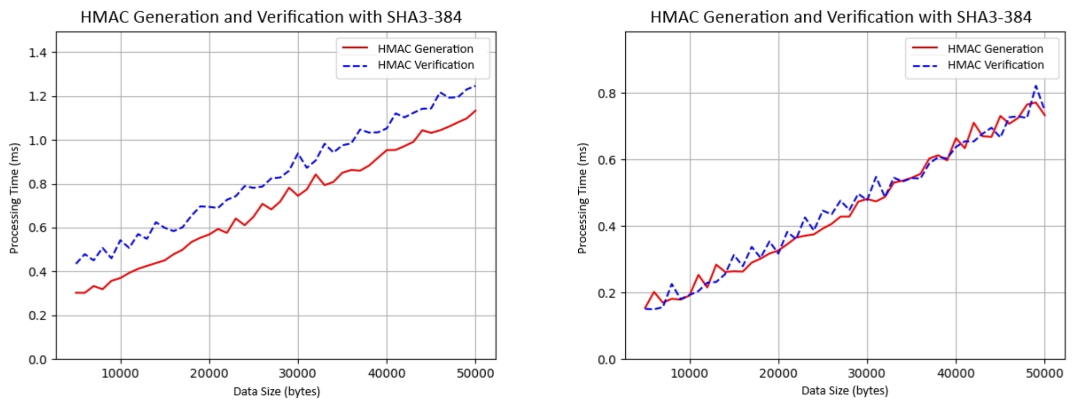
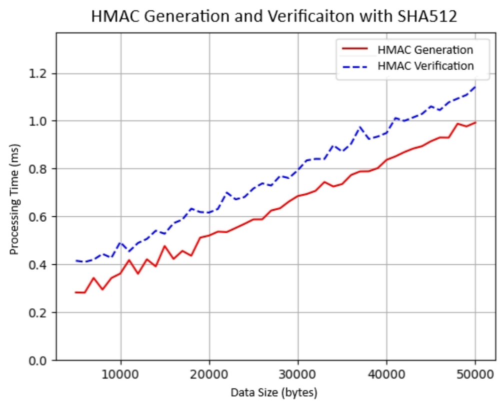


Figure 3.28 Processing time for HMAC generation and verification using SHA3-384 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)



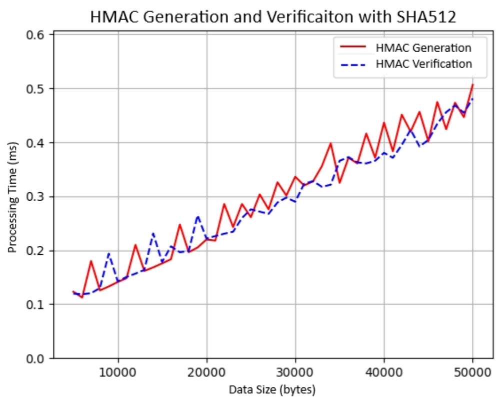
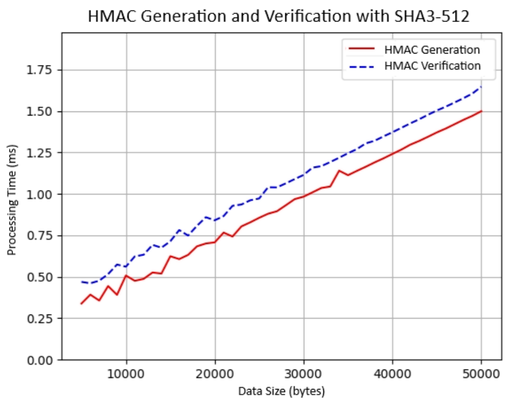


Figure 3.29 Processing time for HMAC generation and verification using SHA512 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)



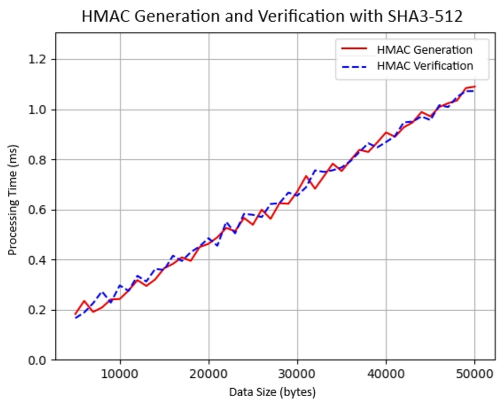


Figure 3.30 Processing time for HMAC generation and verification using SHA3-512 with respect to data size (result obtained with *pycryptodome* library: left; result obtained with *cryptography* library: right)

4. KEY MANAGEMENT

Symmetric encryption and MAC algorithms require both communicating parties to have the same symmetric key, and this symmetric key must be unknown to third parties. To achieve this, a key management protocol must be employed. This key management protocol can be decomposed into two parts. The distribution of keys between the server and clients is the first part. ECDH will be used for this purpose as outlined in the first section. The second part is the secure storage of the distributed keys. The clients do not need additional measures to store the keys since only the root certificate will be stored permanently in the clients, and it is a public data which requires no security. On the other hand, the server needs to store some secrets in the permanent non-volatile memory. To securely store these keys, TPM will be utilized as detailed in the second section. The designed protocol is implemented in the *C* programming language using *OpenSSL* (OpenSSL Foundation, Inc., 2025) and *tpm2-tss* (tpm2-software, 2024) (Trusted Computing Group, 2025) libraries, and the implementation details are described in the third section.

4.1 Key Distribution

Communication between the client and the server will be secured by symmetric encryption methods. The key to be used in symmetric encryption must be known to the server and the client but must remain unknown to third parties. To ensure this, the ECDH key agreement scheme (Barker et al., 2018) will be employed. The steps to be followed for the key exchange, which will be carried out at the start of every session, are shown in Figure 4.1.

Before the key exchange begins, the server already possesses a static private and public key pair that will be used for ECDSA operations. This public key is contained in a digital certificate (Temoshok & Abruzzi, 2018) signed by a certificate authority (Akram, Barker, Clatterbuck, Dodson, Everhart, Gilbert, Haag, Johnson, Kapasouris, Lam, Pleasant, Raguso, Souppaya, Symington, Turner & Wilson, 2020). When communication starts, the server sends this certificate to the client. The client verifies the certificate received from the server by using the root certificate (Akram et al., 2020), which contains the certificate authority's public key. This root certificate has been embedded in the client during its installation. If the client can verify the server's certificate, it will generate an ephemeral private and public key pair that will be used in the key exchange. Next, the client sends the ephemeral public key and a random salt value it has generated to the server along with an HMAC of the message which is computed using a pre-shared key between the client and the server. Upon successful verification of the HMAC, the server generates its own ephemeral private and public key pair. Using the client's ephemeral public key and its own ephemeral private key, the server computes a master key. From this master key, it derives a connection key using the salt value received from the client and a salt value it generates. HKDF (Barker, Chen & Davis, 2018) is used for key derivation. Finally, the server responds to the client with its own ephemeral public key, the signature for the ephemeral public key, a random salt value, and an HMAC (Chen et al., 2023) generated for these messages using the derived key.

Based on the response, the client verifies the server's ephemeral public key. Then, using the server's ephemeral public key and its own ephemeral private key, the client computes the master key. After that, using the salt value it sent to the server along with the salt received from the server, a connection key is derived from the master secret. Finally, using this key, the HMAC of the response message is verified. If all verifications succeed, then the key exchange operation is completed successfully.

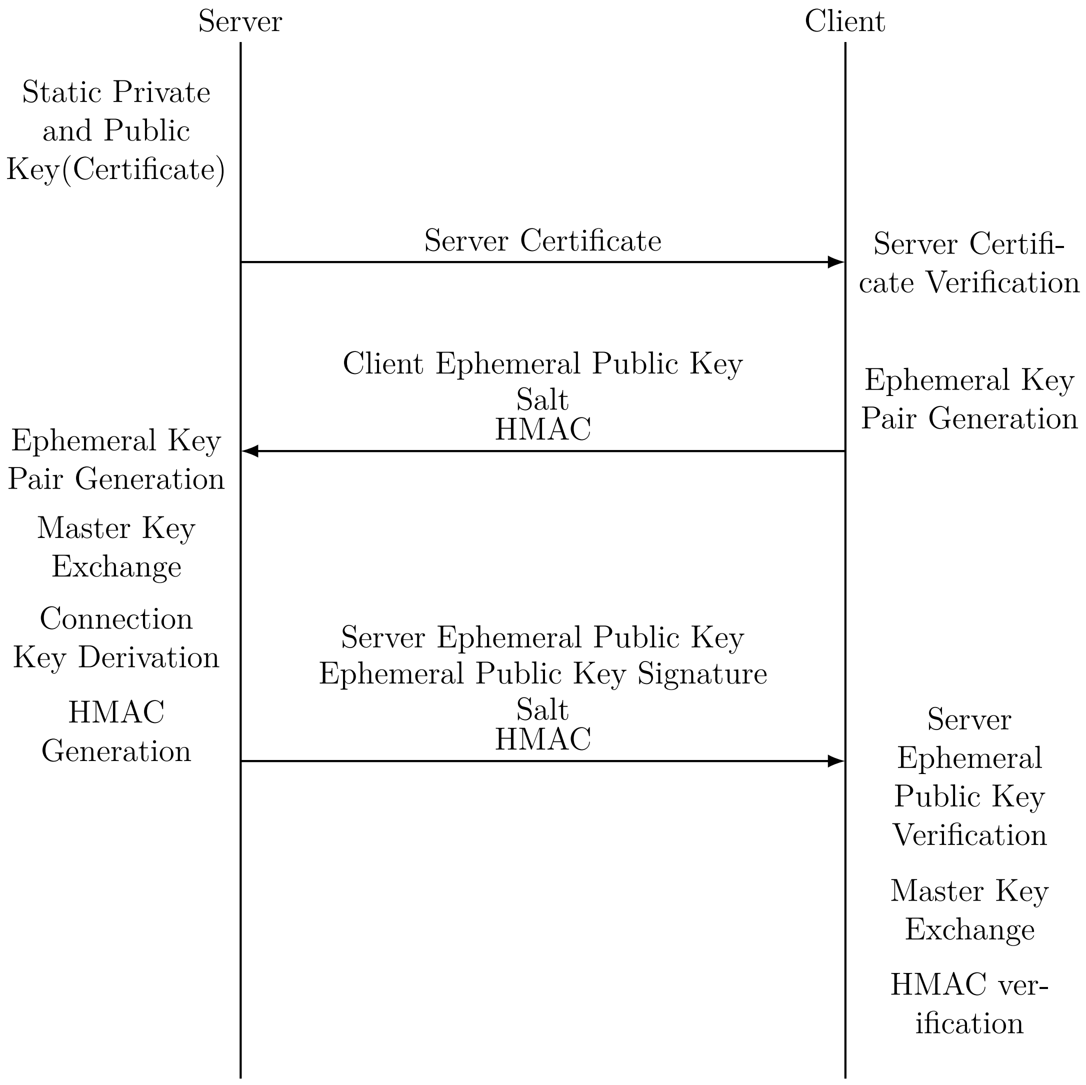

Server
Client
Static Private and Public Key(Certificate)
Server Certificate
Server Certificate Verification
Client Ephemeral Public Key
Salt
HMAC
Ephemeral Key Pair Generation
Ephemeral Key Pair Generation
Master Key Exchange
Connection Key Derivation
HMAC Generation
Server Ephemeral Public Key
Ephemeral Public Key Signature
Salt
HMAC
Server Ephemeral Public Key Verification
Master Key Exchange
HMAC verification

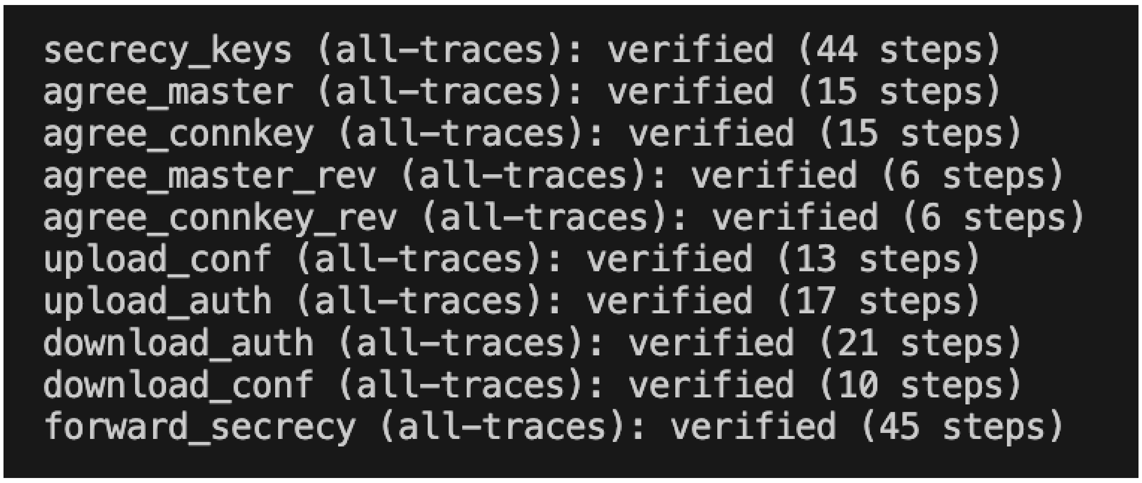
Figure 4.1 Master Key Exchange Between Server and Client

4.2 Formal Verification of the Proposed Key Distribution Protocol

The key exchange protocol that is defined in the previous section is formally verified using the Tamarin prover. The Tamarin prover (Basin et al., 2025) is a model-checking tool that is mainly used for formal verification of security protocols. It has been used to verify protocols such as TLS 1.3 (Cremers, Horvat, Hoyland, Scott & van der Merwe, 2017), and iMessage PQ3 Messaging Protocol of Apple (Linker, Sasse & Basin, 2024). The Tamarin prover supports unbounded protocol sessions, multiset-rewriting rules with states and events, and built-in support for some cryptographic primitives that our key exchange protocol rely on such as Diffie-Hellman, symmetric encryption, signature generation and verification (Basin et al., 2025) (The Tamarin Team, 2024). It features a symbolic Dolev-Yao attacker model (Dolev & Yao, 1983) that controls the network and can intercept, replay and synthesize messages based on known components.

Each step of the protocol is specified using rules in Tamarin. The rules of Tamarin consist of three parts, which are input facts, actions, and output facts. The input facts are consumed upon the transition, and output facts are produced. The actions do not change the state of the model, but they represent the actions taken by the parties, and they are used in property satisfaction. Our model uses the facts to generate random data, store the keys in client and server states, and to write messages to and read messages from the network. The actions are used to mark security related properties, such as declaring terms as a secret, declaring terms as equals, and logging events of sending a message, computing a master key and deriving a connection key. The rules are also used to give attackers more capabilities by revealing the secrets on the network.

After specifying the protocol using the multiset-rewriting rules, the claims for security properties are verified using temporal lemmas over actions. The list of lemmas that are proven, the claim that these lemmas make, and the corresponding security property that they verify can be found in Table 4.1. The security properties that are verified are key confidentiality, data confidentiality, data authenticity, client authenticity, server authenticity, and perfect forward secrecy. To prove key and data confidentiality corresponding lemmas state that if a term is declared as secret or logged in the actions as being sent over the network, then the adversary never knows the term value. For data authentication, corresponding lemmas states that whenever a data is logged in the actions as being received, then it must be logged as being sent by the other party. For server authenticity, the corresponding lemmas states that if there is an action logging that the client completes a key exchange or key derivation, there must be a corresponding action logging that the server also completes the key exchange or key derivation resulting with the same key. For client authenticity, the corresponding lemmas states that if there is an action logging that the server completes a key exchange or key derivation, then there must be actions logging that the client initiated a key exchange or derivation. Finally, for the perfect forward secrecy, the corresponding lemma states that if a key exchange is completed before the long term signature key of the server is exposed, then the exchanged key is never known to the adversary, even after the long term signature key is exposed. The entire theory written in Tamarin can be found in Appendix. Figure 4.2 shows that all lemmas are verified successfully.



```
secrecy_keys (all-traces): verified (44 steps)
agree_master (all-traces): verified (15 steps)
agree_connkey (all-traces): verified (15 steps)
agree_master_rev (all-traces): verified (6 steps)
agree_connkey_rev (all-traces): verified (6 steps)
upload_conf (all-traces): verified (13 steps)
upload_auth (all-traces): verified (17 steps)
download_auth (all-traces): verified (21 steps)
download_conf (all-traces): verified (10 steps)
forward_secrecy (all-traces): verified (45 steps)
```

Figure 4.2 The output of the Tamarin prover on the protocol specification

Table 4.1 List of Proven Lemmas in Tamarin

Lemma Name	Claim	Requirement
secrecy_keys	Claims that the master key, and connection key are never known by the adversary	Key Confidentiality
upload_conf	Claims that the data uploaded from the devices to the server are never known by the adversary	Data Confidentiality
download_conf	Claims that the data downloaded from the server to the client are never known by the adversary	Data Confidentiality
agree_master_rev	Claims that if a master key is computed by the server, the process is initiated by the client	Client Authenticity

Lemma Name	Claim	Requirement
agree_connkey_rev	Claims that if a connection key is computed by the server, the process is initiated by the client	Client Authenticity
agree_master	Claims that if a master key is computed by the client, the server computes the same master key	Server Authenticity
agree_connkey	Claims that if a connection key is computed by the client, the server computes the same connection key	Serve Authenticity
upload_auth	Claims that if data is uploaded to the server, it is received from the client	Data Authenticity
download_auth	Claims that if data is downloaded to the client, it is downloaded from the server	Data Authenticity
forward_secrecy	Claims that if the signature key of the server is exposed, all the master keys and connection keys computed before the exposing are never known to the adversary	Perfect Forward Secrecy

4.3 Storing the Keys and Data

On the server side, the methods by which different keys are stored in memory vary. The private and public key pair to be used in ECDSA operations (Standards for Efficient Cryptography Group, 2010) requires non-volatile memory, because the lifetime of these keys is relatively long. No security measure is required for storing the public key. For permanently storing the private key in memory, however, the TPM (Trusted Computing Group, 2008) will be employed. After the private key is generated, it is sent to the TPM to be stored in non-volatile memory, as depicted in Figure 4.3, and the TPM encrypts this key internally and returns the ciphertext. This ciphertext is then written to disk for storage, and, when the key is needed, it is sent to the TPM for decryption, as shown in Figure 4.4, and the TPM returns the plaintext form of the key. The plaintext version of the key is kept in volatile memory only while it is being processed, and it is deleted from volatile memory once the digital signature generation is finished.

While this key is stored in volatile memory, it is written to a region that has been securely allocated by the *OpenSSL_secure_malloc* (OpenSSL Foundation, Inc., 2016a) function of the *OpenSSL* library. Memory regions allocated with this function do not appear in memory dumps and are inaccessible to any process other than the allocating one. In addition, when memory allocated with this function is freed, the memory cells are not left intact; zeros are written to all cells.

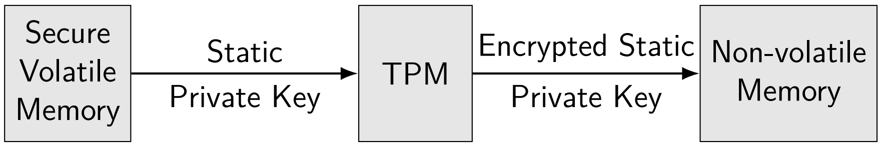


Figure 4.3 Secure Storage of the Private Signature Key

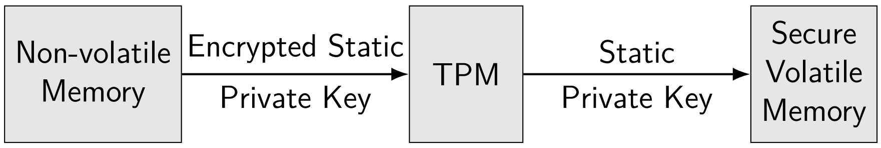


Figure 4.4 Secure Usage of the Private Signature Key

The ephemeral private and public key pair used for the key agreement requires no non-volatile storage because it is used only once and then deleted from memory. Using the *OpenSSL_secure_malloc* function is sufficient for storing the ephemeral private key. Non-volatile storage is required for the master key produced by the key agreement, and this key will also be stored in the TPM, as shown in Figure 4.5. When a connection is established with the client, the plaintext form of the master key obtained from the TPM is written to a region of volatile memory allocated with *OpenSSL_secure_malloc*, as shown in Figure 4.6, and is used there; once the connection key has been generated, the master key is deleted from memory. The connection key does not need non-volatile storage, because it is valid only while an active connection exists and will be used continuously during that period; it will be stored in a region of volatile memory allocated with *OpenSSL_secure_malloc*.

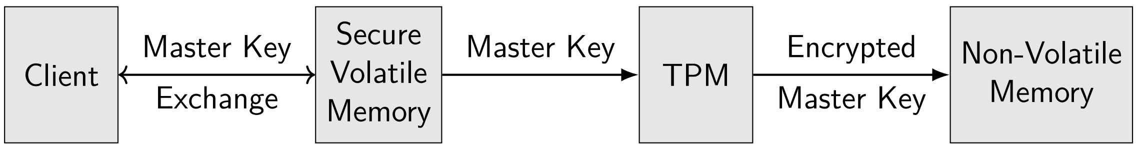


Figure 4.5 Secure Storage of the Master Key

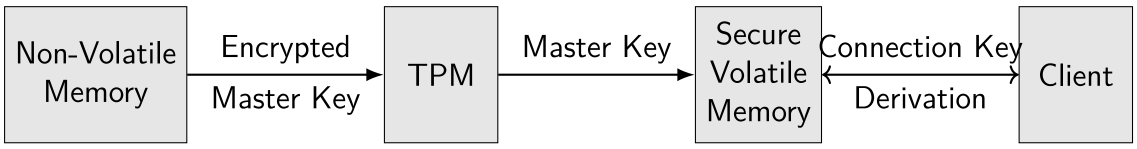


Figure 4.6 Secure Usage of the Master Key

As devices used as clients are not expected to contain a TPM, the master key and connection keys will be written to, and stored in, regions of volatile memory allocated with the *OpenSSL_secure_malloc* function.

Symmetric encryption methods will be used to store the static data —that is, the data collected by the client devices and securely transmitted to the server— and the symmetric key that will be employed to encrypt the stored data will be generated by the server using the *RAND_bytes* function of the *OpenSSL* library (OpenSSL Foundation, Inc., 2020c). Next, that symmetric key is sent to the TPM, its ciphertext is obtained, and this ciphertext is written to non-volatile memory for storage. The storage process of the data is illustrated in Figure 4.7. The data received from the client, encrypted with the connection key, are first decrypted and kept in a securely allocated region of volatile memory. Then, the ciphertext of the symmetric storage key will be sent to the TPM to be decrypted, and the actual key will be stored in a securely allocated region of volatile memory. The data are then re-encrypted with the symmetric storage key and written to non-volatile memory. After that, the plaintext form of the symmetric storage key is deleted from volatile memory.

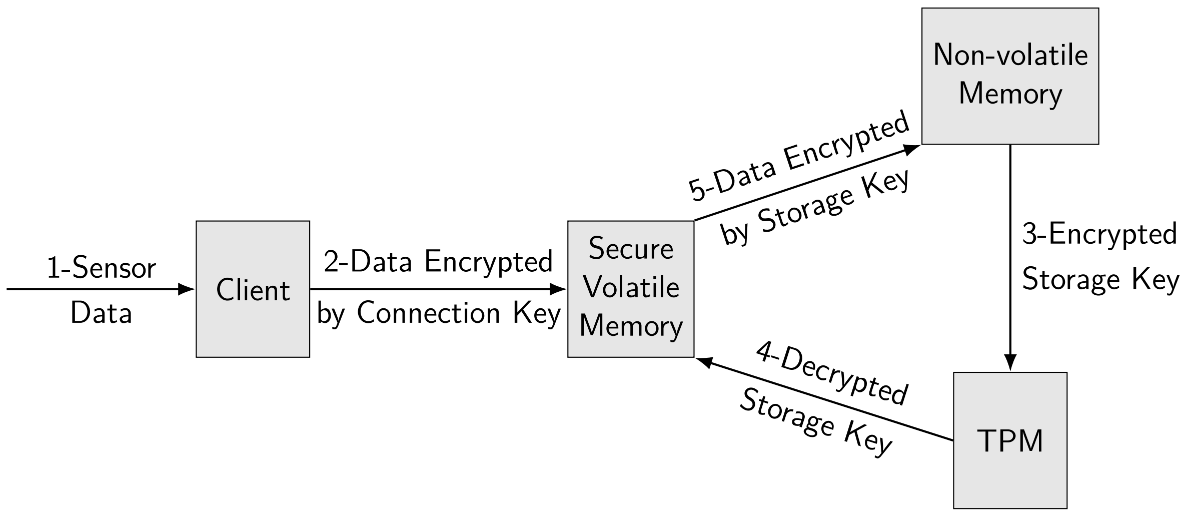


Figure 4.7 Secure Storage of the Static Data

When the static data must be decrypted for use, as shown in Figure 4.8, the symmetric key stored in non-volatile memory as ciphertext is decrypted via the TPM and placed in securely allocated memory. This key is then used to decrypt the required portion of the static data, which is written to securely allocated memory for use, and the plaintext form of the symmetric storage key is erased from volatile memory. Finally, the data is encrypted using the connection key and sent to the client and the plaintext data is removed from volatile memory.

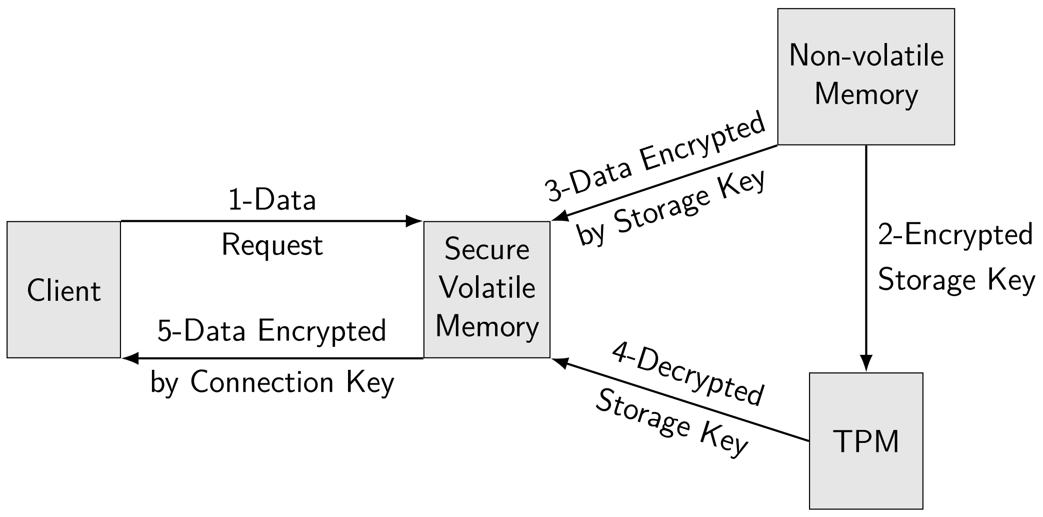


Figure 4.8 Secure Usage of the Static Data

4.4 Implementation Details of Key Management

To demonstrate the usability of the proposed structures and technologies for securing processes such as key exchange, key storage and static data storage, these processes have begun to be developed in appropriate programming languages. In this section, the core application components and the relevant libraries will be mentioned. Also, the performance of the operations are evaluated in this section. The implementation and performance evaluations are carried out on a computer with Intel Core i5 processor running at 1.60 GHz. All applications and the related APIs will be discussed in the next chapter.

The prototype in question has been developed in the C programming language using the *OpenSSL* (OpenSSL Foundation, Inc., 2025) and *tpm2-tss* (tpm2-software, 2024)(Trusted Computing Group, 2025) libraries. The functions of these libraries that are used in the application, and their roles, are listed in Table 4.2.

Table 4.2 Details of Libraries and Functions used in Implementation

Library	Function	Details
OpenSSL	EVP_PKEY_keygen	Generates a key described by the parameters
	X509_sign	Signs a certificate with the key, using a hash function
	RAND_bytes	Generates a random byte array of the requested length
	i2d_PUBKEY	Serializes the public key of a key object
	d2i_PUBKEY	Deserializes a public key into a key object
	EVP_PKEY_derive	Derives a symmetric key from a private and a public key
	EVP_KDF_derive	Derives a symmetric key from another symmetric key
	HMAC_Init_ex HMAC_Update HMAC_Final	Generates an HMAC for the data using a symmetric key and a hash function specified by the parameters
	EVP_EncryptInit_ex EVP_EncryptUpdate EVP_EncryptFinal_ex	Encrypts the data using a symmetric encryption algorithm, key, and IV specified by the parameters
	EVP_DecryptInit_ex EVP_DecryptUpdate EVP_DecryptFinal_ex	Decrypts the ciphertext using a symmetric encryption algorithm, key, and IV specified by the parameters
tpm2-tss	Esys_Initialize	Takes ownership of the TPM for subsequent use
	Esys_CreatePrimary	Creates a primary key whose attributes are set by the parameters
	Esys_EvictControl	Makes a TPM key persistent through an external handle
	Esys_TR_FromTPMPublic	Loads a key to TPM from a persistent handle
	Esys_RSA_Encrypt	Encrypts the data using an RSA key loaded into TPM
	Esys_RSA_Decrypt	Decrypts the ciphertext using an RSA key loaded into TPM

TPM operations have been carried out by using the ESAPI (Enhanced System Application Programming Interface) layer (Trusted Computing Group, 2020) of the *tpm2-tss* library. When the server program starts, the TPM is first initialised with the *Esys_Initialize* (Trusted Computing Group, 2020) function. For this function to complete successfully, the server program must be run with the *sudo* command and the user password must be entered. Then, if the server is being started for the first time, a primary RSA (Rivest-Shamir-Adleman) encryption key is generated with the *Esys_CreatePrimary* (Trusted Computing Group, 2020) function. If the server shuts down for any reason and needs to be restarted, this key is bound to a persistent handle by using the *Esys_EvictControl* (Trusted Computing Group,

2020) function so that the files encrypted with this key can be decrypted. If the server program is closed and run again, the same RSA key becomes accessible to the server with the *Esys_TR_FromTPMPublic* (Trusted Computing Group, 2020) function. In this way encryption and decryption can be carried out with the same RSA key.

The *OpenSSL* library is used first to generate the root key and root certificate that will be employed to authenticate the server's identity. For this purpose, an elliptic curve key is generated with the *EVP_PKEY_keygen* (OpenSSL Foundation, Inc., 2020a) function. This key is then used with the *X509_sign* (OpenSSL Foundation, Inc., 2016b) function to create a self-signed root certificate, and the key and the certificate are written to files for later use. Subsequently, the elliptic curve key that the server will use to sign messages is generated with the *EVP_PKEY_keygen* function, and a CSR (Certificate Signing Request) file is created to generate the certificate that the clients will use to verify these messages. The CSR file is then signed with the root key, and the resulting certificate is stored in a file to be sent to the clients. After a certificate has been created for the signing key, this key is encrypted by the TPM with the primary key generated at program start-up or loaded from the persistent handle, by using the *Esys_RSA_Encrypt* (Trusted Computing Group, 2020) function, and its encrypted form is written to a file. The server generates a symmetric storage key with the *RAND_bytes* (OpenSSL Foundation, Inc., 2020c) function of the *OpenSSL* library to encrypt the messages it receives from the clients and stores in non-volatile memory. This key is also encrypted with the TPM by means of the *Esys_RSA_Encrypt* function, and its encrypted form is written to a file.

When a connection is then established between the server and the client, an elliptic curve key pair is generated on both sides with the *EVP_PKEY_keygen* function, and the public keys are serialised with the *i2d_PUBKEY]* (OpenSSL Foundation, Inc., 2022b) function. After the serialised public keys have been exchanged, both parties compute the master key by using their own private key and the other party's public key, obtained by deserialising with the *d2i_PUBKEY* (OpenSSL Foundation, Inc., 2022b) function, in the *EVP_PKEY_derive* (OpenSSL Foundation, Inc., 2022a) function. The *EVP_KDF_derive* (OpenSSL Foundation, Inc., 2024b) function then derives the connection key from the master key and a salt that is generated randomly by the server and shared with the client. The purpose of using this salt is to enable many different connection keys to be produced from the same master key. To verify that the derived connection key is the same on both the server and the client, the client concatenates the messages exchanged since the connection was established and generates an HMAC by using the *HMAC_Init_ex* (OpenSSL

Foundation, Inc., 2020b), *HMAC_Update* (OpenSSL Foundation, Inc., 2020b) and *HMAC_Final* (OpenSSL Foundation, Inc., 2020b) functions, then sends this HMAC to the server. The server likewise concatenates the messages exchanged since the connection was established and generates an HMAC with the same functions. If the HMAC it has created matches the one received from the client, the identity of the derived connection key is confirmed. When the derivation of the connection key is complete, the master key is encrypted by the TPM with the *Esys_RSA_Encrypt* function for storage in non-volatile memory, and its encrypted form is written to a file.

From this point on, the messages to be sent over the connection are encrypted with the connection key by using the *EVP_EncryptInit_ex* (OpenSSL Foundation, Inc., 2024a), *EVP_EncryptUpdate* (OpenSSL Foundation, Inc., 2024a) and *EVP_EncryptFinal_ex* (OpenSSL Foundation, Inc., 2024a) functions, and are decrypted with the *EVP_DecryptInit_ex* (OpenSSL Foundation, Inc., 2024a), *EVP_DecryptUpdate* (OpenSSL Foundation, Inc., 2024a) and *EVP_DecryptFinal_ex* (OpenSSL Foundation, Inc., 2024a) functions. The server decrypts, with the *Esys_RSA_Decrypt* (Trusted Computing Group, 2025) function in the TPM, the storage key that it wrote to a file in order to encrypt the data it receives from the clients, and uses this key to encrypt the static data. After the static data have been encrypted, the storage key is deleted from volatile memory.

The impact of the TPM is analyzed by recording the time it takes to complete operations in the TPM. Figure 4.9 shows the time it takes to initialize the TPM module and obtain the ownership by the program, to create a 2048-bit primary RSA encryption key, storing the created key in persistent storage of the TPM, and finally loading a persistent key to be used in encryption operations. The results indicate that RSA key generation is the most costly operation while initialization is the least costly operation.

Figures 4.10 and 4.11 show the time taken for encryption and decryption operations with the TPM's primary RSA key across different data sizes. A 2048-bit RSA key can encrypt at most 245 bytes in a single operation; larger inputs must therefore be split into 245-byte chunks. The results show that processing time rises with the number of chunks to encrypt or decrypt, yet stays nearly constant for different overall data sizes when the chunk count is the same. Decryption operations take longer than encryption operations for data of the same size.

Of these tasks, TPM initialization, loading the persistent RSA key, and the encryption and decryption will run most frequently. Because the inputs to encryption and decryption are themselves keys, the data blocks remain small. Even so, these

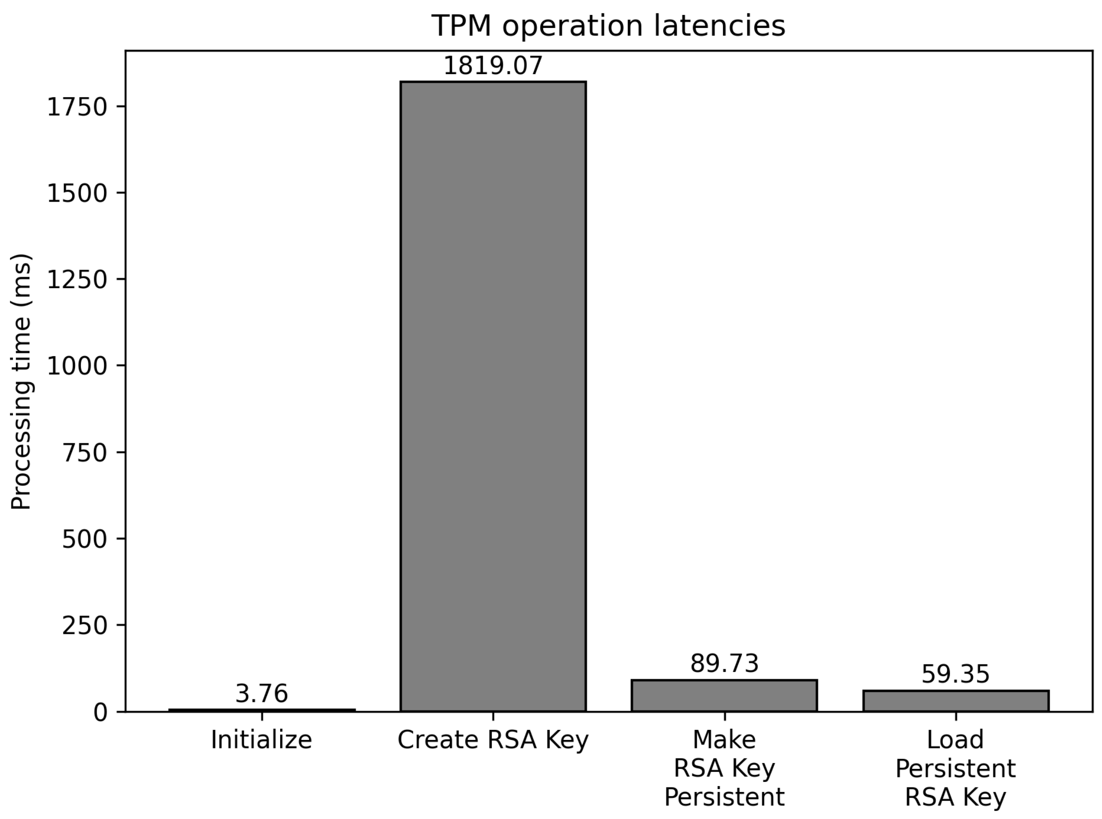


Figure 4.9 Performance of TPM Startup

TPM based operations are more expensive than the purely software based symmetric encryption, message authentication, authenticated encryption, and elliptic-curve cryptosystem operations reported in Chapter 3.

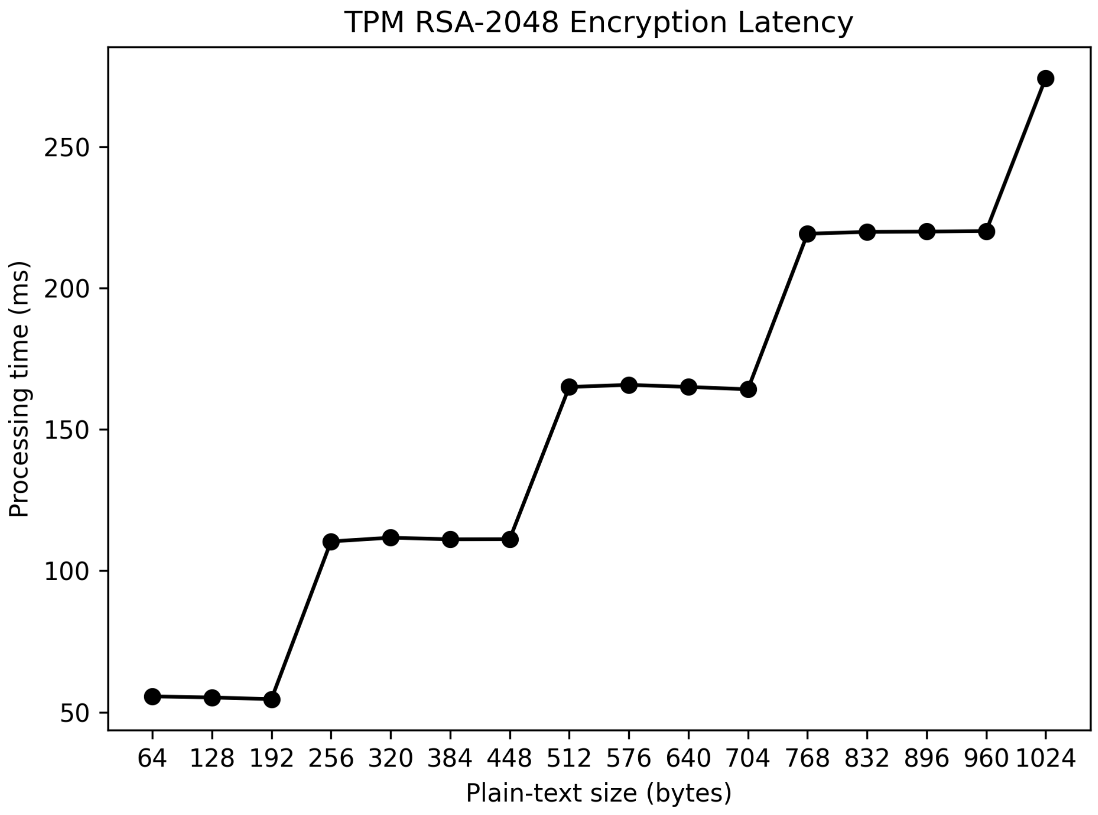


Figure 4.10 Performance of TPM for RSA-2048 Encryption

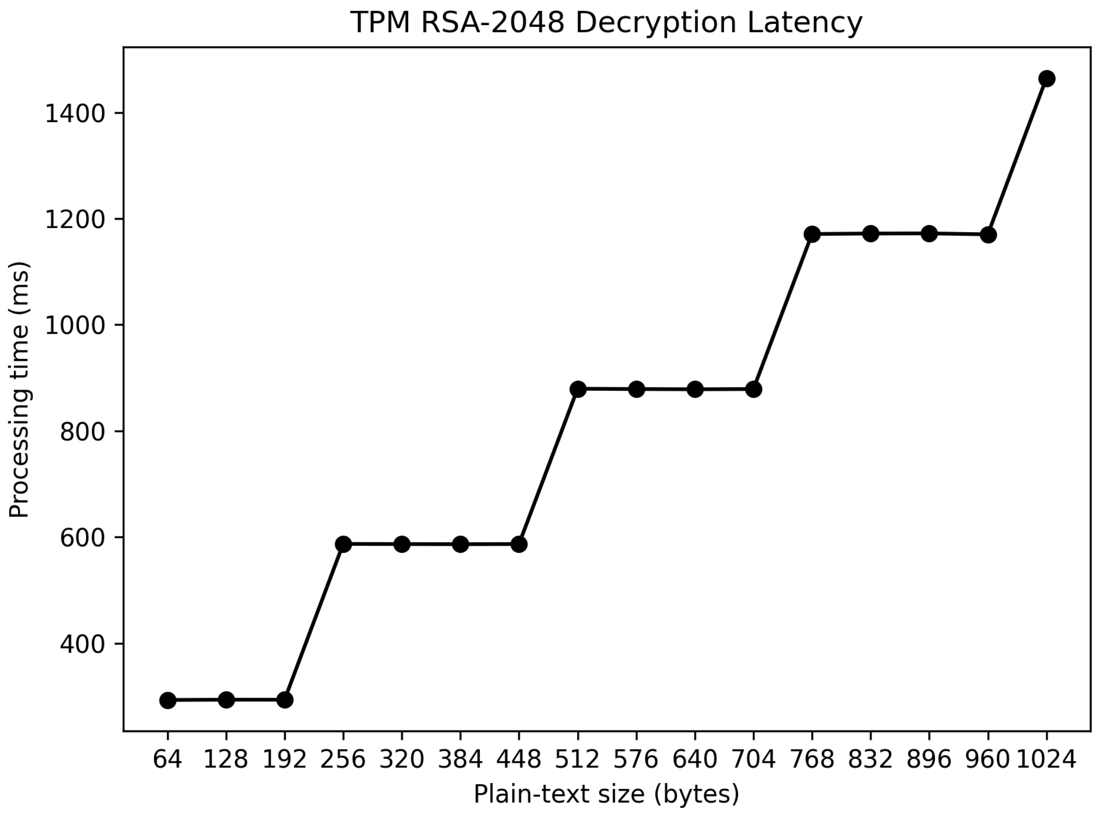


Figure 4.11 Performance of TPM for RSA-2048 Encryption

5. SERVER APPLICATION

Libraries implementing the key management protocol for both the client and server are developed, and integrated into a server-client application. This application runs on the computer described in Chapter 4.3, which features an Intel Core i5 processor running at 1.60 GHz. The operations to be carried out by the server are implemented using *C* programming language where the operations to be carried out by the client are implemented using *Python* programming language. The library developed for the server is integrated with an application developed with *.NET* framework. Some API endpoints are determined for the key agreement operations and when a request arrives these endpoints, appropriate functions from the library are called, and the outputs are sent to the client as response. The cryptoperiods of the keys are also realized in this application, where an expired key is replaced by a new key using necessary requests to specific endpoints. The client library contains functions that sends requests to endpoints in the correct order and use the responses to derive symmetric keys. During operations, if an erroneous response is received from the server indicating either the certificate or the master key is expired, the key agreement operations start over to keep the keys up to date.

5.1 Implementation of Server Library

The encryption, authentication and key agreement functions to be used by the server application are implemented in C programming language using *OpenSSL* (OpenSSL Foundation, Inc., 2025) and *tpm2-tss* (tpm2-software, 2024)(Trusted Computing Group, 2025) libraries. These functions and their descriptions are listed in Table 5.1. A static code analysis is executed on this library implemented in C in order to make sure that it is robust against buffer overflows, improper access controls, and injecting attacks. This analysis is carried out by using the *cppcheck* tool (Marjamäki, 2025). The results presented only style issues, which were fixed subsequently.

Table 5.1 Functions Defined in the Library Implemented in C for the Server

Function	Details
init_OpenSSL	Loads the error codes and algorithms provided by the *OpenSSL* library
generate_EC_key_pair	Generates an elliptic curve key pair
generate_csr	Creates a certificate signing request to be signed by root key to generate the server certificate
save_csr_to_file	Saves the certificate signing request to a file
sign_csr	Generates the server certificate by signing the certificate signing request by root key
save_cert_to_file	Saves the server certificate to a file
base64_encode	Encodes an output to base 64 for it to be used by different programs
base64_decode	Decodes an input that is encoded to base 64
sign_message	Generates a signature for a message using the static private key
compute_ecdh_shared_secret	Computes a master key by ECDH using ephemeral private key of the server and ephemeral public key of the client
keyExchange	Takes the client ephemeral public key and client salt as an input to create a master key using other functions, and returns the server ephemeral public key with its signature, a salt, and a HMAC as a response to the client
generate_random_salt	Generates a salt with a length specified in the parameters
derive_from_master_secret	Derives a connection key using a master key and a salt
create_hmac	Generates a HMAC for a message using a key
generate_symmetric_key	Creates a symmetric key with the desired length
saveCiphertextToFile	Saves a ciphertext to a file
readCiphertextFromFile	Reads a ciphertext saved in a file to memory

Function	Details
createCertificate	Generates a static private signature generation key and static public signature verification key (certificate) and saves them to a file
sendCert	Reads the server certificate from a file and returns it in *PEM* format to respond to the client
OpenSSLSymmetricEncrypt	Encrypts a message with a key and an IV taken as input using 256 bit AES-CBC
OpenSSLSymmetricDecrypt	Decrypts a message with a key and an IV taken as input using 256 bit AES-CBC
receiveData	Takes the ciphertext, salt, and hmac for the message as input, derives a decryption key from previously computed master key using the salt, and decrypts the ciphertext, and finally stores the received data by encrypting with the storage key
sendData	Reads the stored data, decrypts it using storage key, derives a connection key from previously computed master key using a newly generated salt, and returns the encrypted message as a response to the client
TPMLoadPersistentKey	Loads the persistent RSA key using a handle
TPMEncrypt	Encrypts a message using a key in TPM
TPMDecrypt	Decrypts a ciphertext using a key in TPM

The server application is developed using *.NET* framework and C# programming language. As a result, the functions in Table 5.1 cannot be used directly. To be able to use these functions, the source code must be compiled into shared objects. Then, the functions defined in the shared objects can be called from a C# class. To accommodate the need for new functionality, some new functions are defined in this class, which are listed in Table 5.2 along with their descriptions.

Table 5.2 Functions Used by the Server Implemented in C#

Function	Details
createCertificateLib	Calls the *createCertificate* function from C library
createCertificate	Calls the *createCertificateLib* function and records the time of server certificate creation
sendCertLib	Calls the *sendCert* function from C library
sendCert	Called upon certificate request from a client, checks the certificate creation time and if it is expired, creates another certificate; if it is not expired, generates the response message to the client
keyExchangeLib	Calls the *keyExchange* function from C library
keyExchange	Arranges the return type of the *keyExchangeLib* function before calling it
APIkeyExchangeHandler	Called upon client request, checks the certificate creation time and if it is expired, generates the response message with an error code; if it is not expired, calls the *keyExchange* function and returns the result after updating the master key creation time
receiveDataLib	Calls the *receiveData* function from C library
receiveData	Called upon receiving a message, decrypts the incoming message, and stores it securely by calling the *encryptLib* function
sendDataLib	Calls the *sendData* function from C library
sendData	Called upon client request, reads data from the storage, encrypts it by calling the *decryptLib* function, and returns the ciphertext as a response to the client

Among these functions, *APIkeyExchangeHandler* returns a specific error code in case the server certificate is expried. If the clients receive this error code, they are directed to restart the key exchange protocol to update the certificate and master key. On the other hand, *receiveData* and *sendData* respond with a different error code in case the master key attempted to be used is expired, which directs the client to carry out a new key exchange operation.

The communication between the server and the client is established using HTTP (Hypertext Transfer Protocol). The endpoints which are listened by the server application that are responsible for the key exchange and data transfer operations are:

- /crypto/cert
- /crypto/keyExchange
- /crypto/upload
- /crypto/download

After the user connects to the server, to complete the key exchange protocol, first, a *GET* request is sent to */crypto/cert* endpoint, and the *sendCert* function is triggered in the server. If the certificate does not exist or it is expired, the server creates a certificate and sends it to the client as a response. If the certificate exists and it is not expired, then the server responds with the existing certificate.

After receiving the server certificate, the client sends the ephemeral public key with a random salt to the server by a *POST* request to */crypto/keyExchange* endpoint. This request has the HMAC of the message body in the headers, which is computed by the pre-shared key between the client and the server. Upon receiving the request, server application verifies this HMAC. If it is not verified, then it returns with a special error code. Otherwise, this request runs the *APIkeyExchangeHandler* function which checks the validity of the certificate. If it is expired, it informs the client about this by responding with an error code. If it is valid, then the server computes a master key using the received ephemeral public key of the client along with its ephemeral private key. After that, a connection key is derived from the master key using the received salt along with a new salt generated in the server. Finally, the server responds to the client with its ephemeral public key with a signature of it, the salt generated by the server, and an HMAC for the response message to verify that key exchange is successful.

After a successful key exchange, the clients can upload their data to the server, or request to download their data from the server. In order to upload data to the server, the client must generate a salt, derive a connection key from the master key using that salt, encrypt the message, and generate an HMAC for the message, and send these in a *POST* request to */crypto/upload* endpoint which runs the *receiveData* function. As a result, the server decrypts the message, verifies the HMAC, and if it is successful, stores the message after encrypting it with storage key. In order to download data from the server, the client must send a *GET* request to */crypto/-*

download endpoint which runs the *sendData* function. The server reads the client's data by decrypting it with storage key and derives a connection key from the master key using a newly generated salt to encrypt the data. Finally, it responds with the ciphertext, the salt to be used for connection key derivation, and an HMAC of the data.

The static code analysis of the server application is done using the *.NET*'s built-in *RoslynSecurityGuard* package (Arteau, 2025), and *CodeQL* tool (GitHub Inc., 2025). *CodeQL* did not identify any issues with the application. On the other hand, the *RoslynSecurityGuard* found that the */crypto/keyExchange* and */crypto/upload* endpoints are vulnerable to CSRF (Cross-Site Request Forgery) attacks. However, this vulnerability is only a problem when the application is accessed through a web browser, and the credentials are stored in the cookies (Hasan & Anderson, 2024). Therefore, this vulnerability is marked as a false positive since this application is not accessed through web browsers, but from IoT devices only.

5.2 Implementation of Client Library

The functions that will be used by the client in both the key-agreement and data-encryption operations while communicating with the server are implemented in the Python programming language by means of the *cryptohraphy* library. These functions are listed in Table 5.3. For the static code analysis, the *bandit* (Anonymous, 2025b), *ruff* (Astral, 2025), and *pylint* (Anonymous, 2025a) tools are used. The results only included issues regarding style and refactoring, and no security issues were detected.

In order to connect to the server, client application runs the *keyExchange* function, and within this function an HTTP *GET* request is first sent to the */crypto/cert* endpoint. In response, the server certificate is obtained, and this certificate is verified by means of the root certificate. Next, an elliptic curve ephemeral key-agreement key pair is generated along with a random salt. The public key of this pair and the salt is sent to the server in a *POST* request to the */crcrypto/keyExchange* endpoint by attaching the HMAC of the message body to the headers which is computed using the pre-shared key between the client and the server. In return, the server's ephemeral public key-agreement key, the digital signature belonging to this key, a salt, and an HMAC are received. This digital signature is verified by using the server certificate. Then a master session key is produced by using the client's ephemeral private key and the server's ephemeral public key. Finally, a connection key is derived from master key using the salt generated by the client and server, and the received HMAC is verified.

After a successful key exchange, if the client wants to send a data to the server, it will call the *sendData* function. This function will generate a salt, derive a connection key from master key using the salt, encrypt the data, and generate an HMAC of the data. Then, sends the ciphertext, the salt, and the HMAC to the server with a *POST* request to the */crypto/upload* endpoint.

On the other hand, if the client wants to read data from the server, it will call the *receiveData* function. This function will send a *GET* request to */crypto/download* endpoint. As a response, it will obtain a ciphertext, a salt, and an HMAC. Using the salt, it can generate a connection key from master key, and use it to decrypt the ciphertext and verify the HMAC.

Table 5.3 Functions Used by the Client Implemented in Python

Function	Details
load_certificate_from_file	Loads the elliptic curve root certificate stored in PEM (Privacy Enhanced Mail) or DER (Distinguished Encoding Rules) format to memory
verify_certificate_chain	Verifies the server certificate using the root certificate
verify_signature	Verifies a given signature using the server certificate
encrypt_data	Encrypts a message using the connection key
decrypt_data	Decrypts a ciphertext using the connection key
keyExchange	To establish key agreement with the server, sends HTTP requests to two endpoints
sendData	Derives a connection key, encrypts the message and sends it to the server by sending requests to appropriate endpoints
receiveData	Derives a connection key and receives data from the server by sending requests to appropriate endpoints, and decrypts the incoming message

5.3 Testing Methods for Application Performance

The performance of the server application is measured in term of the time taken to complete operations of each endpoint that is responsible for the key exchange process. Three different measurements are recorded for each endpoint: The time it takes to carry out operations related to the endpoint in the client including the request itself, the time it takes to send the request and receive the response without any additional operations, and the time it takes to carry out operations in the server when a request arrives to an endpoint. The duration of opeartions are recorded with the following settings for each endpoint:

- **/crypto/cert:** The certificates are generated on three different curves, *P256*, *P384*, and *P521*. The certificates are used for signature verification in each operation that follows.
- **/crypto/keyExchange:** The ECDH operations are carried out for ephemeral keys on the curves *P256*, *P384*, and *P521*. The durations are recorded for all combinations of certificate and ephemeral key size.
- **/crypto/upload:** Data of various sizes are sent to the server after HMAC generation and encryption.
- **/crypto/download:** Data of various sizes are received from the server. The data is decrypted, and the corresponding HMAC is verified.

For each of test settings in key agreement operations, the processes are repeated 1000 times, and the means of these samples are reported. Elapsed times for each operation are measured in three parts. First is the duration of the operations on only the client that does not include the time it takes for sending the request and receiving a response. Second is the time spent on the network without any operation, neither on the client nor the server. Third is the time it takes on the server to receive a request and return a response. For the upload and download operations, data with size ranging from 50,000 bytes to 3,000,000 bytes are sent to the server, and received from the server respectively. For each data size, the processes are repeated 250 times, and the means of these samples are reported.

5.4 Test Results for Application Performance

The performance of the key exchange operation is reported for two endpoints, */crypto/cert* and */crypto/keyExchange*, separately in terms of time spent for both requests. For each of the endpoints on different devices, a bar chart is presented that includes timings of the client, network, and server separately. Also, performance of certificate retrieval is measured for three different curves, *P256*, *P384*, and *P521*. Performance of key exchange operation is measured for each combination of certificate size, and ephemeral key size. The throughput of upload and download operations on each device can be found in Figure 5.1 and 5.2 respectively. These results show that, the throughput of upload and download operations are heavily dependent on the size of the data that is being transferred. For small data, most of the time is spent by the TPM as it is the most time consuming part of the protocol which creates an equal overhead for all data sizes. Therefore, as the network latency dominates the time it takes to upload data, the throughput increases. The increase in the throughput diminishes as data size increases and it is because the change in the significance of the overhead reduces as it is dominated by the network latency. This makes the protocol more useful for applications where data is transferred from clients to servers or from servers to clients in bulk.

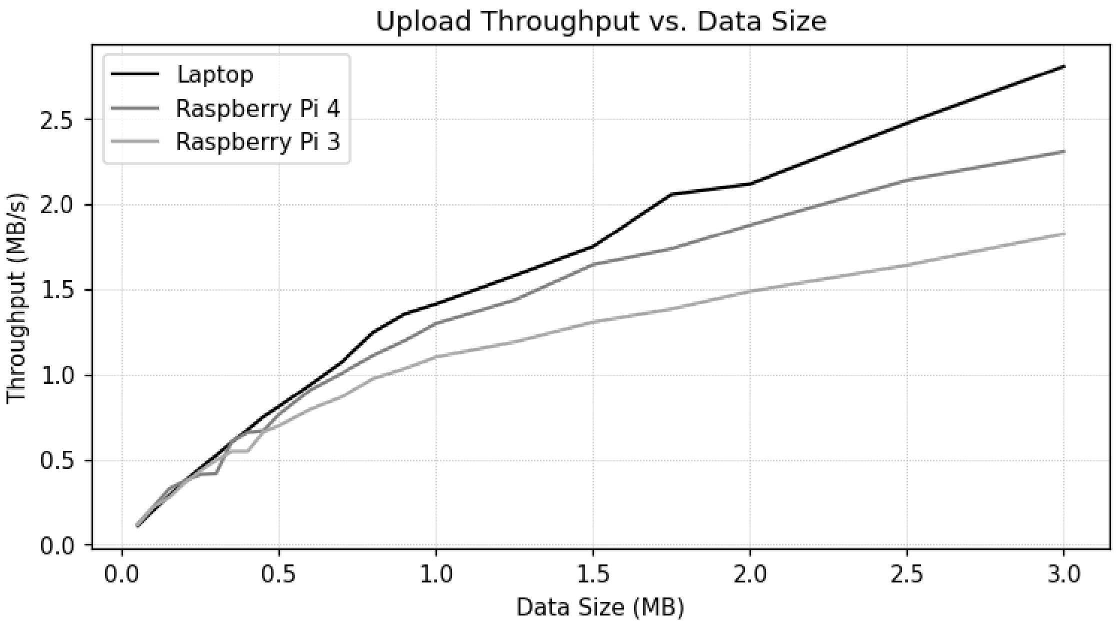


Figure 5.1 Upload throughput with respect to data size

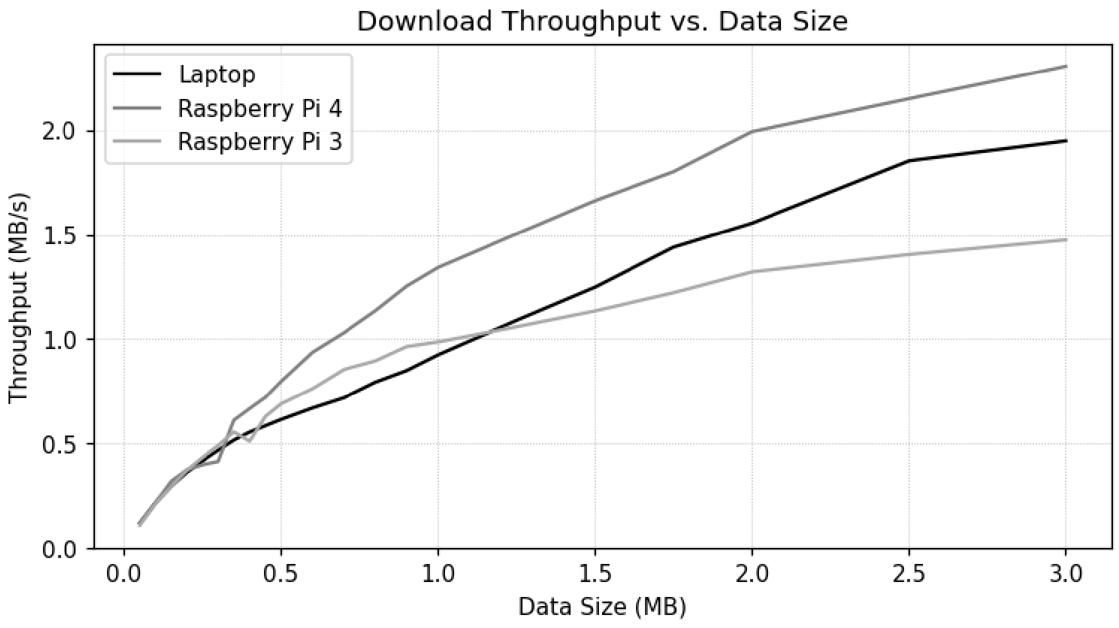


Figure 5.2 Download throughput with respect to data size

5.4.1 Test Results on Laptop

When the client is the laptop, the performance results of the key agreement operations for each endpoint in the server application, */crypto/cert* and */crypto/keyExchange*, can be seen in Figures 5.3 and 5.4 respectively. Time spent to acquire the server certificate by the client is small compared to the rest of the operations, and it is heavily dominated by the duration of the messages being transported on the network. The certificate is generated once for each curve, and is used for the remaining runs. Therefore, the time spent on the server is only for reading the file, and on the client, the certificate is verified by the root certificate. Hence, the duration of actual processes carried out by the client and the server are not significant. As a result, there is no notable difference of performance between different curves.

For computing the master key by ECDH and verifying it, the majority of the time is spent by the server as depicted in Figure 5.4. The effect of the client on the total time elapsed for this operation is minimal. The change in the size of the curve of certificate and the ephemeral keys have a slight impact on the performance, and it is only visible in the client processes.

The time elapsed during the client receiving the server certificate is mainly on the network, and for the rest of the operations, the majority of the time is spent on the server. This is because for the server to send the certificate to the client, it does not carry out any cryptographic operations, but just reads a file after the first request. In contrast, the master key exchange, connection key derivation, and the HMAC

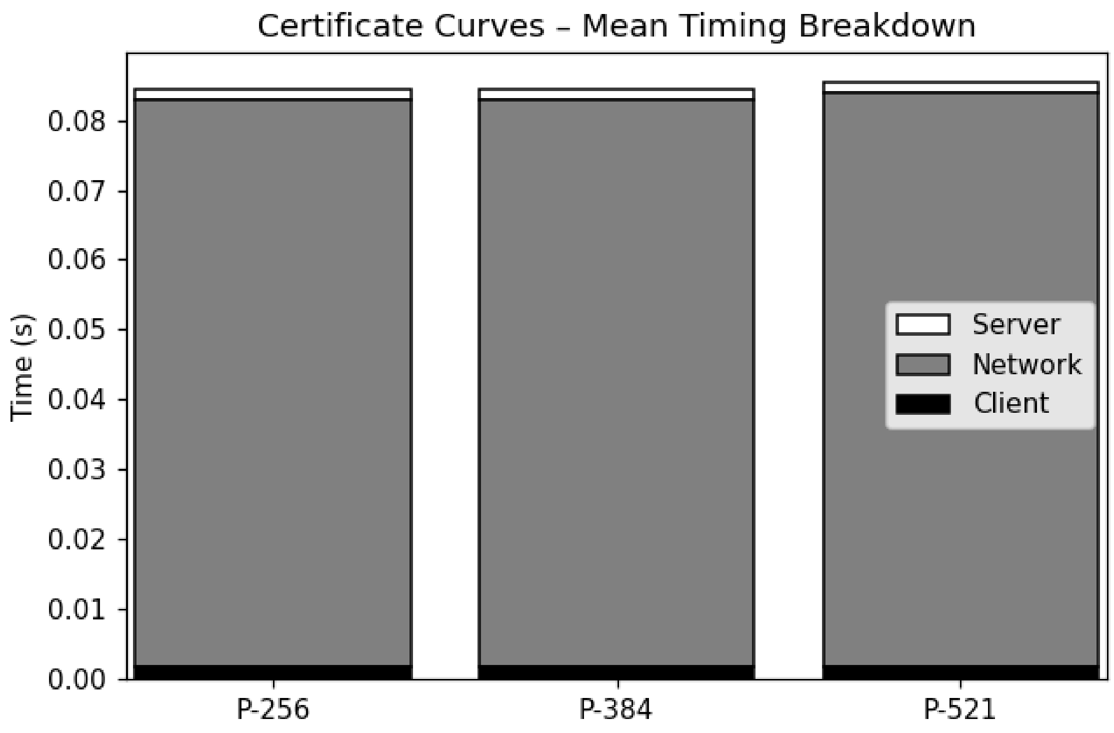


Figure 5.3 Time elapsed for acquiring certificate of different sizes

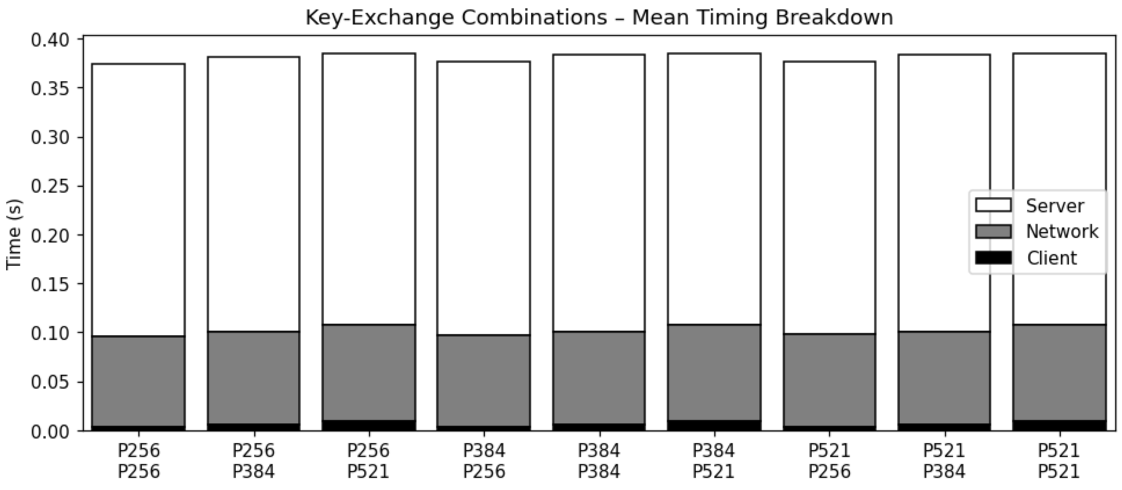


Figure 5.4 Time elapsed for computing master key using different sizes of elliptic curve key and certificate

verification operations require the server to use cryptographic operations, especially the TPM. The time difference between the server and the client is also the result of using TPM on the server but not on the client.

5.4.2 Test Results on Raspberry Pi 4

When the client is Rapsberry Pi 4, the performance results of the key agreement operations for each endpoint in the server application, */crypto/cert* and */crypto/keyExchange*, can be seen in Figures 5.5 and 5.6 respectively. The results for acquiring the certificate present similar properties to that of the laptop, as the time spent by the client and the server is insignificant compared to the time spent for communication on the network and the change in curve size does not affect the performance.

For the key exchange operation, the performance of the client processes have a slightly higher effect compared to that of the laptop. But the difference is not large enough to have a strong impact on the total elapsed time for each operation. Also, the results following the use of TPM on the difference in length of time for receiving the certificate and key exchange duration is similar to that of laptop.

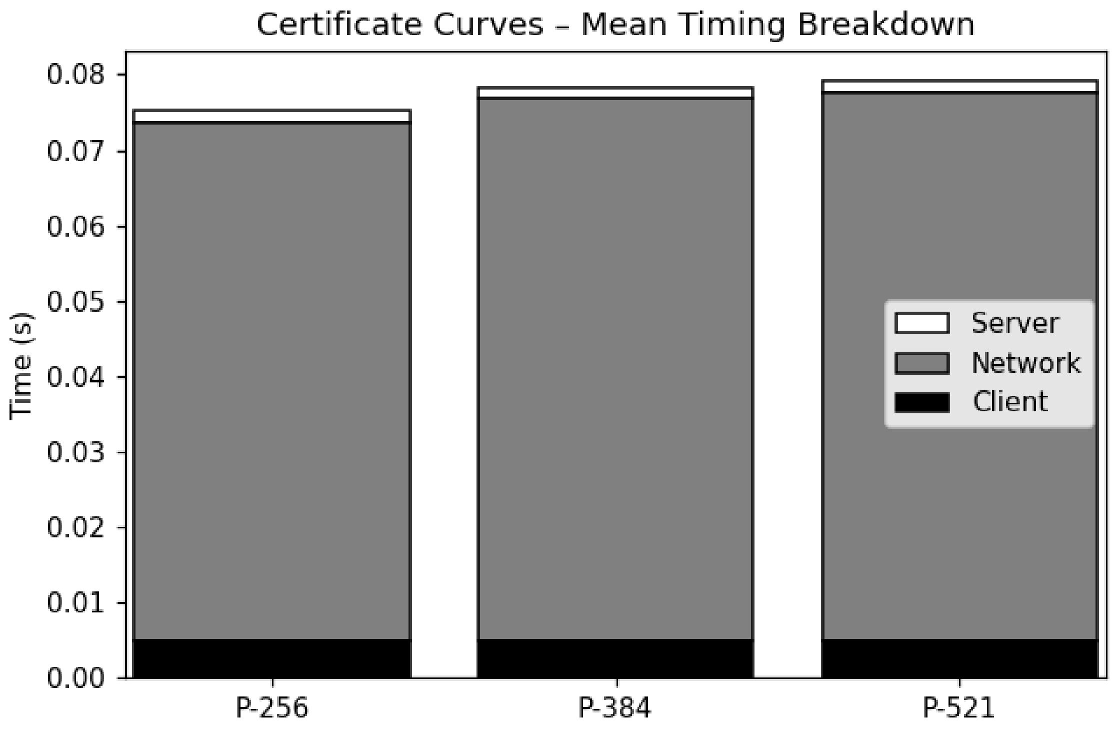


Figure 5.5 Time elapsed for acquiring certificate of different sizes

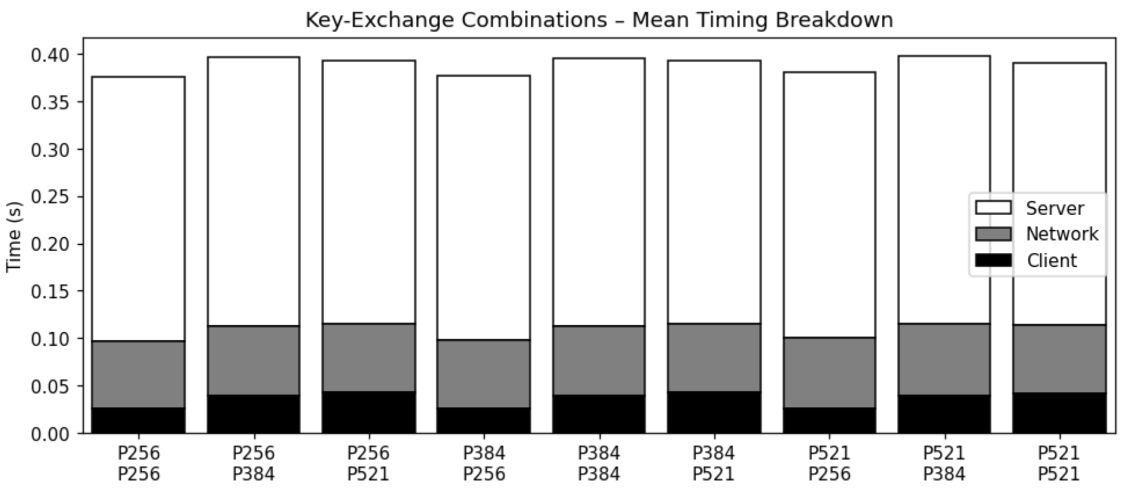


Figure 5.6 Time elapsed for computing master key using different sizes of elliptic curve key and certificate

5.4.3 Test Results on Raspberry Pi 3

When the client is Raspberry Pi 3, the performance results of the key agreement operations for each endpoint in the server application, */crypto/cert* and */crypto/keyExchange*, can be seen in Figures 5.7 and 5.8 respectively. The certificate retrieval presents similar results to that of the laptop and Raspberry Pi 4, as the time spent by the client and the server is insignificant compared to the time spent for communication on the network.

For the key exchange operation, the performance of the client processes have a slightly higher effect compared to that of the laptop, which is very similar to the performance on Raspberry Pi 4. But the difference is not large enough to have a strong impact on the total elapsed time for each operation. The strongest impact on the total duration of the key exchange operation is the network latency for Raspberry Pi 3. The duration of the operation fluctuates as the curve of certificate and ephemeral key changes. However, this change is not due to the difference in server or client performance. It is dictated by the time spent on network. Also, the results following the use of TPM on the difference in length of time for receiving the certificate and key exchange duration is similar to that of laptop and Raspberry Pi 4.

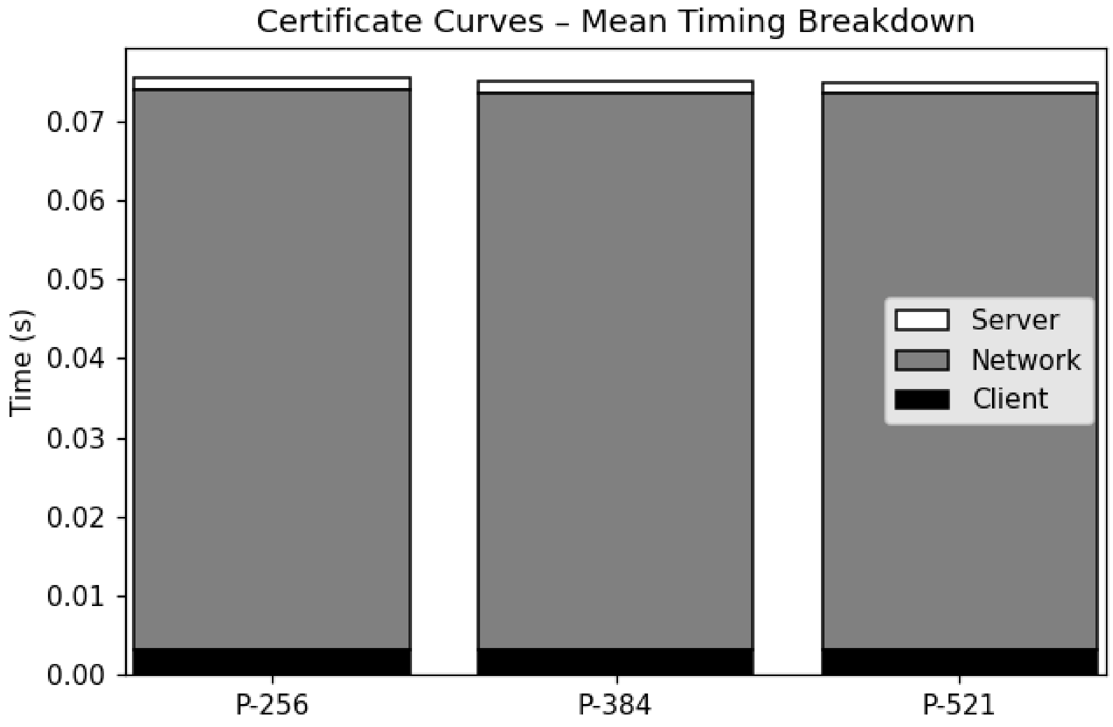


Figure 5.7 Time elapsed for acquiring certificate of different sizes

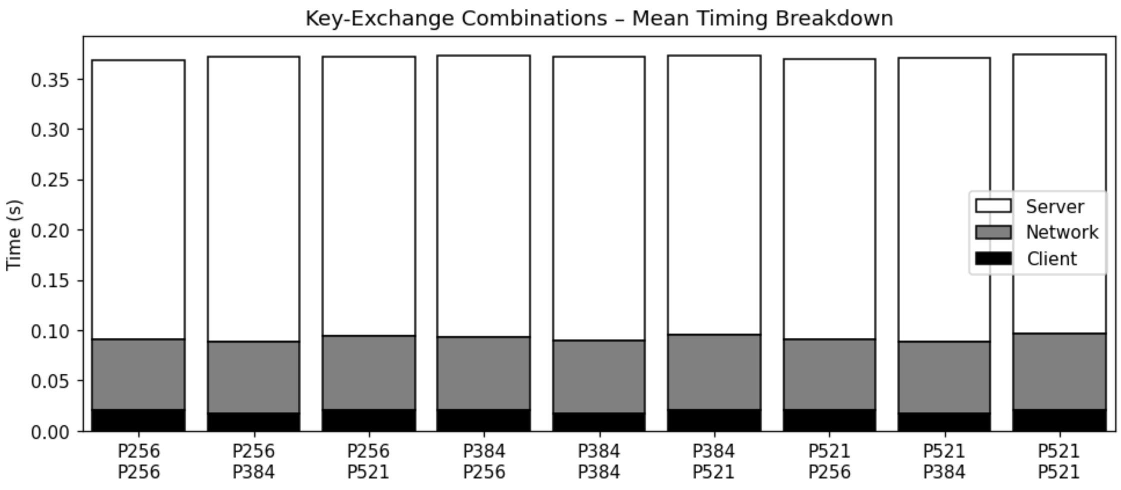


Figure 5.8 Time elapsed for computing master key using different sizes of elliptic curve key and certificate

6. CONCLUSION

With the emergence of smart cities, data produced by IoT devices deployed are collected and processed for various services. Confidentiality, integrity and authenticity of the data must be guaranteed while being transferred from devices to the server and for the still data stored in databases. This thesis presented the design, implementation, formal verification via Tamarin prover, and performance evaluation of a key management framework that can be deployed on resource-constrained IoT devices and achieves the fundamental goals of confidentiality, integrity, authenticity, and forward secrecy.

This work comprised three stages. First, suitable cryptographic primitives were selected and benchmarked on representative hardware. Second, an ECC-based key agreement and key storage mechanism was designed and developed, incorporating a server-side TPM, with its performance subsequently measured. Finally, a proof-of-concept client–server application was built, and its performance was evaluated in detail.

Benchmarks for cryptographic primitives are carried out using two different libraries, *pycryptodome* and *cryptography*. Based on the results, one library is not faster than the other with significant differences for symmetric key operations. On the other hand, the performance of *cryptography* library is significantly better than the performance of *pycryptodome* for ECC operations. Linear increase in duration of the operations is observed with increasing input sizes for symmetric key operations while the impact of input size on ECC operations was not significant.

The key exchange operations are compiled in a library, and this library is used in a server application. These operations are performed in four steps, which are implemented as four requests from client to four endpoints at the server side. Overall, this practical deployment showed approximately 0.45 - 0.6 seconds end-to-end delay that is mostly dominated by TPM operations. However, one should note that this time-consuming operation is to be performed once in a cryptoperiod of a symmetric master key, which typically ranges from one day to one week as given in Table 2.1.

The upload and download speeds can exceed 2.5 MB per second and 2.0 MB per second on large data respectively. These findings confirm that an ECC based key-management architecture supported by a TPM can serve as a practical, efficient, and secure foundation for IoT deployments in smart-city infrastructures.

BIBLIOGRAPHY

Akram, M., Barker, W. C., Clatterbuck, R., Dodson, D., Everhart, B., Gilbert, J., Haag, W., Johnson, B., Kapasouris, A., Lam, D., Pleasant, B., Raguso, M., Souppaya, M., Symington, S., Turner, P., & Wilson, C. (2020). Securing web transactions tls server certificate management volume b: Security risks and recommended best practices. Technical Report NIST Special Publication (SP) 1800-16B, National Institute of Standards and Technology, Gaithersburg, MD.

Alaba, F. A., Othman, M., Hashem, I. A. T., & Alotaibi, F. (2017). Internet of things security: A survey. *Journal of Network and Computer Applications*, *88*, 10–28.

Anonymous (2025a). Pylint 3.3.7 documentation. https://pylint.readthedocs.io/en/stable/. Accessed 7 Aug 2025.

Anonymous (2025b). Welcome to bandit. https://bandit.readthedocs.io/en/latest/. Accessed 7 Aug 2025.

Anonymous (2025c). Welcome to pyca/cryptography. https://cryptography.io/en/latest/. [Online; accessed 18-June-2025].

Anonymous (2025d). Welcome to pycryptodome's documentation. https://pycryptodome.readthedocs.io/en/latest/index.html. [Online; accessed 18-June-2025].

Arteau, P. (2025). Roslynsecurityguard 2.3.0. https://www.nuget.org/packages/RoslynSecurityGuard. Accessed 7 Aug 2025.

Astral (2025). Ruff. https://docs.astral.sh/ruff/. Accessed 7 Aug 2025.

Barker, E. (2020). Recommendation for key management: Part 1 - general. Technical Report NIST Special Publication (SP) 800-57, Part 1, Rev. 4, National Institute of Standards and Technology, Gaithersburg, MD.

Barker, E., Chen, L., & Davis, R. (2018). Recommendation for key-derivation methods in key-establishment schemes. Technical Report NIST Special Publication (SP) 800-56C, Rev. 2, National Institute of Standards and Technology, Gaithersburg, MD.

Barker, E., Chen, L., Roginsky, A., Vassilev, A., & Davis, R. (2018). Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography. Technical Report NIST Special Publication (SP) 800-56A, Rev. 3, National Institute of Standards and Technology, Gaithersburg, MD.

Basin, D., Cremers, C., Dreier, J., & Sasse, R. (2025). Modelingand analyzing security protocols with tamarin: A comprehensive guide. In *CAV*.

Boyd, C. & Gellert, K. (2019). A modern view on forward security. Cryptology ePrint Archive, Paper 2019/1362.

Chen, L., Moody, D., Randall, K., Regenscheid, A., & Robinson, A. (2023). Recommendations for discrete logarithm-based cryptography: Elliptic curve domain parameters. Technical Report NIST Special Publication (SP) 800-186, National Institute of Standards and Technology, Gaithersburg, MD.

Cremers, C., Horvat, M., Hoyland, J., Scott, S., & van der Merwe, T. (2017). A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (pp. 1773–1788)., New York, NY, USA. Association for Computing Machinery.

Dolev, D. & Yao, A. C. (1983). On the security of public key protocols. *IEEE Transactions on Information Theory*.

Dworkin, M. (2001). Recommendation for block cipher modes of operation methods and techniques. Technical Report NIST Special Publication (SP) 800-38A, National Institute of Standards and Technology, Gaithersburg, MD.

Dworkin, M. (2004). Recommendation for block cipher modes of operation: The ccm mode for authentication and confidentiality. Technical Report NIST Special Publication (SP) 800-38C, Errata update July 20, 2007, National Institute of Standards and Technology, Gaithersburg, MD.

Dworkin, M. (2007). Recommendation for block cipher modes of operation: Galois/-counter mode (gcm) and gmac. Technical Report NIST Special Publication (SP) 800-38D, National Institute of Standards and Technology, Gaithersburg, MD.

GitHub Inc. (2025). Codeql. https://codeql.github.com/. Accessed 7 Aug 2025.

Hasan, F. & Anderson, R. (2024). Prevent cross-site request forgery (xsrf/csrf) attacks in asp.net core. https://learn.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-9.0. Accessed 7 Aug 2025.

Linker, F., Sasse, R., & Basin, D. (2024). A formal analysis of apple's iMessage PQ3 protocol. Cryptology ePrint Archive, Paper 2024/1395.

Marjamäki, D. (2025). Cppcheck: A tool for static c/c++ code analysis. https://cppcheck.sourceforge.io/. Accessed 7 Aug 2025.

Miller, C. & Valasek, C. (2015). Remote exploitation of an unaltered passenger vehicle. Technical white paper, IOActive, Seattle, WA.

National Institute of Standards and Technology (2008). The keyed-hash message authentication code (hmac). Technical Report NIST Federal Information Processing Standards Publication (FIPS) 198-1, National Institute of Standards and Technology, Gaithersburg, MD.

National Institute of Standards and Technology (2016). Cryptographic algorithm validation program. https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program. [Online; accessed 18-June-2025].

National Institute of Standards and Technology (2023). Digital signature standard (dss). Technical Report NIST Federal Information Processing Standards

Publication (FIPS) 186-5, National Institute of Standards and Technology, Department of Commerce, Washington, D.C.

National Institute of Standards and Technology (n.d.a). Keyed-hash message authentication code (hmac) using sha3-256. https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/examples/hmac_sha3-256.pdf. [Online; accessed 18-June-2025].

National Institute of Standards and Technology (n.d.b). Keyed-hash message authentication code (hmac) using sha3-384. https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/examples/hmac_sha3-384.pdf. [Online; accessed 18-June-2025].

National Institute of Standards and Technology (n.d.c). Keyed-hash message authentication code (hmac) using sha3-512. https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/examples/hmac_sha3-512.pdf. [Online; accessed 18-June-2025].

OpenSSL Foundation, Inc. (2016a). Openssl_secure_malloc. https://www.openssl.org/docs/man1.1.1/man3/OPENSSL_secure_malloc.html. [Online; accessed 18-June-2025].

OpenSSL Foundation, Inc. (2016b). X509_sign - openssl documentation. https://docs.openssl.org/1.1.1/man3/X509_sign/. [Online; accessed 23-June-2025].

OpenSSL Foundation, Inc. (2020a). Evp_pkey_keygen - openssl documentation. https://docs.openssl.org/1.1.1/man3/EVP_PKEY_keygen/. [Online; accessed 23-June-2025].

OpenSSL Foundation, Inc. (2020b). Hmac - openssl documentation. https://docs.openssl.org/1.1.1/man3/HMAC/. [Online; accessed 23-June-2025].

OpenSSL Foundation, Inc. (2020c). Rand_bytes. https://docs.openssl.org/1.1.1/man3/RAND_bytes/. [Online; accessed 18-June-2025].

OpenSSL Foundation, Inc. (2022a). Evp_pkey_derive - openssl documentation. https://docs.openssl.org/3.3/man3/EVP_PKEY_derive/. [Online; accessed 23-June-2025].

OpenSSL Foundation, Inc. (2022b). X509_pubkey_new - openssl documentation. https://docs.openssl.org/3.3/man3/X509_PUBKEY_new/. [Online; accessed 23-June-2025].

OpenSSL Foundation, Inc. (2024a). Evp_encryptinit - openssl documentation. https://docs.openssl.org/3.1/man3/EVP_EncryptInit/. [Online; accessed 23-June-2025].

OpenSSL Foundation, Inc. (2024b). Evp_kdf - openssl documentation. https://docs.openssl.org/3.3/man3/EVP_KDF/. [Online; accessed 23-June-2025].

OpenSSL Foundation, Inc. (2025). Welcome to openssl! https://www.openssl.org/. [Online; accessed 18-June-2025].

Randolph, K. & Hunt, M. (2021). Security incident report (version 1.2). Technical report, Verkada Inc., San Mateo, CA.

Raspberry Pi Ltd (2025a). Raspberry pi 3 model b+. https://datasheets.raspberrypi.com/rpi3/raspberry-pi-3-b-plus-product-brief.pdf. [Online; accessed 18-June-2025].

Raspberry Pi Ltd (2025b). Raspberry pi 4 model b. https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf. [Online; accessed 18-June-2025].

Springall, D., Durumeric, Z., & Halderman, J. A. (2016). Measuring the security harm of tls crypto shortcuts. In *Proceedings of the 2016 Internet Measurement Conference*, IMC '16, (pp. 33–47)., New York, NY, USA. Association for Computing Machinery.

Standards for Efficient Cryptography Group (2010). Sec 2: Recommended elliptic curve domain parameters. Technical report, Certicom Corporation. Version 2.0.

Temoshok, D. & Abruzzi, C. (2018). Developing trust frameworks to support identity federations. Technical Report NIST Internal Report NISTIR 8149, National Institute of Standards and Technology, Gaithersburg, MD.

The Tamarin Team (2024). *Tamarin-Prover Manual*.

tpm2-software (2024). tpm2-tss github repository. https://github.com/tpm2-software/tpm2-tss?tab=readme-ov-file. [Online; accessed 18-June-2025].

Trusted Computing Group (2008). Trusted platform module (tpm) summary. https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-Summary_04292008.pdf. [Online; accessed 18-June-2025].

Trusted Computing Group (2020). Tcg tss 2.0 enhanced system api (esapi) specification. Technical Report Version 1.00 Revision 08, Trusted Computing Group.

Trusted Computing Group (2025). Tpm 2.0 library. https://trustedcomputinggroup.org/resource/tpm-library-specification/. [Online; accessed 23-June-2025].

U.S. Food and Drug Administration (2017). Warning letter: Abbott (st. jude medical inc.) — MARCS–CMS 519686. Warning Letter 519686, Center for Devices and Radiological Health, U.S. Food and Drug Administration, Silver Spring, MD.

Zanella, A., Bui, N., Castellani, A., Vangelista, L., & Zorzi, M. (2012). Internet of things for smart cities. *Internet of Things Journal, IEEE, 1*.

APPENDIX

Full Tamarin Model

keyExchange.spthy

```
theory FinalKeyExchange
begin
  builtins: diffie-hellman, signing, hashing, symmetric-encryption

  functions: hkdf/2, hmac/2

  /* Client Rules */

  rule Client_Init:
    [ Fr(eskC), Fr(saltC), Fr(hmackey) ]
    --[ClientHello('C','S', 'g'^eskC, saltC, hmac(hmackey, <'g'^eskC, saltC>))]->
    [ Out(<'g'^eskC, saltC, hmac(hmackey, <'g'^eskC, saltC>)>), St_Client(eskC,
        saltC), !HmacKey(hmackey) ]

  rule Client_Finish:
    [St_Client(eskC,saltC),
    !RC('S', rc),
    In(<ltc, ltcsig>),
    In(<epkS, sigS, saltS, macS>)]
    --[Eq(verify(ltcsig, <'S', ltc>, rc), true),
    Eq(verify(sigS, epkS, ltc), true),
    Eq(macS, hmac(hkdf(epkS^eskC, <saltC, saltS>), <epkS, sigS, saltS>)),
    Commit_Client('C', 'S', epkS^eskC),
    Commit_ClientCK('C', 'S', hkdf(epkS^eskC, <saltC, saltS>)),
    Secret(epkS^eskC),
    Secret(hkdf(epkS^eskC, <saltC, saltS>))]->
    [St_Client_masterkey(epkS^eskC)]

  rule Client_Upload:
    [Fr(data), St_Client_masterkey(mkey), Fr(nonce)]
    --[Secret(data), Sent('C','S', data), UseNonce(mkey, nonce)]->
    [Out(<'UPLOAD', nonce,
         senc(data, hkdf(mkey, <nonce>)),
         hmac(hkdf(mkey, <nonce>), <nonce, senc(data, hkdf(mkey, <nonce>))>)>)]

  rule Client_Download:
    [ St_Client_masterkey(mkey),
        In( < 'DOWNLOAD', nS, ct, mac > ) ]
    --[ Eq( mac, hmac(hkdf(mkey, <nS>), <'DOWNLOAD', nS, ct>) ), /* binds the tag
         too */
        Eq( ct, senc(sdec(ct, hkdf(mkey, <nS>)), hkdf(mkey, <nS>)) ), /* shape
            check */
        Delivered('C','S', sdec(ct, hkdf(mkey, <nS>))) ]->
    [ ]

  /* Server Rules */
```

```

rule Register_root:
  [Fr(rk)]
  --[]->
  [!RK('S', rk), !RC('S', pk(rk))]

rule Register_ltk:
  [Fr(ltk), !RK('S', rk)]
  --[]->
  [!Ltk('S', ltk), Out(<pk(ltk), sign(<'S', pk(ltk)>, rk)>)]

rule Server_Resp:
  [!Ltk('S', ltk), !HmacKey(hmackey), In(<epkC, saltC, mac>), Fr(eskS), Fr(
      saltS)]
  --[Eq(mac, hmac(hmackey, <epkC, saltC>)),
  HelloVerified('S','C', epkC, saltC, mac),
  Witness_Server('S','C', epkC ^ eskS),
  Witness_ServerCK('S','C', hkdf(epkC^eskS, <saltC, saltS>))]->
  [Out(<'g'^eskS, sign('g'^eskS, ltk), saltS, hmac(hkdf(epkC^eskS, <saltC,
      saltS>), <'g'^eskS, sign('g'^eskS, ltk), saltS>)>),
  St_Server(epkC^eskS)]

rule Server_Receive:
  [ St_Server(mkey), In(<'UPLOAD', nonce, ct, mac>) ]
  --[ Eq(mac, hmac(hkdf(mkey, <nonce>), <nonce, ct>)),
      Eq(ct, senc(sdec(ct, hkdf(mkey, <nonce>)), hkdf(mkey, <nonce>))),
      Delivered('S','C', sdec(ct, hkdf(mkey, <nonce>))) ]->
  [ ]

rule Server_Download:
  [ St_Server(mkey), Fr(sdata), Fr(nS) ]
  --[ Secret(sdata), /* optional: lets you prove payload secrecy */
      Sent('S','C', sdata),
      UseNonceSC(mkey, nS) ]->
  [ Out( < 'DOWNLOAD', nS,
          senc(sdata, hkdf(mkey, <nS>)),
          hmac(hkdf(mkey, <nS>), <'DOWNLOAD', nS, senc(sdata, hkdf(mkey, <nS>))
              >) > ) ]

/* Compromise Rules */

rule Reveal_LTK:
  [!Ltk('S', ltk)]
  --[RevealLTK('S')]->
  [Out(ltk)]

/* Restrictions and Lemmas */

restriction Equality:
  "All x y #i. Eq(x, y) @i ==> x = y"

restriction NoncePerKeyUnique:
  "All k n #i #j. UseNonce(k,n) @ #i & UseNonce(k,n) @ #j ==> #i = #j"

restriction NoncePerKeyUnique_SC:
  "All k n #i #j. UseNonceSC(k,n) @ #i & UseNonceSC(k,n) @ #j ==> #i = #j"

lemma secrecy_keys:
    "All k #i. not (Ex #t. RevealLTK('S') @ #t) & Secret(k) @ #i ==> not (Ex #j. K(k) @ #j)"

  lemma agree_master:
    "All s c mk #i. not (Ex #t. RevealLTK('S') @ #t) & Commit_Client(c,s,mk) @ #i ==> (Ex #j. Witness_Server(s,c,mk) @ #j)"

  lemma agree_connkey:
    "All s c ck #i. not (Ex #t. RevealLTK('S') @ #t) & Commit_ClientCK(c,s,ck) @ #i ==> (Ex #j. Witness_ServerCK(s,c,ck) @ #j)"

  lemma agree_master_rev:
  "All s c mk epkC saltC mac #i. not (Ex #t. RevealLTK('S') @ #t) & Witness_Server(s,c,mk) @ #i & HelloVerified(s,c,epkC,saltC,mac) @ #i
   ==> (Ex #j. ClientHello(c,s,epkC,saltC, mac) @ #j)"

  lemma agree_connkey_rev:
  "All s c ck epkC saltC mac #i. not (Ex #t. RevealLTK('S') @ #t) & Witness_ServerCK(s,c,ck) @ #i & HelloVerified(s,c,epkC,saltC,mac) @ #i
   ==> (Ex #j. ClientHello(c,s,epkC,saltC, mac) @ #j)"

  lemma upload_conf:
    "All d #i. not (Ex #t. RevealLTK('S') @ #t) & Sent('C','S', d) @ #i ==> not (Ex #t. K(d) @ #t)"

  lemma upload_auth:
    "All d #i. not (Ex #t. RevealLTK('S') @ #t) & Delivered('S','C', d) @ #i ==> (Ex #j. Sent('C','S', d) @ #j)"

  lemma download_auth:
    "All d #i. not (Ex #t. RevealLTK('S') @ #t) & Delivered('C','S', d) @ #i ==> (Ex #j. Sent('S','C', d) @ #j)"

  lemma download_conf:
    "All d #i. not (Ex #t. RevealLTK('S') @ #t) & Sent('S','C', d) @ #i ==> not (Ex #t. K(d) @ #t)"

  lemma forward_secrecy:
    "All mk #i #j. Secret(mk) @ #i & RevealLTK('S') @ #j & #i < #j & not (Ex #k. #k < #i & RevealLTK('S') @ #k) ==> not (Ex #t. K(mk) @ #t)"
end
```