People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University M'Hamed Bougara of Boumerdes

Faculty of Technology

End-of-Study Project: PFE

Presented in partial fulfilment of the requirements for the degree of **Licence en Informatique**

Study and Implementation of a Digital Signature System Based on the RSA Algorithm

Presented by:

Azouaou Toufik
Rabehi Mohamed Sami
Slimani Abd El Djalil

Supervised by:

Mme. Aicha Benyoucef

Examined by:

Mme. Mehibel Nissa

Academic Year: 2025 / 2026

ACKNOWLEDGEMENTS

Above all, we would like to thank Almighty Allah for granting us the strength, patience and wisdom to complete this project.

We would like to express our sincere gratitude to all those who supported us throughout the realization of this project.

First and foremost, we extend our heartfelt thanks to Mme. Aicha Benyoucef, our supervisor, for her valuable guidance, constant encouragement, and insightful feedback. Her support was instrumental in shaping and completing our work.

We are also grateful to Mme. Mehibel Nissa, for agreeing to evaluate our work.

We would like to acknowledge our colleagues and friends for their collaboration, assistance, and motivation during this journey.

Finally, we are deeply grateful to our families for their unconditional support, patience, and encouragement throughout our academic path. This accomplishment would not have been possible without them.

ABSTRACT

Digital files carry no built-in proof of origin or integrity. Any node that intercepts network traffic can silently modify a document and forward it without leaving a trace. This project studies and implements an RSA digital signature system designed to close that gap.

The system is written in Python 3 using the `pycryptodome` library. It consists of three modules: a sender that generates a fresh 2048-bit RSA key pair per session, hashes the target file with SHA-256, and produces a PKCS#1 v1.5 padded signature; a receiver that independently rehashes the received file and verifies the signature against the stored public key; and a shared utility library that wraps key generation, hashing, signing, verification, and timing under one consistent interface. Both modules are operated through a command-line interface with no external configuration.

The theoretical part derives the RSA algorithm from modular arithmetic and the Euler–Fermat theorem, establishes why PKCS#1 v1.5 padding is necessary, and maps the four known cryptanalytic attack routes against RSA to the integer factorisation problem.

Four experiments were run on three Intel platforms: Ice Lake i5-1035G7, Tiger Lake i5-1145G7, and Raptor Lake i9-13900H. The hash benchmark eliminated MD5 and SHA-1 for demonstrated collision vulnerabilities and SHA-512 for unnecessary overhead; SHA-256 delivered the best throughput among the remaining secure candidates on all three machines. Signing latency at 2048 bits ranged from 2.79 ms to 4.30 ms, within the NIST-recommended 112-bit security level. The signature-size experiment confirmed a constant 256-byte overhead regardless of document length. Seven passive tamper scenarios and three active Man-in-the-Middle forgery attempts were all correctly rejected.

Keywords: Digital Signature, RSA, SHA-256, PKCS#1 v1.5, Hash Function, Cryptography, Python, pycryptodome, Information Security, Man-in-the-Middle.

RÉSUMÉ

Les fichiers numériques ne portent par eux-mêmes aucune preuve d'origine ni d'intégrité. Tout intermédiaire qui intercepte le trafic réseau peut modifier un document et le retransmettre sans laisser de trace. Ce projet étudie et met en œuvre un système de signature numérique basé sur RSA pour combler cette lacune.

Le système est développé en Python 3 avec la bibliothèque `pycryptodome`. Il se divise en trois modules : un émetteur qui génère une nouvelle paire de clés RSA de 2048 bits à chaque session, calcule l'empreinte du fichier avec SHA-256 et produit une signature avec le rembourrage PKCS#1 v1.5 ; un récepteur qui recalcule indépendamment l'empreinte du fichier reçu et vérifie la signature à l'aide de la clé publique ; et une bibliothèque utilitaire commune regroupant génération de clés, hachage, signature, vérification et mesure du temps sous une interface uniforme. Les deux modules fonctionnent via une interface en ligne de commande, sans configuration externe.

La partie théorique dérive l'algorithme RSA à partir de l'arithmétique modulaire et du théorème d'Euler–Fermat, justifie la nécessité du rembourrage PKCS#1 v1.5, et rattache les quatre vecteurs d'attaque cryptanalytique connus contre RSA au problème de la factorisation des entiers.

Quatre expériences ont été menées sur trois plateformes Intel (Ice Lake i5-1035G7, Tiger Lake i5-1145G7, Raptor Lake i9-13900H). Le test de hachage a éliminé MD5 et SHA-1 pour leurs vulnérabilités aux collisions démontrées, et SHA-512 pour sa surcharge inutile ; SHA-256 a offert le meilleur débit parmi les candidats sécurisés restants sur les trois machines. La latence de signature à 2048 bits varie entre 2,79 ms et 4,30 ms, dans le niveau de sécurité de 112 bits recommandé par le NIST. L'expérience sur la taille des signatures a confirmé une surcharge constante de 256 octets, indépendante de la taille du document. Sept scénarios de falsification passive et trois tentatives actives d'attaque de l'homme du milieu ont toutes été correctement rejetées.

Mots-clés : Signature Numérique, RSA, SHA-256, PKCS#1 v1.5, Fonction de Hachage, Cryptographie, Python, pycryptodome, Sécurité de l'Information, Attaque de l'Homme du Milieu.

ملخص

لا تحمل الملفات الرقمية بطبيعتها أي دليل على مصدرها أو سلامتها. أي عقدة تعترض حركة الشبكة تستطيع تعديل مستند وإعادة إرساله دون أن يُترك أثر. هذا المشروع يدرس نظام التوقيع الرقمي المبني على RSA وينفذه لسد هذه الفجوة.

النظام مكتوب بلغة Python 3 مع مكتبة `pycryptodome`. يتكون من ثلاثة مكونات: مرسِل يولّد زوج مفاتيح RSA بطول 2048 بت لكل جلسة ويحسب بصمة الملف بـSHA-256 وينتج توقيعاً بحشو PKCS#1 v1.5؛ ومستقبِل يعيد حساب البصمة باستقلالية ويتحقق من التوقيع بالمفتاح العام المخزّن؛ ومكتبة مشتركة تجمع توليد المفاتيح والتجزئة والتوقيع والتحقق وقياس الوقت في واجهة واحدة. كلا المكونين يعملان عبر سطر الأوامر دون إعداد خارجي.

يشتق الجزء النظري خوارزمية RSA من الحساب المعياري ونظرية أويلر-فيرما، ويبرر ضرورة حشو PKCS#1 v1.5، ويربط مسارات الهجوم التحليلي الأربعة المعروفة ضد RSA بمشكلة تحليل الأعداد إلى عوامل.

أُجريت أربع تجارب على ثلاث منصات: Intel Ice Lake i5-1035G7 وTiger Lake i5-1145G7 وRaptor Lake i9-13900H. أسقط اختبار التجزئة MD5 وSHA-1 لثغرات التصادم المُثبتة، وأسقط SHA-512 لتكلفته الحسابية غير المبررة؛ وجاء SHA-256 بأعلى معدل نقل بين الخوارزميات الآمنة المتبقية على الأجهزة الثلاثة. تراوح كمون التوقيع عند 2048 بت بين 2.79 ملي‌ثانية و4.30 ملي‌ثانية، ضمن مستوى الأمان 112 بتاً الذي توصي به NIST. أثبتت تجربة حجم التوقيع أن الزيادة ثابتة عند 256 بايتاً مهما كان طول المستند. رُفضت سبعة سيناريوهات تلاعب سلبي وثلاث محاولات تزوير نشطة بهجوم الرجل في المنتصف جميعها بنجاح.

CONTENTS

Acknowledgements **I**

Abstract **II**

Résumé **III**

Arabic Abstract **IV**

Acronyms **IX**

1 General Introduction **1**
1.1 General context: cryptography and asymmetric encryption 1
1.2 Digital signatures: concept and role 2
1.3 Real-world use of RSA digital signatures 3
1.4 The Man-in-the-Middle Attack 4
1.4.1 How RSA digital signatures stop MITM 4
1.5 Problem statement 5
1.6 Objectives 5

2 Theoretical Framework **6**
2.1 Introduction 6
2.2 Formalizing Digital Signatures 6
2.3 Public Key Cryptography Mechanics 6
2.3.1 The Asymmetric Model 7
2.4 The RSA Algorithm 7
2.4.1 Mathematical Prerequisites 7
2.4.2 RSA Key Generation 8
2.4.3 RSA Encryption and Decryption 9
2.5 RSA Encryption versus RSA Digital Signature 9
2.6 Hash Functions and Padding 10
2.6.1 Formal Hash Requirements 10
2.6.2 PKCS#1 v1.5 Padding 10
2.7 RSA Digital Signature Mechanism 11
2.7.1 Formal Signing and Verification: Process Flow 11
2.7.2 Formal Signing Operation 11

2.7.3 Formal Verification Operation 12
2.7.4 Correctness Proof 12
2.8 Mathematical Formulation 12
2.8.1 Complete Key-Generation Formulas 12
2.8.2 Encryption, Decryption, Signing, Verification 12
2.8.3 Extended Euclidean Algorithm 13
2.9 Security Properties 13
2.9.1 How RSA Mathematically Ensures Security 13
2.9.2 Resistance to Cryptanalytic Attack 13
2.10 Detailed Example 14
2.10.1 Key Generation ($p = 47$, $q = 59$) 14
2.10.2 Message Encoding 15
2.10.3 Signing ($d = 157$, $n = 2773$) 15
2.10.4 Verification ($e = 17$, $n = 2773$) 15
2.11 Limitations and Theoretical Aspects 16
2.12 Conclusion 16

3 Implementation and Results 17
3.1 Introduction 17
3.2 Implementation 17
3.2.1 System Overview 17
3.2.2 Project Structure 17
3.2.3 Core Library: `utils.py` 18
3.2.4 Sender and Receiver 19
3.3 Hash Algorithm Performance Benchmark 19
3.3.1 Experimental Setup 19
3.3.2 Results 20
3.3.3 Analysis 22
3.4 Key Size Impact on Performance 23
3.4.1 Experimental Setup 23
3.4.2 Results 23
3.4.3 Analysis 25
3.5 Signature Size Invariance 26
3.6 Tamper Detection Test 27
3.6.1 Experimental Setup 27
3.6.2 Results 27
3.6.3 Analysis 28
3.7 Conclusion 29

General Conclusion 30

LIST OF FIGURES

1.1 Symmetric vs. asymmetric encryption. *Source:* [1] 2
1.2 RSA digital signature process. *Source:* [2] 3
1.3 Man-in-the-Middle (MITM) attack. *Source:* [3] 4

2.1 RSA key generation procedure 8
2.2 Square-and-multiply algorithm for modular exponentiation 9
2.3 RSA signing and verification process flow 11
2.4 Cryptanalytic attack routes against RSA 14

3.1 Sender–receiver interaction through shared output files 19
3.2 Hash throughput on i5-1035G7 (log–log). SHA-256 is fastest among secure candidates. 21
3.3 Hash performance at 10 MB across all three CPUs (ms). 21
3.4 Signing latency vs. key size. Growth is super-linear above 2048 bits. 24
3.5 Key generation time vs. key size (log scale). All three platforms show super-linear growth. 25
3.6 SHA-256 bytes changed per test scenario (out of 32). All tampered variants trigger near-total diffusion. 28

LIST OF TABLES

2.1 Mathematical tools used in RSA 8
2.2 RSA encryption and decryption operations 9
2.3 RSA encryption versus RSA digital signature 10
2.4 Required properties of the cryptographic hash function 10
2.5 Core RSA operations in mathematical notation 12
2.6 RSA digital signature security guarantees mapped to mathematics 13
2.7 Signing results for `MATHEMATICS` 15
2.8 Verification results for `MATHEMATICS` 16
2.9 Known theoretical limitations of RSA 16

3.1 Source files and their roles 18
3.2 Hash benchmark: i5-1035G7 (ms) 20
3.3 Hash benchmark: i5-1145G7, affinity-pinned (ms) 20
3.4 Hash benchmark: i9-13900H (ms) 21
3.5 Hash performance at 10 MB across all three CPUs (ms) 21
3.6 Cryptographic security status of evaluated hash algorithms 22
3.7 RSA key size benchmark: i5-1035G7 (ms) 24
3.8 RSA key size benchmark: i5-1145G7 (ms) 24
3.9 RSA key size benchmark: i9-13900H (ms) 24
3.10 Cross-CPU RSA performance comparison at 2048-bit key size (ms) 24
3.11 RSA signature size as a function of input file size (2048-bit key) 26
3.12 Passive modification scenario results 27
3.13 Active MITM forgery attempt results 28

ACRONYMS

APT	Advanced Package Tool
ASN.1	Abstract Syntax Notation One
CLI	Command-Line Interface
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
DER	Distinguished Encoding Rules
EEA	Extended Euclidean Algorithm
eIDAS	electronic IDentification, Authentication and trust Services
GNFS	General Number Field Sieve
HTTPS	HyperText Transfer Protocol Secure
IPC	Instructions Per Clock
LSB	Least Significant Bit
MD5	Message Digest Algorithm 5
MITM	Man-in-the-Middle
NIST	National Institute of Standards and Technology
OID	Object Identifier
PDF	Portable Document Format
PEM	Privacy Enhanced Mail
PFE	Projet de Fin d'Études
PKCS	Public Key Cryptography Standards
RAM	Random-Access Memory
RSA	Rivest–Shamir–Adleman
SHA	Secure Hash Algorithm (SHA-1/256/512 are digest-size variants)
SHA-NI	SHA New Instructions (Intel hardware acceleration)

S/MIME	Secure/Multipurpose Internet Mail Extensions
SSH	Secure Shell
TLS	Transport Layer Security

CHAPTER 1

GENERAL INTRODUCTION

Every day, millions of people sign contracts, download software updates, submit official documents, and access government services online. Each of those actions rests on a question that rarely gets asked: how do we know the file we received is the real one?

In the physical world, a handwritten signature ties a person to a document. It is hard to forge, it stays on the paper, and it confirms the signer agreed to the content. That model breaks down with digital files. A Portable Document Format (PDF) can be copied perfectly. A scanned signature can be pasted anywhere. A contract can be silently modified after signing with no trace left behind.

The solution is not to replicate paper documents in digital form, but to apply mathematics directly. Asymmetric cryptography provides a mechanism to attach a value to a document that only the legitimate key holder can produce and that breaks the moment any byte changes. That value is a digital signature. The Rivest–Shamir–Adleman (RSA) algorithm is one of the principal algorithms used to produce it.

1.1 General context: cryptography and asymmetric encryption

Cryptography is the practice of securing information so that only intended parties can read or trust it. For most of its history, cryptography meant symmetric encryption: the sender and receiver share one secret key, and the same key both encrypts and decrypts the message. This works when both sides can agree on a key in advance. Over a network with thousands or millions of users who have never met, agreeing on a shared secret without an eavesdropper intercepting it is the problem itself.

Asymmetric cryptography, introduced by Diffie and Hellman in 1976 [1], solves this by splitting the key into two. One half is public, shared openly. The other half is private, kept by its owner alone. What one key does, only the other can undo. Two strangers can communicate securely without ever exchanging a secret.

Figure 1.1 shows the difference. In symmetric encryption, the same key locks and unlocks the message. If the key leaks, everything encrypted with it is exposed. In asymmetric encryption, the keys are different: the public key encrypts, and only the private key decrypts.

RSA, published in 1978 [2], was one of the first practical asymmetric algorithms. Its security rests on a simple fact: multiplying two large primes together is fast, but factoring

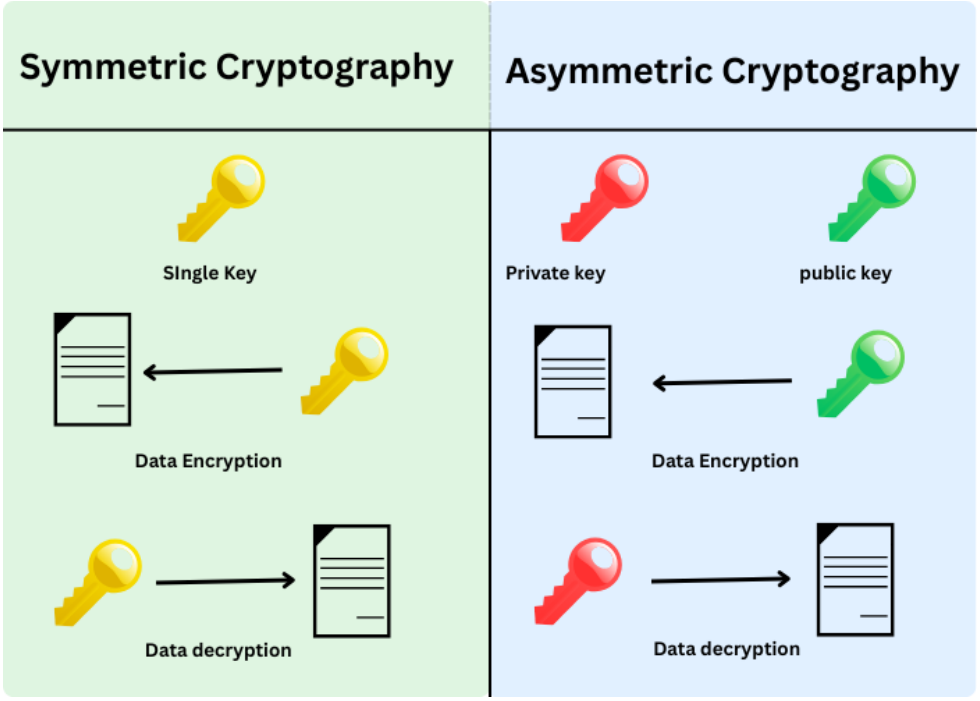


Figure 1.1: Symmetric vs. asymmetric encryption. *Source:* [1]

the result back into those primes is not. That gap produces two mathematically linked keys: one public, one private. RSA can be used for both encryption and digital signatures. This project uses it only for signatures.

1.2 Digital signatures: concept and role

A digital signature is a value attached to a document that ties it to a specific sender [2]. The sender uses their private key to produce the signature. Anyone with the sender's public key can verify it. If the document changes by even one byte after signing, verification fails.

This is different from encryption. Encryption hides the content of a message from everyone except the intended receiver. A digital signature does not hide anything; it proves two things: the document came from a specific sender, and nothing in it was altered [2].

Three properties follow from this [2]:

Authentication Only the person holding the private key can produce a valid signature. Verifying it with the matching public key confirms the sender's identity.

Integrity The signature is computed from the document's content. Any modification, even a single changed character, produces a different hash and the check fails.

Non-repudiation Because only the signer holds the private key, they cannot later claim someone else produced the signature.

Figure 1.2 shows how RSA is used to create and verify a digital signature. The sender hashes the document with the Secure Hash Algorithm 256-bit (SHA-256), then signs that hash with their private key. The receiver independently hashes the document they received and uses the sender's public key to recover the original hash from the signature. If the two hashes match, the document is authentic [2].

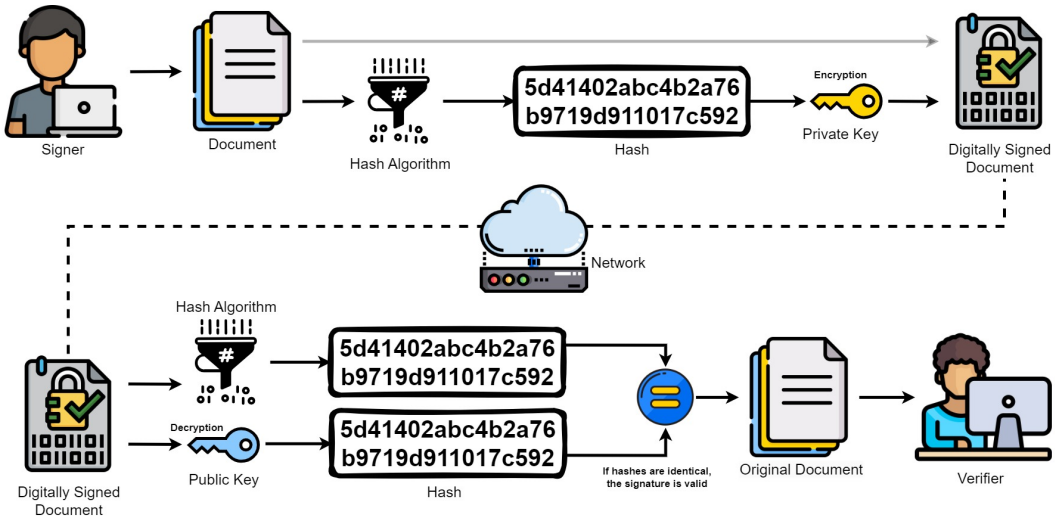


Figure 1.2: RSA digital signature process. *Source:* [2]

1.3 Real-world use of RSA digital signatures

RSA signatures are used in most systems where data crosses a network and the receiver needs to trust what arrived.

- Secure web connections via HyperText Transfer Protocol Secure (HTTPS): When a browser connects to a website over HTTPS, the server presents a certificate signed by a trusted Certificate Authority using RSA. The browser verifies this signature before exchanging any data. In Transport Layer Security (TLS) 1.3, RSA is no longer used for key exchange, but RSA signatures still authenticate server certificates [4].
- Software distribution: Operating systems and package managers verify an RSA signature on every update before installing. If the package was modified after the developer signed it, the check fails and installation stops. Debian's Advanced Package Tool (APT), Microsoft's Authenticode, and Apple's code signing all rely on this [5].
- Legal and official documents: Governments and courts in many countries accept digitally signed documents as legally binding. The EU's electronic IDentification, Authentication and trust Services (eIDAS) regulation gives qualified digital signatures the same legal weight as handwritten ones [6].
- Email authentication: Secure/Multipurpose Internet Mail Extensions (S/MIME) uses RSA signatures so recipients can verify a message came from the real sender and was not modified in transit. Without it, forging the "From" header in an email takes no special skill [7].
- Code signing: Developers sign compiled software so users and deployment systems can confirm the binary comes from the real author. An unsigned binary, or one whose signature fails, is blocked by the operating system before it runs [5].
- PDF and document signing: Applications such as Adobe Acrobat embed RSA signatures directly in PDF files. The signature is tied to the document's content: change a single word and it invalidates [8].

- Secure Shell (SSH) authentication: Servers use RSA key pairs to authenticate users without passwords. The user signs a challenge with their private key; the server verifies using the public key stored in the `authorized_keys` file [9].

In every case, the mechanism is the same. A private key signs. A public key verifies. The signature is tied to the exact bytes of the document, so it breaks the moment anything changes.

1.4 The Man-in-the-Middle Attack

A network is not a sealed pipe between two people. Traffic passes through routers, access points, and servers along the way. Any of those can be compromised. In many cases, no one needs to compromise them at all: a public Wi-Fi network is enough [3].

This attack, known as a Man-in-the-Middle (MITM) attack, puts an attacker between a sender and a receiver. The attacker intercepts every message, reads it, modifies it if they choose, and forwards it. Neither side knows. The receiver gets what looks like the sender's document, acts on it, and has no reason to suspect anything went wrong [3].

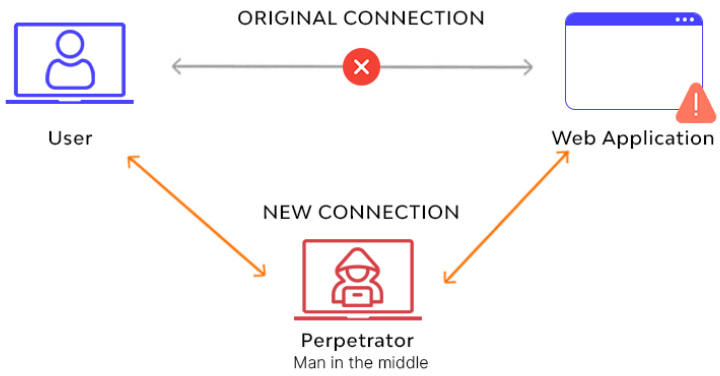


Figure 1.3: Man-in-the-Middle (MITM) attack. *Source:* [3]

A modified bank transfer, a tampered contract, a software package that now installs malware: the attack is the same in each case. The document looks real. If there is a signature, it is just an image pasted at the bottom. Nothing is verified.

1.4.1 How RSA digital signatures stop MITM

An RSA digital signature makes this attack detectable. Before sending, the sender computes a hash of the document and signs it with their **private key**. The signature travels with the document.

On the receiving end, the receiver recomputes the hash from the document they received, then uses the sender's **public key** to recover the original hash from the signature. If the two match, the document is exactly what the sender produced. If they do not, something changed in transit [3].

The attacker can still intercept. They can still modify the document. But the moment they do, the hashes diverge and the check fails. To forge a new valid signature, they would need the sender's private key, which they do not have. The attack fails.

1.5 Problem statement

In digital communication, three questions come up every time a file crosses a network:

- Does the document actually come from who it claims to come from?
- Has anyone changed its content between sending and receiving?
- Can the sender later deny having sent it?

These are the problems of **authentication**, **integrity**, and **non-repudiation**. Without a mechanism that addresses all three, any document received over a network is just bytes with no proof attached.

This project addresses a more specific question: how do we implement RSA digital signatures correctly, securely, and with measurable design choices?

Three sub-problems come with it.

The first is the **hash algorithm**. RSA operates on inputs smaller than the key modulus, which is 256 bytes for a 2048-bit key. Real documents are larger, so the signature is computed over a hash of the document rather than the document itself. The algorithm choice matters: Message Digest Algorithm 5 (MD5) and SHA-1 are both broken [10, 11], meaning two different documents can be engineered to produce the same hash. An attacker could craft a malicious document that shares a hash with a legitimately signed one, then reuse the valid signature. SHA-256 has no known collision attacks [12], but the speed difference between algorithms needs to be measured, not assumed.

The second is **padding**. Signing a raw hash with RSA is not secure. Without padding, certain mathematical properties of RSA allow signature forgery on specially crafted inputs. Public Key Cryptography Standards #1 (PKCS#1) v1.5 defines a padding scheme that closes these attack vectors [8]. Any correct implementation must include it.

The third is **key size**. A 1024-bit key no longer meets current security standards [5]. Larger keys are slower. The 2048-bit key is the current National Institute of Standards and Technology (NIST)-recommended minimum, but that choice should be confirmed with performance measurements rather than cited from convention.

1.6 Objectives

This project has four objectives:

1. Implement a Python application that generates RSA key pairs, signs arbitrary files, and verifies signatures, using SHA-256 hashing and PKCS#1 v1.5 padding, with a clean command-line interface.
2. Benchmark MD5, SHA-1, SHA-256, and SHA-512 across different file sizes and use the results to justify the algorithm choice.
3. Quantify how RSA key size affects signing and verification time, and use that data to justify the 2048-bit choice.
4. Run the system against multiple modification scenarios to confirm that any change to a signed document is correctly detected and rejected.

CHAPTER 2

THEORETICAL FRAMEWORK

2.1 Introduction

After establishing the motivation for digital signatures and introducing the key concepts in Chapter 1, this chapter works through the mathematical framework. It starts from the arithmetic prerequisites (modular arithmetic, Euler's theorem, the Extended Euclidean Algorithm) and builds up through key generation, the signing and verification operations, padding, and the known cryptanalytic attacks. The goal is to show exactly why each design choice holds, not just that it does.

2.2 Formalizing Digital Signatures

Any digital signature system must mathematically enforce two constraints to be cryptographically sound [2]:

- It must be **message-dependent**: the signature is inextricably tied to the exact bytes of the document. A one-byte change produces a different mathematical output and invalidates the signature.
- It must be **signer-dependent**: only the holder of the private key possesses the secret parameters required to produce a valid signature. Anyone can verify it using the public parameters, but no one can forge it.

2.3 Public Key Cryptography Mechanics

Notation used in this chapter. Before the formal development, the symbols used throughout this chapter are listed once here for easy reference:

- M: the document or message to be signed (treated as an integer).
- E, D: the public-key and private-key operations of a generic public-key scheme. In encryption, E (public) encrypts and D (private) decrypts; in a digital signature the roles invert, so D (private) signs and E (public) verifies.
- p, q: two large secret prime numbers chosen at random.

- $n = p\,q$: the RSA modulus; made public.
- $\varphi(n) = (p-1)(q-1)$: Euler's totient of n; kept secret (used only during key generation).
- e: the public exponent (commonly 65537).
- d: the private exponent; the modular inverse of e modulo $\varphi(n)$, i.e. $ed \equiv 1 \pmod{\varphi(n)}$.
- $(e,\, n)$: the *public key* (anyone can use it to verify).
- $(d,\, n)$: the *private key* (only the signer holds it).
- $H(\cdot)$: a cryptographic hash function (e.g. SHA-256).
- $h = H(M)$: the hash digest of the document.
- $\widetilde{h}$: the padded hash block produced by applying PKCS#1 v1.5 padding to h.
- S: the digital signature itself, computed as $S \equiv \widetilde{h}^{\,d} \pmod{n}$.
- h', h_r: on the verifier side, h' is the hash recomputed from the received document and h_r is the hash recovered from S using the public key; the signature is accepted only when $h' = h_r$.

2.3.1 The Asymmetric Model

A public-key cryptosystem uses a linked key pair. For it to work securely, it must satisfy four properties [2]:

(a) $D(E(M)) = M$ (decryption strictly inverts encryption)

(b) E and D are both computationally efficient

(c) Knowing E does not practically reveal D

(d) $E(D(M)) = M$ (required for signatures)

Property (d) is the cornerstone of digital signatures: applying the private key to a message produces a state that the public key can invert back to the original message. A function satisfying (a)–(c) is a *trap-door one-way function*; one that also satisfies (d) is a *trap-door one-way permutation* [2].

2.4 The RSA Algorithm

2.4.1 Mathematical Prerequisites

Table 2.1 lists the mathematical concepts that the key-generation and operation steps build on directly.

Table 2.1: Mathematical tools used in RSA

Concept	Definition and role in RSA
Modular arithmetic	$a \equiv b \pmod{n}$ iff $n \mid (a-b)$. All RSA operations are performed in this ring.
Coprimality	$\gcd(a,b) = 1$. Required for selecting e and computing d.
Euler's totient $\varphi(n)$	Counts integers in $\{1,\ldots,n-1\}$ coprime to n. For $n = pq$: $\varphi(n) = (p-1)(q-1)$ [2].
Euler–Fermat theorem	$M^{\varphi(n)} \equiv 1 \pmod{n}$ for $\gcd(M,n) = 1$. Guarantees correctness of RSA [2].
Extended Euclidean Alg.	Computes d such that $ed \equiv 1 \pmod{\varphi(n)}$ in $O(\log n)$ steps [2].

Key Identity

From the Euler–Fermat theorem, for any non-negative integer k:[1]

$$M^{k\varphi(n)+1} \equiv M \pmod{n}. \tag{2.1}$$

Since $ed \equiv 1 \pmod{\varphi(n)}$ means $ed = k\varphi(n) + 1$ for some integer k, we get:

$$M^{ed} \equiv M \pmod{n}. \tag{2.2}$$

This single identity proves both that decryption undoes encryption and that signature verification recovers the original message.

2.4.2 RSA Key Generation

Figure 2.1 traces the six steps that produce a linked public/private key pair from two large primes.

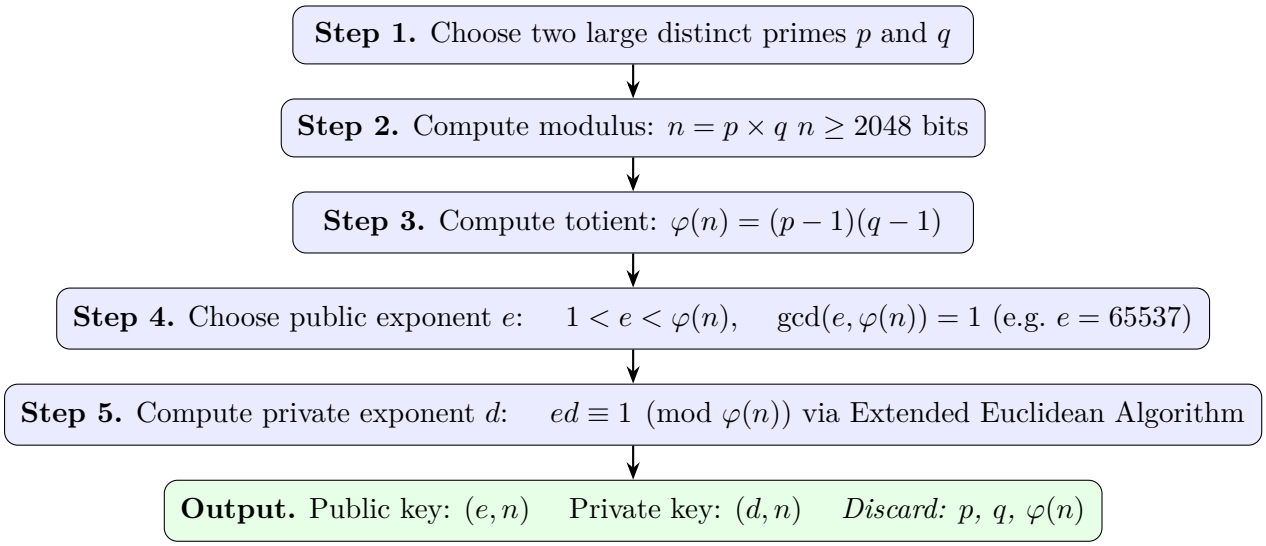


Figure 2.1: RSA key generation procedure

[1]The Euler–Fermat theorem requires $\gcd(M,n) = 1$. For the negligible case where M shares a factor with n, the identity holds by Fermat's Little Theorem applied mod p and mod q separately (since p and q are prime), then combined via the Chinese Remainder Theorem (CRT).

Note on exponent choice. The values 3, 17, and $65537 = 2^{16} + 1$ (Fermat primes F_0, F_2, F_4) are the most common choices for e. Their sparse binary representations reduce the cost of modular exponentiation [13].

2.4.3 RSA Encryption and Decryption

Table 2.2 sets out the two operations side by side, showing the key used and the modular exponentiation at each step.

Table 2.2: RSA encryption and decryption operations

Step	Encryption (Alice → Bob)	Decryption (Bob)
Key used	Bob's public key (e_B, n_B)	Bob's private key (d_B, n_B)
Operation	$C \equiv M^{e_B} \pmod{n_B}$	$M \equiv C^{d_B} \pmod{n_B}$
Correctness	$C^{d_B} = M^{e_B d_B} = M^{k\varphi(n)+1} \equiv M \pmod{n}$ (by eq. 2.2)	
Cost	$\leq 2\log_2 e$ mult. via square-and-multiply ($e = 65537$, fast) [2]	$\approx 2\log_2 d$ mult. via square-and-multiply ($d \approx n$, slow) [2]

Square-and-Multiply Algorithm

Figure 2.2 traces the algorithm's loop, showing the square-always, multiply-if-set structure that computes $M^e \bmod n$ in $O(\log e)$ multiplications.

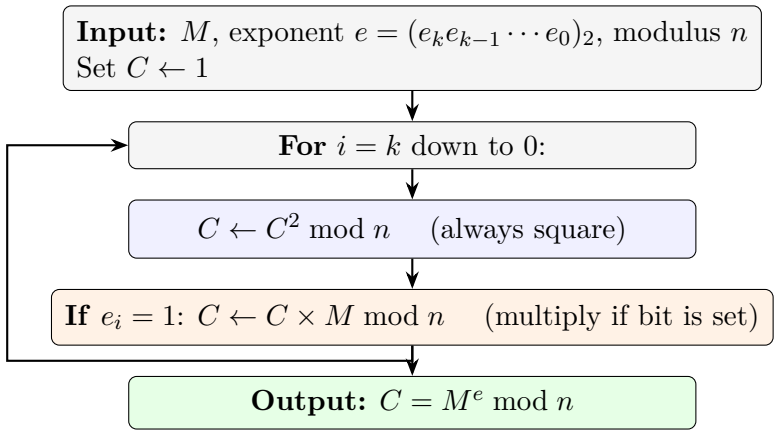


Figure 2.2: Square-and-multiply algorithm for modular exponentiation

2.5 RSA Encryption versus RSA Digital Signature

Both operations use modular exponentiation, but the keys are assigned in opposite roles. Table 2.3 shows the full comparison.

Table 2.3: RSA encryption versus RSA digital signature

Dimension	RSA Encryption	RSA Digital Signature
Goal	Hide message content	Prove origin, integrity, non-repudiation
Key at sender	Recipient's *public* key (e_B, n_B)	Sender's own *private* key (d_A, n_A)
Key at recipient	Recipient's *private* key (d_B, n_B)	Sender's *public* key (e_A, n_A)
Sender operation	$C \equiv M^{e_B} \pmod{n_B}$	$S \equiv H(M)^{d_A} \pmod{n_A}$
Recipient operation	$M \equiv C^{d_B} \pmod{n_B}$	Compute $H(M)$; check $S^{e_A} \equiv H(M) \pmod{n_A}$
Who can perform it	Anyone with the recipient's public key	Only the private-key holder can sign; anyone can verify
Forgeability	Secrecy protected by recipient's private key	Cannot be forged without the private key [2]
Non-repudiation	Not provided	Provided [2]
Mathematical duality	Decryption *mirrors* signing (both use the private key); encryption *mirrors* verification (both use the public key) [13]	

2.6 Hash Functions and Padding

2.6.1 Formal Hash Requirements

Hashing is what makes signing a large document possible: RSA operates on a fixed-size input, so the signature is computed over the hash, not the document itself. The hash algorithm must satisfy three properties for this to be secure. If any one fails, the signature scheme breaks:

Table 2.4: Required properties of the cryptographic hash function

Property	Requirement
Pre-image resistance	Given h, infeasible to find any M with $H(M) = h$
Second pre-image resistance	Given M, infeasible to find $M' \neq M$ with $H(M') = H(M)$
Collision resistance	Infeasible to find any pair (M, M'), $M \neq M'$, with $H(M) = H(M')$

2.6.2 PKCS#1 v1.5 Padding

Signing a bare hash is mathematically unsafe: algebraic properties of unpadded RSA allow signature forgery on crafted inputs. PKCS#1 v1.5 standardizes a padding scheme that wraps the hash before signing:

$$\mathrm{EM} = \underbrace{\texttt{0x00} \parallel \texttt{0x01}}_{\text{header}} \parallel \underbrace{\texttt{0xFF} \cdots \texttt{0xFF}}_{\text{padding to key width}} \parallel \underbrace{\texttt{0x00} \parallel T}_{\text{hash block}} \tag{2.3}$$

where T is the Distinguished Encoding Rules (DER) encoding of DigestInfo, an Abstract Syntax Notation One (ASN.1) structure that holds the hash algorithm Object Identifier (OID) together with the digest $H(M)$. The padded block $\widetilde{h}$ fills the full key width, which closes the algebraic attack vectors present in unpadded RSA. On the verification side, the recovered block is checked for exactly this structure before the hash is extracted; any deviation aborts verification before the digest comparison ever runs.

2.7 RSA Digital Signature Mechanism

2.7.1 Formal Signing and Verification: Process Flow

Figure 2.3 shows the complete RSA signing and verification process end to end.

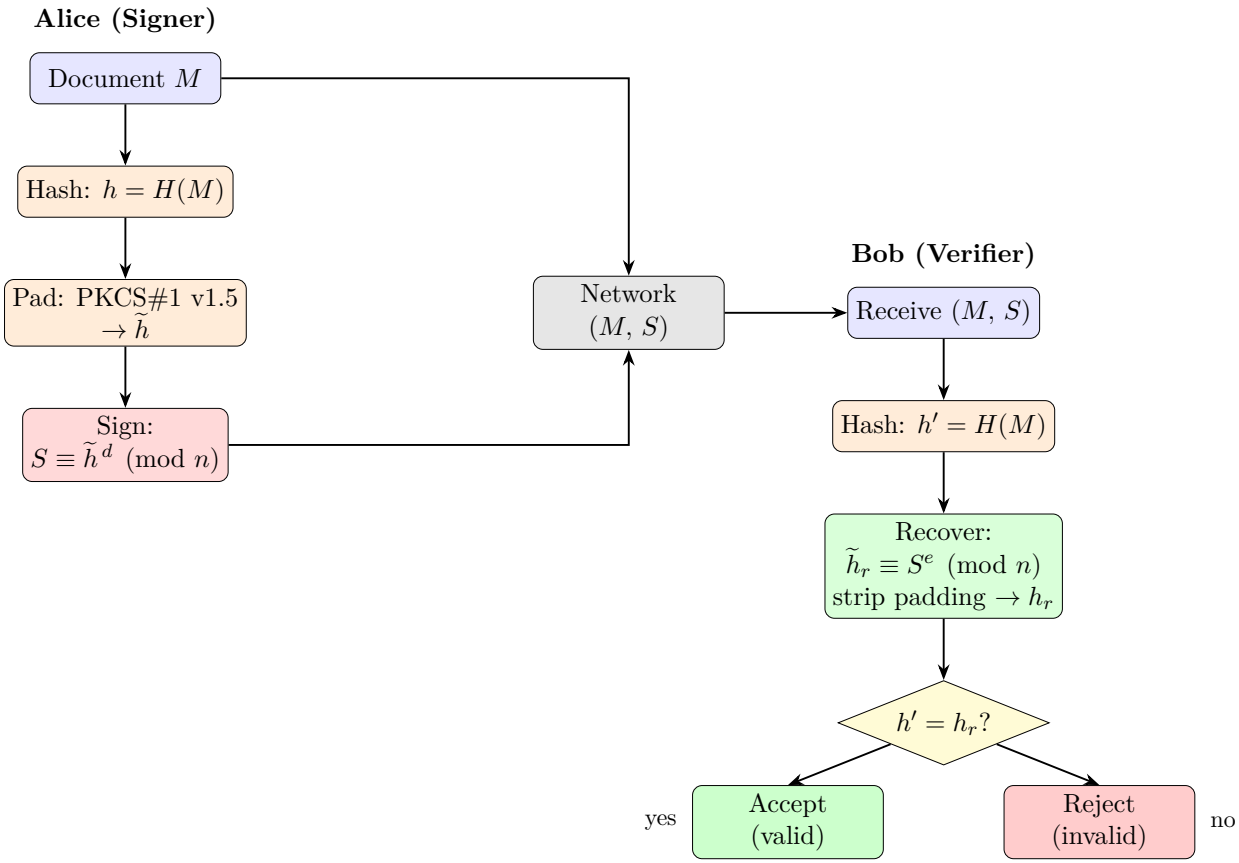


Figure 2.3: RSA signing and verification process flow

2.7.2 Formal Signing Operation

Alice holds private key (d, n) and signs document M. Let $\widetilde{h} = \text{PKCS1v15_pad}(H(M))$ denote the padded hash block defined in equation (2.3):

$$h = H(M) \tag{2.4}$$

$$S \equiv \widetilde{h}^{\,d} \pmod{n} \tag{2.5}$$

She transmits the pair (M, S) [13].

2.7.3 Formal Verification Operation

Bob holds Alice's public key (e, n) and receives (M, S):

$$h' = H(M) \tag{2.6}$$
$$\widetilde{h}_r \equiv S^e \pmod{n} \tag{2.7}$$
$$h_r = \text{strip_padding}(\widetilde{h}_r) \tag{2.8}$$

Accept if $h' = h_r$; **Reject** otherwise [13].

2.7.4 Correctness Proof

$$S^e = (\widetilde{h}^{\,d})^e = \widetilde{h}^{\,de} \equiv \widetilde{h}^{\,k\varphi(n)+1} \equiv \widetilde{h} \pmod{n} \tag{2.9}$$

The last step uses equation (2.1) (the same footnote applies: for the negligible case where $\widetilde{h}$ shares a factor with n, the step holds by Fermat's Little Theorem mod p and mod q and CRT). The verification therefore recovers the exact padded hash that was signed. If the document is unchanged, hashing it again produces the same h', and the mathematical comparison passes [2, 13].

2.8 Mathematical Formulation

2.8.1 Complete Key-Generation Formulas

The RSA key-generation and operation relationships are summarised in the following equations.

$$n = p \times q \tag{2.10}$$
$$\varphi(n) = (p-1)(q-1) \tag{2.11}$$
$$e: \quad 1 < e < \varphi(n), \quad \gcd(e, \varphi(n)) = 1 \tag{2.12}$$
$$d: \quad e \cdot d \equiv 1 \pmod{\varphi(n)} \tag{2.13}$$

2.8.2 Encryption, Decryption, Signing, Verification

Table 2.5 collects all four operations in a single compact notation.

Table 2.5: Core RSA operations in mathematical notation

Operation	**Input**	**Key**	**Output**
Encrypt	M	(e, n) public	$C \equiv M^e \pmod{n}$
Decrypt	C	(d, n) private	$M \equiv C^d \pmod{n}$
Sign	$\widetilde{h}$	(d, n) private	$S \equiv \widetilde{h}^d \pmod{n}$
Verify	S	(e, n) public	$\widetilde{h}_r \equiv S^e \pmod{n}$

2.8.3 Extended Euclidean Algorithm

To compute d from e and $\varphi(n)$, build the sequence $x_0 = \varphi(n)$, $x_1 = e$, $x_{i+1} = x_{i-1} \bmod x_i$, stopping when $x_k = 1$ [2]. Track coefficients b_i with $x_i = a_i x_0 + b_i x_1$. When $x_k = 1$, then $d \equiv b_k \pmod{\varphi(n)}$. The algorithm terminates in fewer than $2 \log_2 n$ steps.

2.9 Security Properties

2.9.1 How RSA Mathematically Ensures Security

Table 2.6 maps each of the three security properties to the mathematical mechanism that enforces it.

Table 2.6: RSA digital signature security guarantees mapped to mathematics

Property	Mathematical Mechanism	Why it holds
Authentication	Only d-holder can compute $S \equiv \widetilde{h}^d \pmod{n}$	Verified via public key (e, n); cryptographically tied to the generator [2]
Integrity	Signature algebraically covers $H(M)$	Any byte change alters $H(M)$; equation $h' = h_r$ fails [13]
Non-repudiation	Exclusive ownership of private parameter d	No other party holds d; forgery is conjectured to require factoring n (the equivalence is unproved) [2]

2.9.2 Resistance to Cryptanalytic Attack

Figure 2.4 maps each attack route to the computational reduction that makes it infeasible at 2048-bit keys.

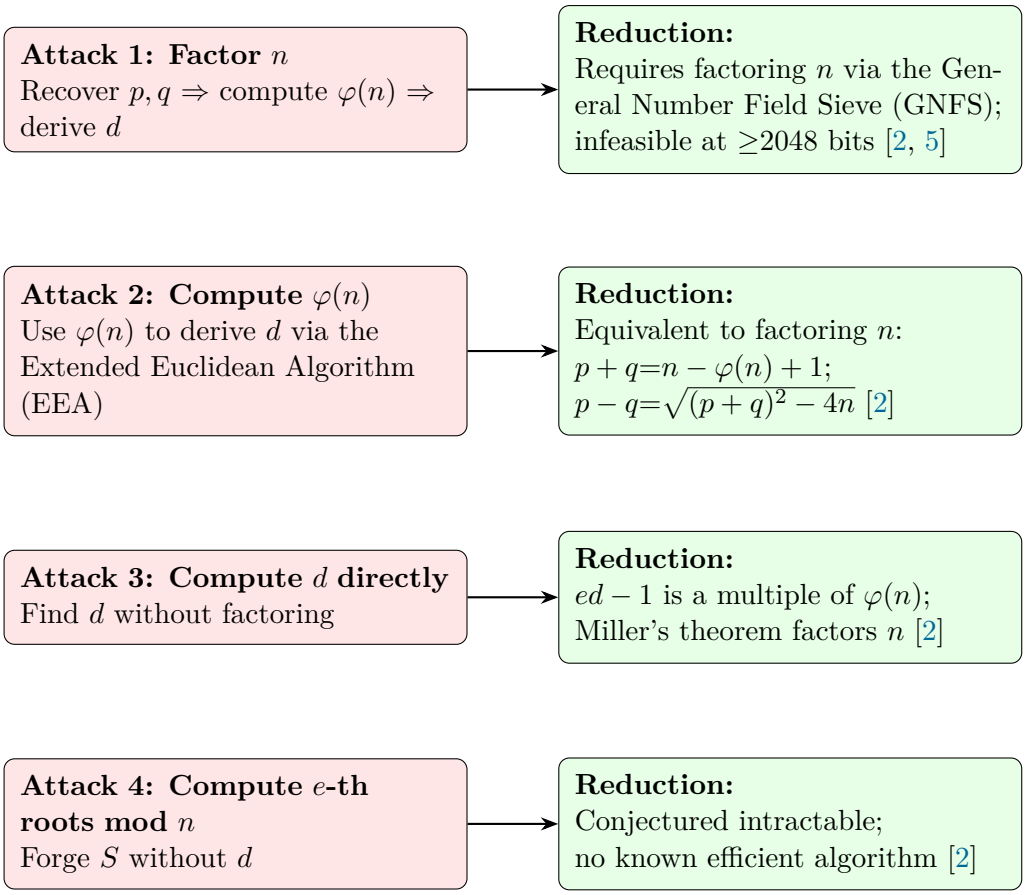


Figure 2.4: Cryptanalytic attack routes against RSA

2.10 Detailed Example

The following example uses parameters from Sharma and Badoga [13] and Rivest et al. [2].

2.10.1 Key Generation ($p = 47$, $q = 59$)

$$n = 47 \times 59 = 2773 \tag{2.14}$$
$$\varphi(n) = 46 \times 58 = 2668 \tag{2.15}$$
$$e = 17 \quad [\gcd(2668, 17) = 1] \tag{2.16}$$
$$d = 157 \quad [17 \times 157 \equiv 1 \pmod{2668}] \tag{2.17}$$

Extended Euclidean trace (solving $17d \equiv 1 \pmod{2668}$):

i	x_i	a_i	b_i
0	2668	1	0
1	17	0	1
2	16	1	−156
3	1	−1	157

Since $x_3 = 1$ and $b_3 = 157$, the computation confirms $d = 157$ [2]. Key pair: **Public:** (17, 2773) **Private:** (157, 2773).

2.10.2 Message Encoding

Message: `MATHEMATICS` (A=00, B=01, ..., Z=25). A trailing 'A' completes the last block [13]:

$$m_1 = 1200, \quad m_2 = 1907, \quad m_3 = 0412, \quad m_4 = 0019, \quad m_5 = 0802, \quad m_6 = 1800$$

2.10.3 Signing ($d = 157$, $n = 2773$)

Detailed computation for $m_1 = 1200$ via repeated squaring:

$$\begin{aligned} 1200^2 &\equiv 813 \pmod{2773} \\ 1200^4 &\equiv 995 \pmod{2773} \\ 1200^8 &\equiv 64 \pmod{2773} \\ 1200^{16} &\equiv 1323 \pmod{2773} \\ 1200^{32} &\equiv 566 \pmod{2773} \\ 1200^{64} &\equiv 1461 \pmod{2773} \\ 1200^{128} &\equiv 2084 \pmod{2773} \end{aligned}$$

Since $157 = 128 + 16 + 8 + 4 + 1$:

$$1200^{157} \equiv 2084 \times 1323 \times 64 \times 995 \times 1200 \equiv 2497 \pmod{2773}$$

Table 2.7: Signing results for `MATHEMATICS`

Block m_i	**Operation**	**Signature** S_i
1200	$1200^{157} \bmod 2773$	2497
1907	$1907^{157} \bmod 2773$	0239
0412	$0412^{157} \bmod 2773$	0707
0019	$0019^{157} \bmod 2773$	1537
0802	$0802^{157} \bmod 2773$	0723
1800	$1800^{157} \bmod 2773$	0741

2.10.4 Verification ($e = 17$, $n = 2773$)

For $S_1 = 2497$: since $17 = 2^4 + 1$, $2497^{16} \equiv 1563 \pmod{2773}$ by the square-and-multiply steps, so $2497^{17} = 2497^{16} \times 2497 \equiv 1563 \times 2497 \equiv 1200 \pmod{2773}$

Table 2.8: Verification results for `MATHEMATICS`

Signature S_i	**Operation**	**Recovered m'_i**	**Match?**
2497	$2497^{17} \bmod 2773$	1200	✓
0239	$0239^{17} \bmod 2773$	1907	✓
0707	$0707^{17} \bmod 2773$	0412	✓
1537	$1537^{17} \bmod 2773$	0019	✓
0723	$0723^{17} \bmod 2773$	0802	✓
0741	$0741^{17} \bmod 2773$	1800	✓

All six blocks recover correctly. The decoded sequence 1200 1907 0412 0019 0802 1800 maps back to `MATHEMATICS`, confirming the signature [13].

2.11 Limitations and Theoretical Aspects

Table 2.9 summarises the main theoretical limitations of RSA that informed the parameter choices in the implementation.

Table 2.9: Known theoretical limitations of RSA

Limitation	Explanation
No formal security proof	No technique exists to prove any encryption scheme secure in an absolute sense. Confidence in RSA comes from failed attacks over decades [2].
Factoring not provably hard	RSA assumes integer factorisation is infeasible, but this has never been proved. A future algorithm could break all existing RSA keys [2].
RSA $\not\equiv$ factoring (unproved)	The obvious attacks reduce to factoring, but Rivest et al. could not prove the converse: that *every* attack on RSA implies factoring. The equivalence is conjectured [2].
Reblocking problem	When a message is both signed (n_A) and encrypted (n_B), block sizes may not align if $n_A > n_B$. Two solutions: (1) threshold h keeping all signature $n < h$ and all encryption $n > h$; (2) re-encipher ciphertext above h until it falls within range [2].
Key size vs. speed	The private-key operation (signing) grows as $O((\log n)^3)$; public-key operations (verification) with a small fixed e grow as $O((\log n)^2)$. Security requires ≥ 2048 bits today; 4096 bits adds a security margin at measurable cost [2].
Prime selection constraints	p and q must differ in length, both $p-1$ and $q-1$ must have large prime factors, and $\gcd(p-1, q-1)$ must be small, to resist specialised factoring algorithms [2].

2.12 Conclusion

This chapter has covered the mathematical foundations of the implemented system: RSA key generation, the signing and verification operations, PKCS#1 v1.5 padding, and the cryptanalytic attacks that motivate the parameter choices made in the implementation. Chapter 3 applies these foundations directly, evaluating the system through four experiments and cross-CPU benchmarks.

CHAPTER 3

IMPLEMENTATION AND RESULTS

3.1 Introduction

Chapters 1 and 2 established the motivation and the mathematical foundations. This chapter describes the Python implementation and then presents the four experiments that justify each design choice from the problem statement: the hash algorithm, the key size, the constant signature length, and the tamper-detection behaviour.

3.2 Implementation

3.2.1 System Overview

The digital signature system is implemented in Python 3 using the `pycryptodome` library for all cryptographic operations (the standalone hash benchmark uses Python's built-in `hashlib`; see Section 3.3.1). It consists of three core modules and four standalone experiment scripts, all operated through a command-line interface with no external configuration required. The system generates RSA key pairs on demand, signs arbitrary files using a user-selected hash algorithm and PKCS#1 v1.5 padding, and verifies signatures against their public keys. The private key is not persisted between sessions; a fresh pair is generated for each signing operation. The public key is written to `public_key.pem` on disk for the receiver.

3.2.2 Project Structure

Table 3.1 lists each source file and its role in the system.

Table 3.1: Source files and their roles

Module	Role
`utils.py`	Shared library: key generation, hashing, signing, verification, and timing
`sender.py`	Interactive Command-Line Interface (CLI) sender: prompts for a file, hash algorithm, and key size; produces the signature and public key
`receiver.py`	Verifier: reads sender output, rehashes the received document, and reports the result
`exp_hash_speed.py`	Benchmark: hash algorithm throughput across five input sizes
`exp_key_size.py`	Benchmark: key generation, signing, and verification time per key size
`exp_sig_size.py`	Invariance test: signature byte length as a function of document size
`exp_tamper.py`	Tamper detection: passive modification and active MITM forgery scenarios
`run.bat`	Launcher: opens sender.py and receiver.py side-by-side in Windows Terminal
`experiments.bat`	Runner: executes all four experiment scripts in sequence

3.2.3 Core Library: `utils.py`

The shared library exposes four functions used by both the sender and the receiver. `generate_keys(key_size)` calls `RSA.generate()` from `pycryptodome` to produce a linked key pair, returning both the private key object and the derived public key. `hash_data(data, algo)` instantiates the appropriate hash object without finalising it, since the `pkcs1_15` signer requires the live object rather than a digest string. `sign(h, priv_key)` builds the PKCS#1 v1.5 padded block internally and performs the private exponentiation $S \equiv \widetilde{h}^{\,d} \pmod{n}$, returning the raw 256-byte signature. `verify(h, sig, pub_key)` applies the public exponentiation, strips the padding, compares the recovered hash with the freshly computed one, and returns a Boolean result, catching any exception so the caller never receives an unhandled error.

A fifth utility, `measure(func, *args)`, wraps any callable with `time.perf_counter()` timing and returns both the result and the elapsed time in milliseconds. It is used consistently across all modules and experiment scripts to ensure a uniform timing methodology.

Listing 3.1: Core cryptographic functions in `utils.py`

```
def generate_keys(key_size=2048):
    key = RSA.generate(key_size)
    return key, key.publickey()

def sign(h, priv_key):
    # PKCS#1 v1.5 padding is applied internally before signing
    return pkcs1_15.new(priv_key).sign(h)

def verify(h, sig, pub_key):
    # decrypts signature, strips padding, compares hash
```

```
    try:
        pkcs1_15.new(pub_key).verify(h, sig)
        return True
    except Exception:
        return False

def measure(func, *args):
    start = time.perf_counter()
    result = func(*args)
    elapsed = (time.perf_counter() - start) * 1000
    return result, elapsed
```

3.2.4 Sender and Receiver

The sender removes stale output files on startup so the receiver cannot read results from a previous session. It then prompts the user interactively for the file path, hash algorithm, and RSA key size. Once the inputs are collected, the signing pipeline runs in three steps: key generation, hashing, and signing. The prime factors p and q are printed in hexadecimal for inspection. Output is written in a specific order: the public key first, then the signature, and finally the metadata file `sender_meta.txt` which records the document path and hash algorithm. Writing the metadata file last acts as a completion signal: the receiver polls for it at 300 ms intervals, and its presence guarantees that the public key and signature are fully on disk before verification starts.

The receiver reads the document path and hash algorithm from the metadata, loads the document and signature from disk, rehashes the document with the same algorithm, and calls `verify()`. The result is printed as either **VALID** or **INVALID**, together with a timing breakdown covering hashing and verification. Figure 3.1 shows the complete interaction.

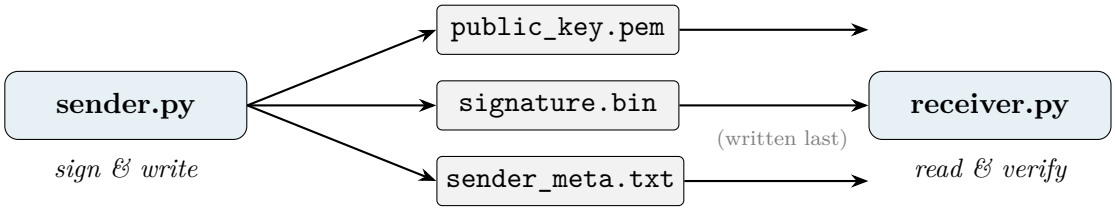


Figure 3.1: Sender–receiver interaction through shared output files

3.3 Hash Algorithm Performance Benchmark

3.3.1 Experimental Setup

We benchmarked four hash algorithms, namely MD5, SHA-1, SHA-256, and Secure Hash Algorithm 512-bit (SHA-512), to determine the best fit for the signing pipeline. The evaluation covers both computational speed and cryptographic security. To assess how microarchitecture affects throughput, the benchmark ran on three Intel Core processors: Ice Lake (i5-1035G7), Tiger Lake (i5-1145G7), and Raptor Lake (i9-13900H). The benchmark is implemented in `experiments/exp_hash_speed.py` and executed under the following environments:

- Intel Core i5-1035G7 (Ice Lake, 4 cores, base 1.20 GHz, boost 3.70 GHz)
- Intel Core i5-1145G7 (Tiger Lake, 4 cores, base 1.10 GHz, boost 4.40 GHz)
- Intel Core i9-13900H (Raptor Lake, 14 cores, base 2.60 GHz, boost 5.40 GHz)
- Random-Access Memory (RAM): 8 GB DDR4
- Operating System: Windows 11 (64-bit)
- Python: 3.10.12
- Library: `hashlib` (Python Standard Library)

Input data consisting of pseudo-random byte sequences was generated using `os.urandom` at five sizes: 1 KB, 10 KB, 100 KB, 1 MB, and 10 MB. Each algorithm processed every input once per run, and timing was captured with `time.perf_counter()`, which provides sub-millisecond resolution. The digest was produced via `hashlib.new(algo, data).hexdigest()` through the unified `hashlib` interface, so all four algorithms run under identical conditions on the same underlying C implementations.

All timings are single-run measurements; results indicate relative performance rather than statistically precise values.

3.3.2 Results

Tables 3.2–3.4 present the hashing time in milliseconds recorded for each algorithm across all input sizes, per processor. Table 3.5 consolidates the 10 MB results across all three Central Processing Units (CPUs) for direct architectural comparison at peak throughput.

Table 3.2: Hash benchmark: i5-1035G7 (ms)

Input Size	MD5 (ms)	SHA-1 (ms)	SHA-256 (ms)	SHA-512 (ms)
1 KB	0.039	0.034	0.015	0.019
10 KB	0.026	0.009	0.012	0.022
100 KB	0.158	0.069	0.086	0.177
1 MB	1.668	0.681	0.893	1.848
10 MB	15.835	6.764	9.012	18.360

Table 3.3: Hash benchmark: i5-1145G7, affinity-pinned (ms)

Input Size	MD5 (ms)	SHA-1 (ms)	SHA-256 (ms)	SHA-512 (ms)
1 KB	0.051	0.017	0.013	0.020
10 KB	0.026	0.009	0.011	0.020
100 KB	0.131	0.064	0.091	0.234
1 MB	1.231	0.610	0.836	1.912
10 MB	13.023	7.227	6.804	14.461

Table 3.4: Hash benchmark: i9-13900H (ms)

Input Size	MD5 (ms)	SHA-1 (ms)	SHA-256 (ms)	SHA-512 (ms)
1 KB	0.020	0.172	0.253	0.230
10 KB	0.017	0.006	0.006	0.012
100 KB	0.109	0.043	0.048	0.106
1 MB	0.937	0.355	0.409	0.960
10 MB	9.601	3.666	4.351	9.342

Table 3.5: Hash performance at 10 MB across all three CPUs (ms)

Processor	Generation	MD5	SHA-1	SHA-256	SHA-512
i5-1035G7	Ice Lake	15.835	6.764	9.012	18.360
i5-1145G7	Tiger Lake	13.023	7.227	6.804	14.461
i9-13900H	Raptor Lake	9.601	3.666	4.351	9.342

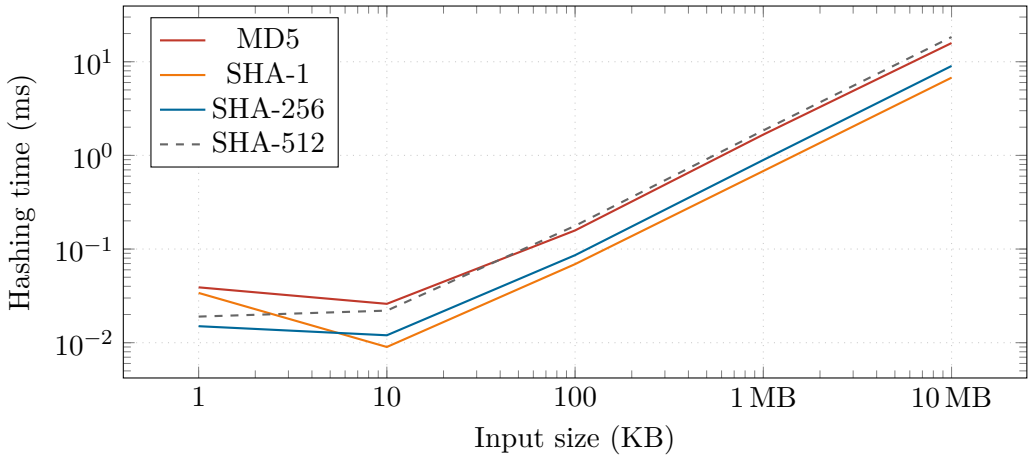


Figure 3.2: Hash throughput on i5-1035G7 (log–log). SHA-256 is fastest among secure candidates.

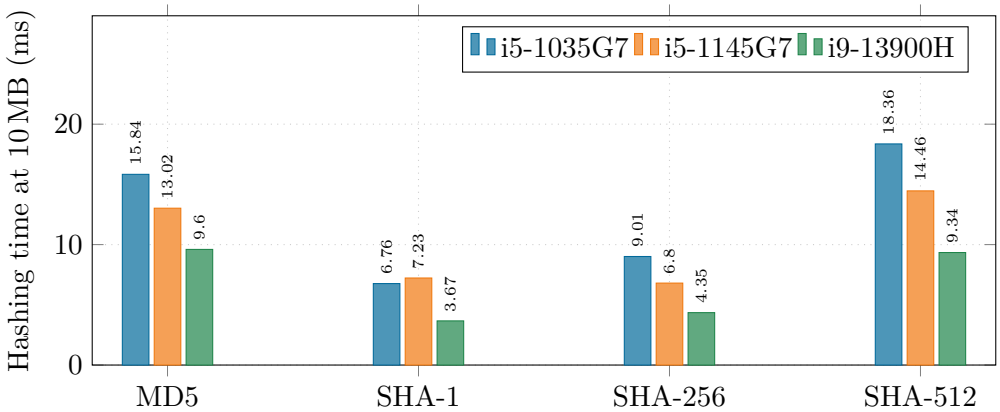


Figure 3.3: Hash performance at 10 MB across all three CPUs (ms).

Table 3.6: Cryptographic security status of evaluated hash algorithms

Algorithm	Digest Size (bits)	Security Status
MD5	128	**Broken**, practical collision attacks demonstrated
SHA-1	160	**Deprecated**, collision attack demonstrated (2017)
SHA-256	256	**Secure**, no known practical attacks
SHA-512	512	**Secure**, no known practical attacks

3.3.3 Analysis

Scaling Behaviour

Hashing time scales linearly with input size across all four algorithms, consistent with their block-processing construction: Merkle–Damgård for MD5, SHA-1, and SHA-256, and the wide-pipe variant for SHA-512.

Per-Algorithm Observations

At the 10 MB level on the i5-1035G7, SHA-1 achieves the lowest latency at 6.764 ms, followed by SHA-256 at 9.012 ms. MD5 records 15.835 ms, performing substantially worse than both SHA-1 and SHA-256, contrary to its common reputation for speed. Intel SHA Extensions (SHA-NI) are present on all three tested processors and provide dedicated hardware circuits for SHA-256 and SHA-1; MD5 and SHA-512 receive no such acceleration and execute entirely in software, which accounts for the reversal. SHA-512 is the slowest at 18.360 ms, attributable to its 80-round compression function operating on a 512-bit internal state.

On the i5-1145G7 (Tiger Lake), the benchmark was re-run with the process pinned to a single logical core using the Windows `start /affinity 1` command to eliminate OS thread-scheduling variance. Under affinity pinning, SHA-256 records 6.804 ms, SHA-1 records 7.227 ms, MD5 records 13.023 ms, and SHA-512 records 14.461 ms. SHA-256 is the fastest secure algorithm on this platform, consistent with the i5-1035G7 result. The 6.804 ms figure is lower than the i5-1035G7's 9.012 ms, indicating that the free-scheduling run was not significantly inflated by thread migration.

The i9-13900H (Raptor Lake) achieves the lowest absolute latencies across all algorithms: SHA-1 at 3.666 ms, SHA-256 at 4.351 ms, MD5 at 9.601 ms, and SHA-512 at 9.342 ms. The near-parity between SHA-512 and MD5 on this platform is notable and is explained by Intel SHA Extensions (SHA-NI): SHA-NI accelerates SHA-256 and SHA-1 in hardware but provides no acceleration for SHA-512 or MD5. Both algorithms execute entirely in software on all three platforms; the convergence on the i9-13900H reflects proportional gains in general-purpose integer throughput rather than any specific architectural benefit for SHA-512. The anomalously high values at 1 KB for SHA-1 (0.172 ms), SHA-256 (0.253 ms), and SHA-512 (0.230 ms) on this platform are a cold-start effect: the CPU instruction cache and branch predictor are unprimed on the first timing call, producing a one-time latency spike that does not recur at larger input sizes.

Architectural Impact

The i9-13900H cuts SHA-256 latency by 52% over the i5-1035G7 (4.351 ms vs. 9.012 ms) and 36% over the affinity-pinned i5-1145G7 (4.351 ms vs. 6.804 ms). For SHA-1, the improvement

over the i5-1035G7 is 46% (3.666 ms vs. 6.764 ms). These gains reflect Raptor Lake's higher clock speed (5.40 GHz), wider integer pipeline, and improved SHA-NI throughput.

Across all three platforms, hashing takes a minor fraction of total signing latency relative to modular exponentiation.

Algorithm Selection

From a security standpoint, both MD5 and SHA-1 are disqualified. MD5 is vulnerable to practical collision attacks, and SHA-1 was conclusively broken by Stevens et al. in 2017. A collision attack allows an adversary to construct two distinct documents M and M' such that $H(M) = H(M')$, which means a legitimately signed document could be paired with a malicious one carrying the same valid signature, completely defeating the integrity guarantee of the system.

SHA-512 is cryptographically sound but imposes measurable overhead relative to SHA-256, with no practical security benefit for this application: both provide a collision resistance level that exceeds the 112-bit security margin of 2048-bit RSA.

SHA-256 is therefore the best choice among the evaluated algorithms. It is immune to all known practical attacks, its 256-bit output fits into 2048-bit RSA signatures under PKCS#1 v1.5 padding, and it is the fastest secure candidate on every tested processor.

3.4 Key Size Impact on Performance

3.4.1 Experimental Setup

Key size directly governs both the security level and the cost of key generation, signing, and verification. Four standard sizes were benchmarked to confirm the 2048-bit choice and to show how the trade-off shifts across processor generations. The benchmark is implemented in `experiments/exp_key_size.py` using the `pycryptodome` library. RSA key pairs are generated via `RSA.generate(size)`, signing is performed with `pkcs1_15.new(key).sign()`, and verification with `pkcs1_15.new(key.publickey()).verify()`. In all cases the document is hashed with SHA-256 prior to signing. A fixed 1 KB pseudo-random payload generated by `os.urandom` is used across all key sizes. Each operation is timed individually with `time.perf_counter()`. The benchmark was executed on the same three processor configurations described in Section 3.3.1 under identical software conditions.

All timings are single-run measurements. Signing and verification are deterministic; key generation times vary between runs due to probabilistic prime search.

3.4.2 Results

Tables 3.7–3.9 present the key generation, signing, and verification times in milliseconds for each processor. Table 3.10 consolidates the signing and key generation times across all three CPUs at the standard 2048-bit key size.

Table 3.7: RSA key size benchmark: i5-1035G7 (ms)

Key Size (bits)	Key Generation (ms)	Sign (ms)	Verify (ms)
1024	115.32	1.34	0.30
2048	947.69	4.30	0.56
3072	5883.09	10.34	1.46
4096	8284.63	24.54	1.81

Table 3.8: RSA key size benchmark: i5-1145G7 (ms)

Key Size (bits)	Key Generation (ms)	Sign (ms)	Verify (ms)
1024	34.86	0.98	0.19
2048	774.52	3.92	0.51
3072	1079.74	8.45	0.94
4096	5793.48	18.42	1.56

Table 3.9: RSA key size benchmark: i9-13900H (ms)

Key Size (bits)	Key Generation (ms)	Sign (ms)	Verify (ms)
1024	61.28	0.71	0.15
2048	425.86	2.79	0.37
3072	6076.96	7.45	0.76
4096	2281.81	13.40	1.24

Table 3.10: Cross-CPU RSA performance comparison at 2048-bit key size (ms)

Processor	Generation	Key Gen (ms)	Sign (ms)	Verify (ms)
i5-1035G7	Ice Lake	947.69	4.30	0.56
i5-1145G7	Tiger Lake	774.52	3.92	0.51
i9-13900H	Raptor Lake	425.86	2.79	0.37

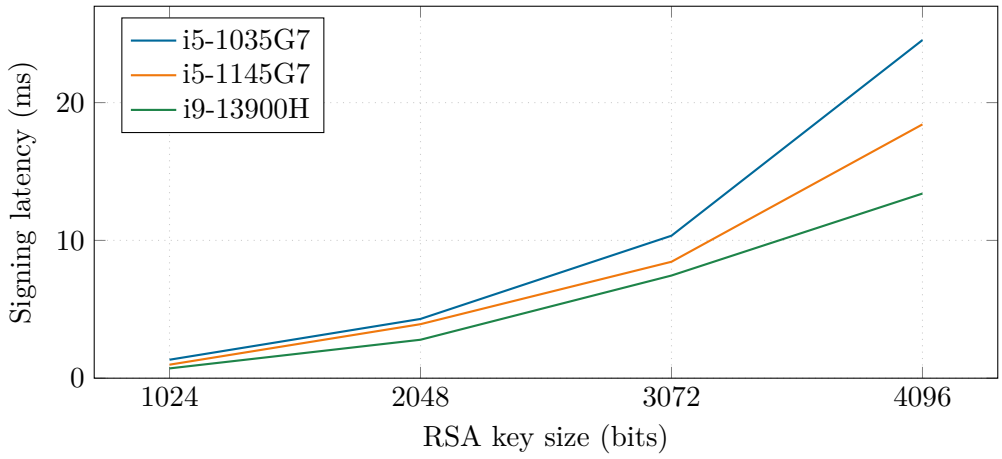


Figure 3.4: Signing latency vs. key size. Growth is super-linear above 2048 bits.

3.4.3 Analysis

Key Generation Scalability

Key generation grows super-linearly with key size on all three processors, because the cost is dominated by random prime search and the Extended Euclidean Algorithm. Since keys are generated once and saved to disk, even the longest observed run is acceptable for an interactive tool. Two points deviate from this trend: the i5-1145G7 at 3072 bits runs in only 1079.74 ms (compare 774.52 ms at 2048 bits and 5793.48 ms at 4096 bits), and the i9-13900H at 4096 bits (2281.81 ms) is faster than its own 3072-bit run (6076.96 ms). Both numbers come from a single execution, and prime search is probabilistic: its runtime depends on how many candidates Miller–Rabin rejects before one passes, so a lucky draw can shorten a run several times over. The two dips are therefore variance of the prime search and not an architectural property of the CPU. Figure 3.5 shows the growth curve across all three platforms.

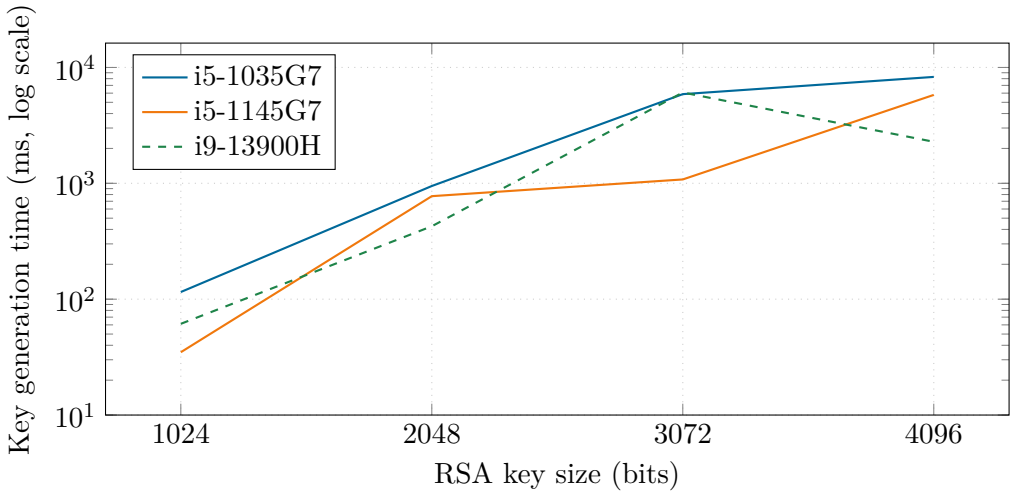


Figure 3.5: Key generation time vs. key size (log scale). All three platforms show super-linear growth.

Signing and Verification Latency

Signing time follows a super-linear growth curve with key size across all platforms, since the operation involves modular exponentiation $m^d \bmod n$ with the private exponent d, whose bit length is proportional to the key size. At 2048 bits, signing requires 4.30 ms on the i5-1035G7, 3.92 ms on the i5-1145G7, and 2.79 ms on the i9-13900H, a reduction of approximately 35% from the oldest to the most recent micro-architecture.

Verification remains significantly faster than signing at all key sizes and across all platforms. This asymmetry is a structural property of RSA: the public exponent $e = 65537 = 2^{16} + 1$ has a sparse binary representation requiring only 17 squarings and 2 multiplications in the square-and-multiply algorithm, regardless of the key size. At 2048 bits, verification times are 0.56 ms, 0.51 ms, and 0.37 ms on the three platforms respectively, confirming that architectural improvements affect verification proportionally but that the absolute latency remains negligible across all tested configurations.

Impact of CPU Architecture on RSA Workload

The i9-13900H consistently outperforms both i5 variants in signing latency. The improvement is most pronounced at 1024 bits (0.71 ms vs. 1.34 ms on the i5-1035G7, a 47% reduction); the direction remains consistent across all key sizes, though the margin varies, dropping to 28% at 3072 bits before recovering to 45% at 4096 bits. This is attributable to the higher single-core boost frequency of the Raptor Lake architecture (5.40 GHz), its wider integer execution pipeline, and its improved Montgomery modular multiplication throughput relative to Ice Lake and Tiger Lake.

The i5-1145G7 (Tiger Lake) offers a moderate improvement over the i5-1035G7 (Ice Lake) at 2048 bits (3.92 ms vs. 4.30 ms, an 8.8% reduction), consistent with the incremental instructions per clock (IPC) and frequency gains between those two micro-architectural generations.

Key Size Selection

A 1024-bit key is no longer considered secure by current NIST standards, as advances in the general number field sieve have reduced its effective security margin below an acceptable threshold. A 4096-bit key provides a higher security margin but introduces a signing latency exceeding 13 ms even on the fastest tested platform, which is excessive for an interactive application.

The 2048-bit key represents the best compromise across all tested platforms: it meets the NIST-recommended 112-bit security level, generates in under 1 s on Tiger Lake and Raptor Lake hardware (a one-time cost), and signs in under 5 ms on all evaluated processors, well within acceptable bounds for a command-line tool.

3.5 Signature Size Invariance

The experiment in `experiments/exp_sig_size.py` generates a 2048-bit RSA key pair, signs five payloads of increasing size (1 KB to 10 MB) using `pkcs1_15.new(key).sign(SHA256.new(data))`, and records the byte length of each resulting signature.

Table 3.11 shows that the RSA signature size does not depend on the size of the signed document. The same holds on every evaluated processor, since this is a mathematical invariant of the scheme rather than a platform-dependent characteristic.

Table 3.11: RSA signature size as a function of input file size (2048-bit key)

File Size	Signature Size (bytes)
1 KB	256
10 KB	256
100 KB	256
1 MB	256
10 MB	256

The constant output size follows directly from the construction of the scheme. The signing operation applies modular exponentiation to the PKCS#1 v1.5-padded SHA-256 digest, producing a result whose byte length is fixed by the modulus: $2048/8 = 256$ bytes. Because the hash function reduces any document to a fixed-size digest before signing, the

length of the original document has no effect on the signature. Bandwidth and storage overhead are therefore constant regardless of document size. The same invariant holds for any RSA key size: a 4096-bit key would produce 512-byte signatures (4096/8 = 512), and a 1024-bit key would produce 128-byte signatures, in both cases independently of the document length.

3.6 Tamper Detection Test

3.6.1 Experimental Setup

A digital signature system has to detect unauthorised modifications to a signed document. The experiment in `experiments/exp_tamper.py` systematically exercises this property across seven test cases (one unmodified baseline and six modification scenarios) and three active forgery attempts modelling a man-in-the-middle (MITM) adversary. A 512-byte pseudo-random payload is generated with `os.urandom` and signed with a freshly generated 2048-bit RSA key pair using SHA-256 and PKCS#1 v1.5 padding. The payload size is intentionally modest: since the RSA signature scheme operates on the fixed-length SHA-256 digest rather than on the raw document, the tamper detection properties of the system are mathematically independent of the payload length. A 512-byte payload is therefore fully representative of the scheme's behaviour for documents of any size, as confirmed by the signature size invariance demonstrated in Section 3.5. Seven variants of the payload are then constructed and submitted to `pkcs1_15.new(pub_key).verify()`: the original unmodified data, and six tampered variants representing different classes of modification. In addition, three active forgery attempts are simulated to model a realistic MITM threat model in which the attacker intercepts the data in transit but does not possess the sender's private key. The verification function raises an exception on failure; the script catches it and records the result.

3.6.2 Results

Table 3.12 lists each test case, the number of SHA-256 hash bytes that changed, and the verification outcome.

Table 3.12: Passive modification scenario results

Test	Modification Applied	Hash Bytes Changed	Verification Result
1	Original file (no modification)	0/32 (0%)	VALID
2	Flip first byte (`XOR 0xFF`, 8 bits)	32/32 (100%)	INVALID
3	Flip one bit (least significant bit, LSB, of first byte)	32/32 (100%)	INVALID
4	Append one zero byte (`b'\x00'`)	32/32 (100%)	INVALID
5	Remove last byte	32/32 (100%)	INVALID
6	Replace 10 bytes in the middle with random data	32/32 (100%)	INVALID
7	Replace entirely with random content (same length)	31/32 (97%)	INVALID

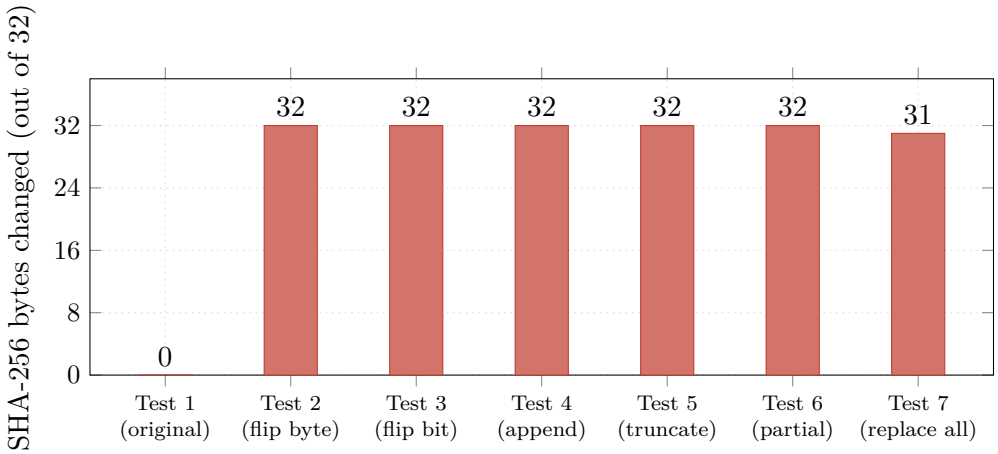


Figure 3.6: SHA-256 bytes changed per test scenario (out of 32). All tampered variants trigger near-total diffusion.

Table 3.13: Active MITM forgery attempt results

Attempt	Forgery Strategy	Verification Result
F1	Random 256-byte blob submitted as forged signature	INVALID
F2	Tampered data signed with MITM's own RSA-2048 private key	INVALID
F3	Original sender signature reused against tampered data	INVALID

3.6.3 Analysis

Passive Modification Scenarios

The system correctly validates the unmodified document (Test 1) and rejects all six tampered variants without exception. These results confirm the avalanche effect of SHA-256: even a single flipped bit in the input (Test 3) produces a completely different 256-bit digest, causing the hash recovered from the signature to diverge from the freshly computed hash of the received document, and verification fails. In Tests 2 through 7, 31 to 32 out of 32 hash bytes change, illustrating the near-total bit diffusion guaranteed by the SHA-256 compression function.

The seven test cases cover the principal classes of document modification encountered in practice: byte-level corruption (Test 2), single-bit corruption (Test 3), length extension (Test 4), truncation (Test 5), partial substitution (Test 6), and complete content replacement (Test 7). The consistent INVALID result across all tampered inputs confirms that the implemented system satisfies the integrity requirement of a digital signature scheme.

Active Forgery Attempts

The three MITM forgery scenarios target a stronger adversarial model in which the attacker actively tries to substitute a valid signature alongside the tampered payload.

Attempt F1 demonstrates that signature forgery by random guessing is computationally infeasible: the RSA signature space contains approximately 2^{2048} possible values (the modulus n is a 2048-bit semiprime, not exactly a power of two), making a random 256-byte blob statistically indistinguishable from zero probability of success.

Attempt F2 addresses the case in which the MITM possesses a valid RSA-2048 key pair of their own. Although the forged signature is cryptographically correct with respect to the MITM's private key, verification is performed against the sender's public key, and the two keys are mathematically unrelated. The verification therefore fails, confirming that the binding between signer identity and key material is enforced by the scheme.

Attempt F3 models the replay scenario: the MITM reuses the original sender's signature unmodified but pairs it with the tampered payload. Because the signature encodes the hash of the *original* document, the hash recomputed from the tampered payload does not match the hash embedded in the signature, and verification fails.

All three forgery attempts confirm that, without the sender's private key, no computationally feasible forgery strategy succeeds against RSA-2048 with SHA-256 and PKCS#1 v1.5 padding.

3.7 Conclusion

Each design decision in this chapter is backed by measurement data, and the results show how CPU micro-architecture shapes cryptographic workload performance. All benchmark results are single-run measurements; inter-platform differences of less than approximately 5% should be treated as indicative rather than statistically conclusive.

SHA-256 was selected as the hash function on the basis of its security properties and competitive performance across all three evaluated platforms (Tables 3.2–3.5). MD5 and SHA-1 were eliminated due to demonstrated collision vulnerabilities (Table 3.6); SHA-512 was eliminated due to unnecessary computational overhead. Across the Ice Lake, Tiger Lake, and Raptor Lake micro-architectures, SHA-256 consistently delivered competitive throughput among the secure candidates, with the i9-13900H reducing SHA-256 latency at 10 MB by approximately 52% relative to the i5-1035G7. On the i5-1145G7 under single-core affinity pinning, SHA-256 records 6.804 ms, below the i5-1035G7's 9.012 ms, and no platform anomaly was observed.

A 2048-bit RSA key gives the best balance between the NIST-recommended 112-bit security level and acceptable signing latency. The cross-CPU analysis (Table 3.10) demonstrated that signing latency at 2048 bits ranges from 2.79 ms on the i9-13900H to 4.30 ms on the i5-1035G7, confirming that the scheme performs well across the full range of evaluated hardware. The signature size experiment (Table 3.11) confirmed that the scheme introduces constant-size overhead of 256 bytes regardless of document length or processor, a property that extends by the same invariant to any RSA key size.

The tamper detection tests (Tables 3.12 and 3.13), covering seven test cases (one unmodified baseline and six modification scenarios) and three active MITM forgery attempts, validated that the system correctly rejects all tested classes of unauthorised modification and that no forgery strategy is computationally feasible in the absence of the sender's private key.

GENERAL CONCLUSION

The central question was practical: how to implement RSA digital signatures correctly, securely, and with design choices backed by measurement rather than convention. The work was organised around three sub-problems (hash algorithm selection, padding, and key size), and addressed each one through a combination of theoretical grounding and empirical measurement.

Chapter 1 established the motivation. Digital files offer no inherent proof of origin or integrity, and the Man-in-the-Middle attack demonstrates exactly how that gap can be exploited. RSA digital signatures close it by tying a mathematically unforgeable value to the exact bytes of a document, providing authentication, integrity, and non-repudiation simultaneously.

Chapter 2 built the theoretical foundation. Starting from modular arithmetic and the Euler–Fermat theorem, the RSA key generation procedure, signing operation, and verification operation were derived and proved correct. The necessity of PKCS#1 v1.5 padding was established, and the four known cryptanalytic attack routes against RSA were analysed, showing that the security of the scheme reduces to the intractability of integer factorisation.

Chapter 3 provided the empirical justification for each design decision. The hash benchmark showed that SHA-256 outperforms both MD5 and SHA-512 on all three evaluated platforms while remaining the faster of the two candidates free of known practical attacks; MD5 and SHA-1 were eliminated due to demonstrated collision vulnerabilities. The key size benchmark confirmed that 2048-bit RSA achieves the NIST-recommended 112-bit security level with a signing latency of approximately 3–5 ms across evaluated hardware, making it the right balance between security and performance. The signature size experiment verified that the scheme introduces a constant 256-byte overhead regardless of document length, which keeps bandwidth and storage costs predictable. Finally, the tamper detection tests confirmed that the system correctly rejects every tested class of unauthorised modification (bit-level corruption, length extension, truncation, partial substitution, and complete content replacement), validating the avalanche effect of SHA-256 in a real signing context.

The implemented system meets all four objectives stated in Chapter 1: it generates 2048-bit RSA key pairs, signs arbitrary files with SHA-256 and PKCS#1 v1.5 padding, verifies signatures through a command-line interface, and its design choices are backed by measured data rather than convention.

Three extensions are worth pursuing. The padding scheme could be upgraded from PKCS#1 v1.5 to the Probabilistic Signature Scheme (PSS), which offers a tighter provable security reduction. The command-line interface could be extended into a graphical tool

or integrated into a document-management workflow. Finally, the system could be benchmarked against the Elliptic Curve Digital Signature Algorithm (ECDSA), which provides comparable security to 2048-bit RSA with significantly smaller key and signature sizes, to determine whether the trade-off is worthwhile for resource-constrained environments.

BIBLIOGRAPHY

[1] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976. [pp. VII, 1, and 2]

[2] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. [pp. VII, 1, 2, 3, 6, 7, 8, 9, 10, 12, 13, 14, 15, and 16]

[3] Robert W. Shirey. Internet Security Glossary, Version 2. RFC 4949, Internet Engineering Task Force, August 2007. Available at https://tools.ietf.org/html/rfc4949. [pp. VII and 4]

[4] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Internet Engineering Task Force, August 2018. Available at https://www.rfc-editor.org/rfc/rfc8446. [p. 3]

[5] Elaine Barker. Recommendation for Key Management: Part 1 – General. NIST Special Publication 800-57 Part 1 Rev. 5, National Institute of Standards and Technology, May 2020. Available at https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final. [pp. 3, 5, and 14]

[6] European Parliament and Council of the European Union. Regulation (EU) No 910/2014 on electronic identification and trust services for electronic transactions in the internal market (eIDAS), July 2014. Official Journal of the European Union, L 257/73. [p. 3]

[7] Jim Schaad, Blake Ramsdell, and Sean Turner. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification. RFC 8551, Internet Engineering Task Force, April 2019. Available at https://www.rfc-editor.org/rfc/rfc8551. [p. 3]

[8] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, Internet Engineering Task Force (IETF), November 2016. Available at https://tools.ietf.org/html/rfc8017. [pp. 3 and 5]

[9] Tatu Ylönen and Chris Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252, Internet Engineering Task Force, January 2006. Available at https://www.rfc-editor.org/rfc/rfc4252. [p. 4]

[10] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. In *Advances in Cryptology – CRYPTO 2004*, pages 199–213. Springer, 2004. First practical collision attack on MD5. [p. 5]

[11] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. CWI Amsterdam & Google Research, February 2017. Available at https://shattered.io/. [p. 5]

[12] National Institute of Standards and Technology. Secure Hash Standard (SHS). Technical Report FIPS PUB 180-4, NIST, August 2015. Available at https://csrc.nist.gov/publications/detail/fips/180/4/final. [p. 5]

[13] Himani Sharma and Arun Badoga. A study on RSA algorithm for cryptography. *International Journal of Trend in Scientific Research and Development*, 12(1):37–42, 2020. [pp. 9, 10, 11, 12, 13, 14, 15, and 16]