Project 2: The Speed of Digital Signatures

Cybersecurity MSc: Applied Cryptography

Pedro Costa (130720)   Henrique Dias (131144)

November 2025

Abstract

This project evaluates the performance of RSA and ECDSA digital signatures across three programming environments: C (OpenSSL), Python (Cryptography), and Java (Bouncy Castle). We benchmarked signature creation and validation times for multiple key sizes and curve types according to NIST standards. The results highlight the trade-offs between algorithm efficiency and implementation overhead, demonstrating that while C and Python offer superior raw performance, Java provides robust support for complex curves at a slight performance cost.

1 Introduction

Digital signatures provide authenticity, integrity, and non-repudiation, forming the backbone of modern secure communications. As defined in the NIST Digital Signature Standard (FIPS 186-4) [3], these algorithms rely on asymmetric cryptography to bind an identity to a message.

However, the computational cost varies significantly based on the chosen algorithm and key length [7]. This study measures these costs in a controlled environment to understand the impact of:

1. **Algorithm:** Integer factorization (RSA) vs. Elliptic Curve Discrete Logarithm (ECDSA) [4, 5].
2. **Key Size:** Increasing security levels (e.g., RSA-1024 vs. RSA-4096).
3. **Implementation:** Native code (C) vs. Interpreted/JIT (Python/Java).

2 Methodology and Setup

2.1 Hardware Environment

All tests were executed on a standardized machine to minimize hardware variables.

- **CPU:** Intel(R) Core(TM) i7-3615QM CPU @ 2.30GHz (4 Cores, 8 Threads).
- **OS:** Linux (running on Apple Mac mini hardware).
- **Conditions:** Minimal background load; no GUI overhead during measurement loops.

2.2 Measurement Strategy

To ensure accuracy and eliminate Operating System noise (context switching, interrupts), we adopted a **Loop N / Loop M** strategy:

1. **Outer Loop ($N = 5$):** The measurement block is repeated 5 times. We record the *minimum* average time, representing execution closest to the theoretical limit.
2. **Inner Loop ($M = 100$ to 500):** The cryptographic operation is executed M times consecutively to gain precision on the system clock.

Time excludes key generation and hashing (SHA-256), focusing strictly on the asymmetric crypto operations.

2.3 Implementation Logic

The workflow for both algorithms was standardized across all languages (Figures 1 and 2).

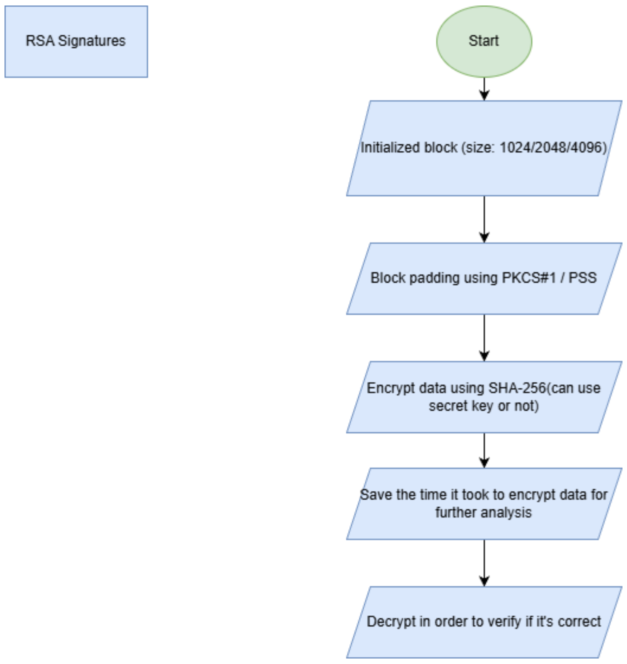
RSA Signatures
Start
Initialized block (size: 1024/2048/4096)
Block padding using PKCS#1 / PSS
Encrypt data using SHA-256(can use secret key or not)
Save the time it took to encrypt data for further analysis
Decrypt in order to verify if it's correct

Figure 1: RSA Workflow.

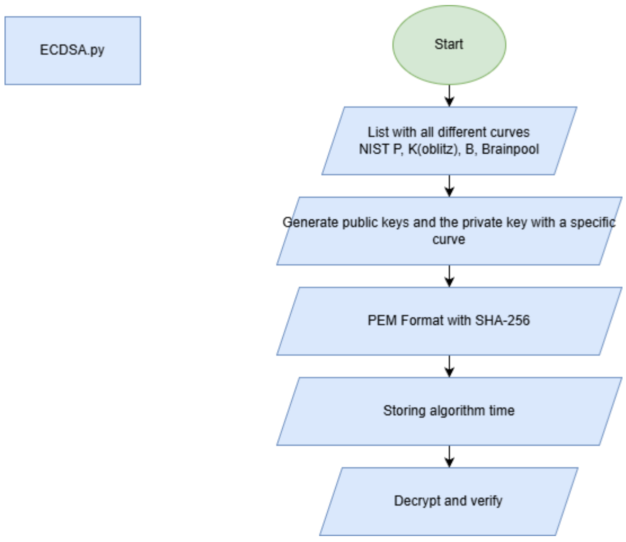
ECDSA.py
Start
List with all different curves NIST P, K(oblitz), B, Brainpool
Generate public keys and the private key with a specific curve
PEM Format with SHA-256
Storing algorithm time
Decrypt and verify

Figure 2: ECDSA Workflow.

3 Implementation Details

3.1 Libraries Used

We adhered to the requirement of selecting distinct libraries and consulted their documentation [1, 2]:

- **C:** `OpenSSL 3.0` (EVP High-level API).
- **Python:** `cryptography` (Bindings for OpenSSL).
- **Java:** `Bouncy Castle Provider` (bcprov-jdk18on).

3.2 Code Snippets (Measurement Loop)

The core logic was replicated in all languages. Below are excerpts illustrating the timing strategy.

```c
// ... Setup (KeyGen and Context) ...
for (int n = 0; n < N_ITERATIONS; n++) {
    clock_t start = clock();
    for (int m = 0; m < M_OPERATIONS; m++) {
        // Only the signing operation is inside the inner loop
        EVP_PKEY_sign(sign_ctx, sig, &len, digest, d_len);
    }
    clock_t end = clock();
    double avg = ((double)(end - start) / CLOCKS_PER_SEC) / M;
    if (avg < min_time) min_time = avg;
}
```

Listing 1: C Measurement Loop (RSA excerpt)

```python
for name, curve_oid in curves_list:
    # ... Key Generation outside the loop ...
    min_sign_time = float('inf')
    for _ in range(N_ITERATIONS):
        start = time.perf_counter()
        for _ in range(M_OPERATIONS):
            private_key.sign(MESSAGE, ec.ECDSA(hashes.SHA256()))
        end = time.perf_counter()
        # ... Calculate Average ...
```

Listing 2: Python Measurement Loop (ECDSA excerpt)

4 Results and Analysis

4.1 RSA Performance

We tested RSA with PKCS#1 v1.5 and PSS padding. Table 1 shows the results for PKCS#1.

Key Size	C (ms)	Python (ms)	Java (ms)
RSA 1024	0.19	0.25	0.73
RSA 2048	1.37	1.36	2.26
RSA 4096	9.18	9.40	14.18

Table 1: RSA Signature Creation Time (PKCS#1).

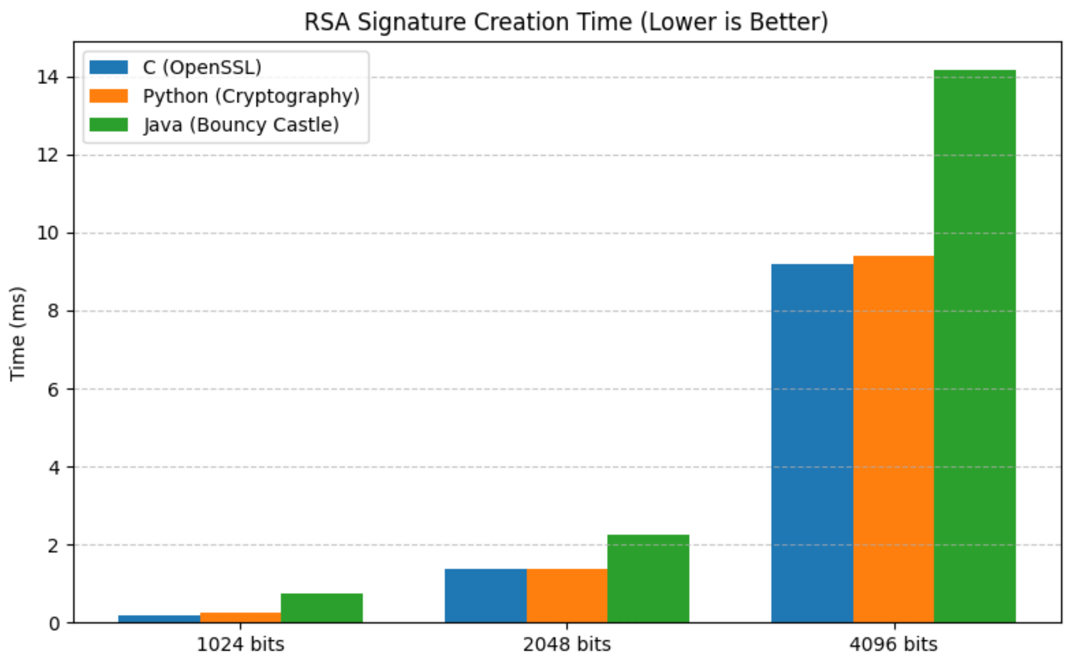

Figure 3: RSA Signature Creation. Performance degrades cubically with key size.

Analysis: C and Python perform almost identically, which is expected as Python's `cryptography` library is a thin wrapper around OpenSSL. Java (Bouncy Castle) is consistently slower (approx. 1.5x slower at 4096 bits), likely due to JVM overhead.

4.2 ECDSA Performance

ECDSA results were obtained for 9 curves. Below we compare representative curves. See [6] for a broader survey on ECC performance.

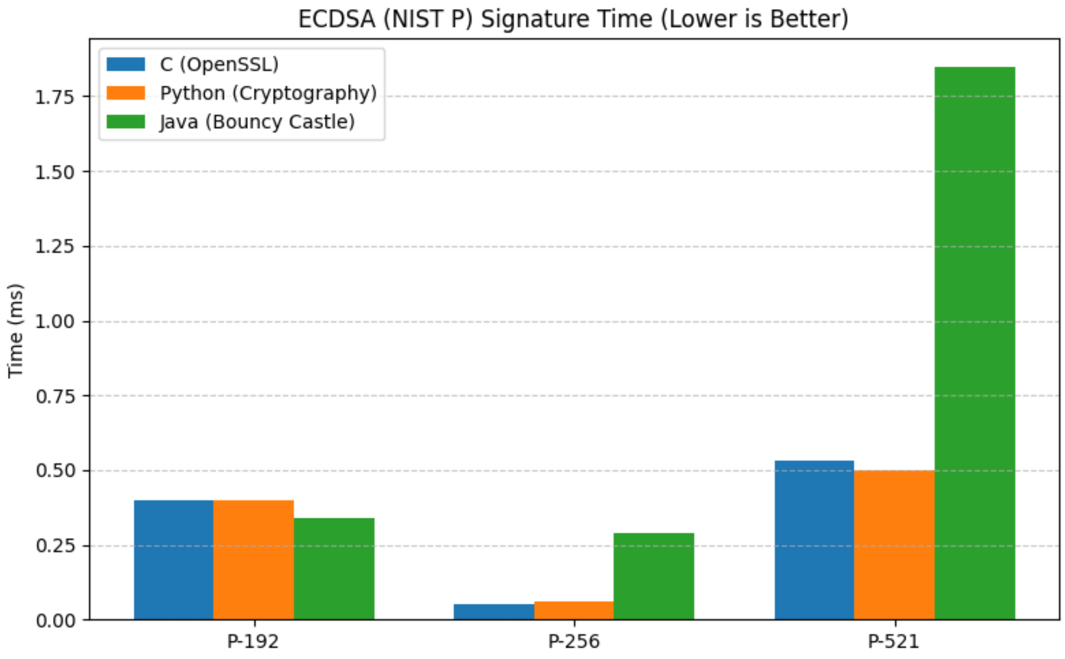

Figure 4: ECDSA Signature Creation comparison.

Analysis: ECDSA signing is extremely fast. Even at high security levels (P-521), it remains under 2ms in all languages, whereas RSA-4096 takes nearly 10-14ms. Interestingly, for Binary curves (e.g., B-571), the Java implementation remained competitive.

4.3 Verification: RSA vs. ECDSA

A crucial observation is the asymmetry of the algorithms.

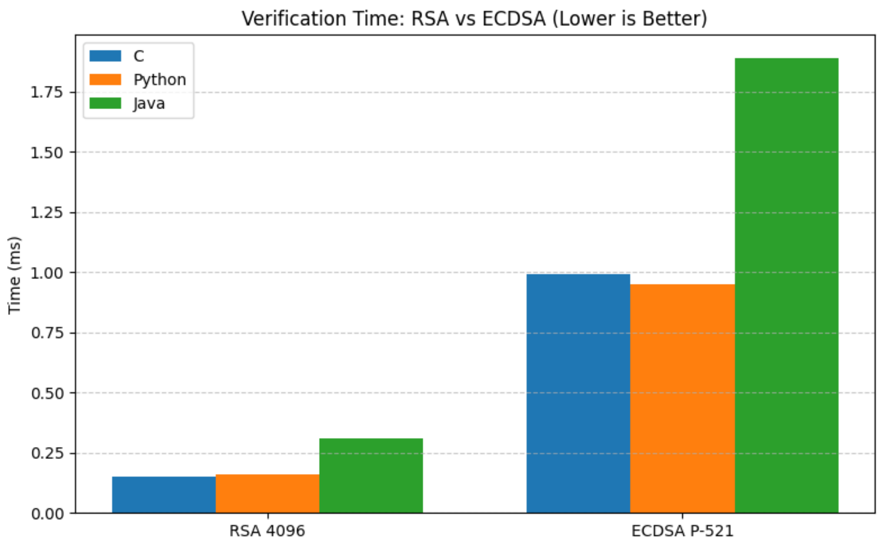

Figure 5: Verification Speed: RSA-4096 vs ECDSA P-521.

While RSA is slow to sign, it is **exceptionally fast to verify** (approx. 0.15ms). ECDSA verification is mathematically more complex than its signing counterpart, taking approx. 1.0ms. This makes RSA ideal for scenarios with one signer and many verifiers (e.g., certificates).

5 Conclusion

This project benchmarked digital signatures adhering to a rigorous N/M loop methodology. We conclude that:

1. **Language Efficiency:** C and Python provide the best raw performance. Java is a viable alternative but incurs a performance penalty.
2. **Algorithm Trade-off:** RSA scales poorly with key size for signing but excels at verification. ECDSA offers consistent, fast signing with much smaller keys.
3. **Library Maturity:** OpenSSL (C) and Bouncy Castle (Java) both correctly implement complex standards like PSS and Koblitz curves.

References

[1] Python Cryptography Authority. *Cryptography Library Documentation.* [Online]. Available: https://cryptography.io/en/latest/

[2] PyCryptodome. *PyCryptodome Documentation.* [Online]. Available: https://pycryptodome.readthedocs.io/en/latest/index.html

[3] National Institute of Standards and Technology (NIST). *FIPS 186-4: Digital Signature Standard (DSS).* July 2013. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

[4] F5 DevCentral Community. *Elliptic Curve Cryptography Overview.* YouTube, 2015. Available: https://www.youtube.com/watch?v=NF1pwjL9-DE

[5] Computerphile. *Elliptic Curves - Computerphile.* YouTube, 2018. Available: https://www.youtube.com/watch?v=dCvB-mhkT0w

[6] ScienceDirect. *Article on Elliptic Curve Cryptography (PII: S1574013722000648).* Elsevier, 2022. Available: https://www.sciencedirect.com/science/article/abs/pii/S1574013722000648

[7] Wikipedia. *Digital Signature.* [Online]. Available: https://en.wikipedia.org/wiki/Digital_signature