Revista Eletrônica de Iniciação Científica em Computação, 2026, 24:1
ISSN: 3085-8461 • doi: 10.5753/reic.2026.6758

Research Paper

Performance and Security Evaluation of RSA Cryptosystem Implementations on Different Hardware Configurations

Ana Carla Quallio Rosa [Universidade de São Paulo | *anacarlaquallio@ime.usp.br*]
Rodrigo Campiolo [Universidade Tecnológica Federal do Paraná |*rcampiolo@utfpr.edu.br*]
Daniel Macêdo Batista [Universidade de São Paulo |*batista@ime.usp.br*]

Department of Computer Science, University of São Paulo, R. do Matão, 1010, Butantã, São Paulo, SP, 05508-010, Brazil.

Abstract. RSA remains one of the central cryptographic systems for securing data and systems. This study evaluates implementations of the RSA algorithm across various programming languages and libraries, including C, C++, Java, JavaScript, Python, and Rust. The experiments assessed the performance of key generation, encryption, and decryption operations in two distinct environments: a high-performance setting and another with hardware and memory constraints. Implementations in Rust, C, C++, and Python exhibited lower execution times overall. Analysis of data structures revealed similar patterns across languages, while memory dumps identified fragments of private keys in several implementations, with full key recovery observed in JavaScript implementations. The results provide practical guidance for performance and security considerations in RSA implementations.

Keywords: Performance analysis, Asymmetric cryptography, Programming languages.

Received: 27 October 2025 • **Accepted:** 14 January 2026 • **Published:** 30 January 2026

1 Introduction

In light of the expanding communication landscape and the increasing flow of information across electronic devices, cryptographic systems have become essential for maintaining the authenticity, confidentiality, and integrity of data. By transforming data into unintelligible formats for unauthorized entities, cryptography ensures that only legitimate senders and recipients can access the original content [Stallings, 2008].

One approach uses symmetric-key systems, in which users encrypt and decrypt messages with the same key. However, guaranteeing secrecy requires sharing the key through a channel separate from the one used for information exchange, which can pose challenges.

From this perspective, public-key systems, also known as asymmetric cryptography, were introduced. This method employs two keys: one for encrypting messages that users can share publicly, and another, accessible only to the recipient, for decrypting them. The RSA (Rivest–Shamir–Adleman) algorithm emerged as one of the most widely adopted systems within this cryptographic approach [Imam *et al*., 2021].

Since its publication by Rivest *et al*. [1978], RSA has found applications in various scientific fields, including digital signature generation, website authentication, and transaction security. Nonetheless, in a scenario of constant developments in Information Security, the original RSA algorithm becomes more susceptible to attacks [Imam *et al*., 2021]. Various studies focus on improving RSA; however, there remains a need for a consistent taxonomy to align requirements such as execution time, performance, and security.

Moreover, the security and efficiency of RSA depend not only on the algorithm's mathematical process but also on the execution context and the quality of its implementation across different hardware platforms. With the advancement of the Internet of Things (IoT) and embedded systems, there is an increasing need to evaluate performance in environments with limited processing power and memory.

The main objective of this study, which complements the results presented in Rosa *et al*. [2025], is to evaluate RSA implementations. The specific objectives are to: (i) analyze the execution time of RSA across distinct Application Programming Interfaces (APIs) within the same programming language and across different programming languages; (ii) compare the performance of implementations in two execution environments, one high-performance and one with hardware and memory constraints; (iii) investigate the algorithm modifications adopted by each implementation; and (iv) analyze the data structures used for storing confidential information.

This article is organized into five sections. Section 2 presents the theoretical foundation of the RSA algorithm. Section 3 discusses related work. Section 4 describes the research method and evaluation criteria adopted. Section 5 presents and discusses the results obtained, while Section 6 provides the conclusions and future work.

2 Background

2.1 The RSA Algorithm

RSA is an asymmetric cryptography algorithm based on the difficulty of factoring prime numbers with a large number of digits. According to Imam *et al*. [2021], the most recent applications of standard RSA[1], since its publication in 1978, range from key exchange and digital signatures to any other means of communication where secure transmission between two parties is required.

Concisely, RSA is a block cipher scheme where the plaintext and ciphertext are represented as integers ranging from 0 to $n - 1$, where n is an integer that makes up the public

[1]That is, the algorithm initially proposed by Rivest *et al*. [1978], without mathematical changes.

key [Stallings, 2008]. The mathematical principle of RSA lies in generating a key pair, where the first step is to find two distinct prime numbers, p and q, such that $n = pq$.

The public key is given in the format (n, e) and the private key by the pair (n, d), both mathematically related by Congruence 1:

$$e \cdot d \equiv 1 \pmod{(p-1) \cdot (q-1)} \tag{1}$$

After key generation, the encryption process for a message m (converted to an integer) is performed using the public key pair (n, e), resulting in the ciphertext c, according to Congruence 2:

$$c \equiv m^e \pmod{n} \tag{2}$$

The decryption process is performed using the private key (n, d), recovering the original message m, as shown in Congruence 3:

$$m \equiv c^d \pmod{n} \tag{3}$$

The security of the algorithm stems from the fact that, although the terms n and e are public, it is only possible to find the private key d by factoring n into its prime factors p and q. Since keys with a large number of bits are generally used, factoring numbers of this magnitude becomes computationally infeasible.

2.2 Recommendations for RSA Implementation

Several modifications to RSA are found in the literature, aiming to propose better performance. This raises the question: what criteria determine that a cryptographic algorithm implementation is considered secure?

We can classify RSA implementations by source code nature and abstraction level. Pure or native implementations encode the algorithm directly in a programming language without relying on external libraries. In contrast, implementations using wrappers or high-level libraries encapsulate established cryptographic functions, such as `OpenSSL`, to optimize complex operations.

To ensure interoperability and security across different implementations, standards such as PKCS (Public-Key Cryptography Standards) have been established, defining key syntax, message formats, and digital signature mechanisms. Furthermore, organizations such as ISO (International Organization for Standardization), NIST (National Institute of Standards and Technology), and IEEE (Institute of Electrical and Electronics Engineers) publish guidelines that standardize the use of public-key cryptography. In the case of RSA, the use of padding algorithms, such as OAEP (Optimal Asymmetric Encryption Padding) and PKCS#1 v1.5, is essential to mitigate ciphertext-based attacks and strengthen the encryption scheme against vulnerabilities.

Regardless of the programming language or library, every application that performs encryption operations needs to load the cryptographic key into the system's main memory (RAM) at some point. Although RAM is volatile, its use during software execution creates an exposure window that can be exploited through digital forensic analysis. This analysis is made possible by obtaining memory dumps, which provide a static snapshot of volatile memory. In many cases, keys are stored in memory while the program is running, allowing them to be recovered through pattern-matching and string analysis techniques.

Developers can perform memory dumping at different levels. For a specific process, `gcore` extracts the memory page information associated with a process identifier (PID).

In this context, the persistence of cryptographic keys in memory, even after the process ends, is considered a vulnerability [Halderman *et al*., 2009]. Consequently, an essential requirement for the secure implementation of cryptographic algorithms is proper memory management, including practices such as overwriting and explicitly cleaning memory regions used by sensitive keys.

3 Related Work

Since the publication by Rivest *et al*. [1978], researchers have proposed several modifications to RSA.

Islam *et al*. [2018], for example, generalized RSA to use n prime numbers, increasing complexity and computational cost. In this context, aiming to improve efficiency, Alzaher *et al*. [2022] proposed parallelizing an RSA approach that uses two public keys to encrypt the message twice and, subsequently, two private keys to decrypt it on the receiver side. The RSA implementation did not use cryptographic libraries. Experimental results showed that the parallelized version performed better than the sequential version.

Following a line of RSA enhancement via heuristic algorithms, Maalavika *et al*. [2024] implemented a version using Cuckoo Search, a search algorithm for finding prime numbers that generate public keys more quickly. The authors compared this approach with traditional RSA and Elliptic Curve Cryptography (ECC), considering the time required to break 256, 512, and 1024 bit keys. The results indicated that the proposed algorithm showed greater resistance to key breaking.

Other studies have compared RSA with algorithms such as ECC and Diffie-Hellman (DH). Jintcharadze and Abashidze [2023] and Dalal *et al*. [2024] generally concluded that ECC achieved better performance and efficiency, in terms of execution time and key size, at equivalent security levels.

In the context of vulnerabilities, Afrose *et al*. [2019] proposed a benchmark to detect misuse of cryptographic APIs in Java and Android applications, although the focus was not on the implementation details of algorithms.

Related work presents adaptations to improve the efficiency and security of RSA. However, the existing literature lacks an approach that explores variations in algorithm implementations and the implications of adopting different libraries within the same programming language and across various languages.

4 Materials and Methods

The research method adopted in this work aims to conduct an experimental evaluation of the performance and security of different RSA algorithm implementations. The study consists of the following steps:

1. Selection of languages/libraries and RSA implementation: This step involves choosing programming lan-

guages and cryptographic libraries for RSA implementation;
2. Performance comparison between implementations: This step compares the cryptographic processes across the selected libraries, covering key generation, encryption, and decryption. Measurements are performed through microbenchmarking, considering two distinct execution environments: one high-performance and another with hardware and memory constraints;
3. Interpretation of results, analysis of structures, and memory dump: This step aims to analyze and discuss the charts and tables obtained by microbenchmarking. In this sense, it presents an overview of the selected languages and the two evaluated execution environments. Furthermore, the patterns and data structures used to store confidential information are examined based on each library's documentation. Finally, a memory dump is analyzed with the `gcore` tool to verify the presence of private key fragments.

4.1 Execution Context and Selection

The experiments were conducted on two distinct platforms to evaluate RSA performance under different processing conditions.

The first platform represents a high-performance environment. It is a machine with an AMD Ryzen 5 processor (64-bit), operating at 2.1 GHz, 12 GB of RAM, a 240 GB NVMe SSD, and the Linux Mint 22.1 Xia - 64-bit operating system. The second corresponds to an environment with hardware and memory constraints, based on a Raspberry Pi 3 Model B, equipped with a Broadcom BCM2837 processor (ARMv8 Cortex-A53 Quad-Core, 64-bit) operating at 1.2 GHz and 1 GB of RAM.

In the first execution environment, experiments were conducted using the Xfce graphical interface. We employed the following compilers and interpreters: `GCC` 11.4.0 (C/C++), `OpenJDK` 11.0.27 (Java), `Node.js` 20.19.0 (JavaScript), `Python` 3.10.1, and `rustc` 1.86.0 (Rust). All experiments were conducted on a single processing core, utilizing CPU affinity to prevent interference from concurrent system activities.

In the second execution environment, the operating system was configured for maximum performance. The processor frequency was fixed at 1,200 MHz to minimize time–frequency variability in the measurements. We used `GCC` 14.2.0 (C/C++), which was pre-installed in the environment. As in the first environment, the same single-core execution policy was enforced. No form of parallel or multi-core execution was allowed in this evaluation, ensuring comparability between environments.

We evaluated RSA in six programming languages: C, C++, Java, JavaScript, Python, and Rust. We selected these languages based on the TIOBE [TIOBE, 2024] and PYPL [PYPL, 2024] Indices, which rank them among the most widely used in general-purpose contexts. The inclusion of Rust is justified because it is a systems language designed with a focus on memory safety and concurrency, offering a relevant point of comparison for the study. For each language, cryptographic libraries with the highest adoption and available documentation were chosen.

Table 1 presents the relationship between the libraries and the microbenchmarking tools used in each language.

Table 1. Relationship between libraries and benchmark tools per language

Language	Libraries	Tool
C	OpenSSL [2024], Libgcrypt [2023]	Standard time library
C++	Botan [2025], Crypto++ [2023]	Google Benchmark [2025]
Java	Bouncy Castle [2013], Java Cryptography [2025]	JMH [2025]
JavaScript	Crypto [2025], Forge [2024]	Standard time library
Python	Cryptography [2024], PyCryptodome [2024]	Pytest [2025]
Rust	Rust OpenSSL [2025], Rust Crypto [2024]	Cargo Bench [2025]

It is noteworthy that languages such as C++, Java, Python, and Rust use established microbenchmarking tools (`Google Benchmark`, `JMH`, `Pytest`, and `Cargo Bench`, respectively). These tools improve measurement reliability by incorporating warm-up phases that stabilize the execution environment prior to data collection, resulting in reduced variance across samples.

For C and JavaScript, no widely adopted language-specific microbenchmarking framework comparable to those used in managed or systems languages is available. Therefore, execution time was measured using standard timing libraries, and statistical metrics were computed manually following the same experimental protocol.

The use of different benchmarking tools across languages reflects a methodological choice rather than a limitation. Currently, there is no generic benchmarking framework capable of providing equally accurate and reliable measurements across compiled, interpreted, and JIT-compiled languages. Languages such as Java, C++, and Rust require dedicated tools to correctly account for just-in-time compilation, garbage collection, compiler optimizations, and warm-up effects.

Accordingly, for each language, we adopted the most widely accepted benchmarking approach within its ecosystem, ensuring that all libraries implemented in the same language were evaluated under identical conditions. While absolute timing values may vary across tools, the relative performance comparison between libraries within the same language remains valid and consistent.

We acknowledge that differences in measurement precision may affect results at the microsecond scale. However, employing a single generic benchmarking tool would likely introduce greater systematic bias, particularly for languages that rely on runtime optimizations, thereby reducing rather than improving the validity of the comparative analysis. This study, therefore, focuses on comparative performance trends rather than absolute timing values.

4.2 Performance Evaluation

The performance evaluation stage consisted of the individual execution of key generation, encryption, and decryption processes in each library and the two considered execution environments.

To ensure consistent results, we adopted some standards. In all runs, we used a message file in `.txt` format containing 90 bytes, which corresponds to the maximum limit allowed for encryption with a 2048-bit RSA key when applying the OAEP padding algorithm provided by the `Forge` library in JavaScript.

For the encryption and decryption workloads, the 2048,

3072, and 4096-bit RSA keys were generated once using `OpenSSL` and subsequently reused across all evaluated languages and libraries. This approach ensures that all implementations operate on identical key material, reducing variability caused by differences in randomly generated key parameters.

In contrast, for the key generation workload, each library generated its own RSA keys according to its default implementation and configuration. This allows the evaluation of the intrinsic performance of key generation routines provided by each library.

Each process was executed with sets of 10, 100, 1000, and 10000 iterations. However, the 10-iteration set was excluded from the analysis because it exhibited higher variance and lower statistical stability. The final results were obtained by averaging five independent executions for each set of iterations.

The maximum message size, in bytes, that can be encrypted with RSA depends on both the key size and the specific implementation of the libraries and the data structures used. For example, for a 2048-bit key, the maximum allowed message size is 256 bytes. For a 4096-bit key, the limit is 512 bytes. This maximum size accounts for the applied padding algorithm, which occupies part of the available space for the message.

We conducted the experiments with keys up to 4096 bits, the largest size supported by the `Rust Crypto` library. Most other libraries, however, support keys up to 16384 bits. Table 2 shows the relationship between the maximum allowed message size, key length, and the padding algorithm used by each library.

Most of the evaluated libraries accept keys in PEM (Privacy Enhanced Mail) format, with some exceptions. The `Crypto++` library, in the C++ language, exclusively supports the DER (Distinguished Encoding Rules) format, which requires converting the keys originally generated in PEM format by `OpenSSL`. The `Libgcrypt` library, written in C, only supports keys in the S-Expression format[2].

To overcome this limitation, we used the `monkeysphere` package to extend OpenPGP (`Libgcrypt`) functionality and convert PEM keys to the GPG format. Subsequently, a Python script was developed to extract the public and private key data and convert them to the S-Expression format, enabling the keys to be loaded and used in the experiments.

4.3 Memory Dump Analysis

The final step consisted of memory-dump analysis to identify possible fragments of the private key.

The memory dump was not captured at a precisely synchronized instruction-level moment. Instead, dumps were obtained from the running process after the execution of cryptographic workloads that require access to the private key, particularly after decryption operations. Our goal was to assess whether private key material persists in memory beyond its immediate use, indicating insufficient memory cleansing by the library.

Considering that libraries can apply manipulations on the keys, such as conversion to byte vectors or sequence inversion, the search strategy was composed of two complementary approaches:

1. String search: the command `strings dump_file | grep key_fragment` was used to directly locate key fragments in readable formats (PEM, DER, or S-Expression);
2. Byte search: a Python script was developed capable of applying byte manipulations (conversion and inversion) and, subsequently, searching the memory dump for corresponding patterns. To test fragmentation, the converted key was subdivided into 16 byte sequences and searched in both the original and inverted versions.

This analysis aimed to verify the effectiveness of the memory cleaning adopted by each implementation and the possibility of full key recovery by an attacker with access to the memory dump.

5 Results

First, we measured the average execution time of key generation, encryption, and decryption across all selected libraries and execution environments. Next, we examined the implementation patterns and the security mechanisms each library adopted. Finally, we ran `gcore` on the active process after cryptographic operations to check for residual private-key fragments in memory.

5.1 Average Execution Time

The average time analysis was divided into three main stages: key generation, encryption, and decryption.

To facilitate interpretation of the results, the experimental data were organized by cryptographic process and key size, across two execution environments: `Desktop` (high performance) and `Raspberry Pi` (hardware and memory constraints).

Figure 1 presents the classification of times related to key generation. Subsequently, Figure 2 shows the results of the encryption process. Finally, Figure 3 illustrates the average times obtained in the decryption process.

Figures 1, 2, and 3 show that the performance of the libraries varies according to the cryptographic process and the execution environment.

In the `Desktop` environment, the `Crypto++` (C++) and `Cryptography` (Python) libraries presented the lowest average key generation times. In the encryption and decryption processes, the `Rust OpenSSL` (Rust), `Cryptography` (Python), and `OpenSSL` (C) libraries achieved the best performance.

In the `Raspberry Pi` environment, the lowest key generation times were obtained by the `Crypto++` and `OpenSSL` libraries. In the encryption and decryption processes, the `Rust OpenSSL`, `Cryptography`, and `OpenSSL` libraries again stood out, with lower average times than the others. In contrast, the `Forge` (JavaScript) library generally performed worst in the experiments, both on the `Desktop` and the `Raspberry Pi`.

The key generation process has a higher standard deviation than encryption and decryption because the time depends on the selected exponents and the mathematical operations involved, which may differ across executions.

[2]Data structures used internally in asymmetric cryptographic implementations.

Table 2. Relationship between maximum message size (bytes) allowed per library, key, and padding algorithm

Library	2048 bits	3072 bits	4096 bits	Padding Algorithm
OpenSSL (C)	245	373	501	PKCS#1 v1.5
Libgcrypt (C)	214	318	470	None by default (OAEP applied)
Botan (C++)	190	318	446	OAEP
Crypto++ (C++)	214	342	470	OAEP
Bouncy Castle (Java)	192	288	384	None by default
Java Crypto (Java)	183	279	375	PKCS#1 v1.5
Crypto (JavaScript)	213	342	470	OAEP
Forge (JavaScript)	90	159	223	OAEP
Cryptography (Python)	190	318	446	OAEP
Pycryptodome (Python)	214	318	470	OAEP
Rust OpenSSL (Rust)	245	373	501	PKCS#1 v1.5
Rust Crypto (Rust)	245	373	501	PKCS#1 v1.5

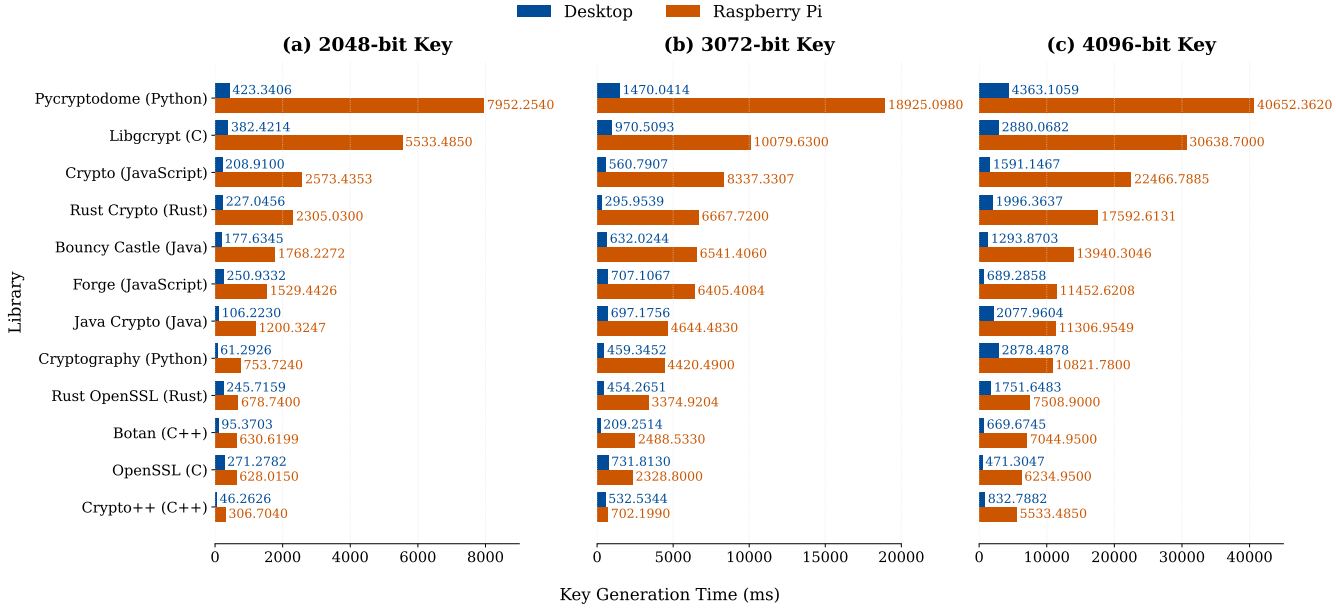

Figure 1. Classification of the key generation process

5.2 Analysis of Selected Library Implementations

The analysis of RSA implementations revealed that the `OpenSSL` library is widely used as a wrapper in Rust (`Rust OpenSSL`), Python (`Cryptography`), and JavaScript (`Crypto`). This integration explains the superior performance observed in Rust, C, and Python in encryption and decryption operations.

In terms of standardization, most libraries follow the protocols described in RFCs 8017 and 5208. The X.509 standard is predominant for public keys, except for `PyCryptodome`, which uses PKCS#1, while PKCS#8 is adopted for private keys. ASN.1 encoding is the most common, except for `Libgcrypt`, which uses the S-Expression structure. Furthermore, some libraries, such as `Libgcrypt` and `Crypto`, use the same data structure for both encryption and decryption, whereas others, such as `Crypto++` and `Bouncy Castle`, use distinct structures for each operation.

The security techniques[3] implemented varies among the libraries. `Libgcrypt`, `Botan`, and `Rust Crypto` employ obfuscation methods, such as the introduction of random factors, to mitigate timing attacks. These techniques add processing overhead but increase resistance to side-channel attacks. The `Crypto++` library, on the other hand, distinguishes itself by using a public exponent of 17 rather than the standard 65537, which can affect performance and compatibility across systems.

In contrast, some vulnerabilities were also observed. In the `Cryptography` (Python) library, the memory-cleaning process is limited by the immutability of internal language structures, such as bytes. This factor prevents the secure destruction of private keys after use, leaving the program vulnerable to attacks exploiting memory residue. Furthermore, the use of the PKCS#1 v1.5 standard in online contexts is discouraged, as this format does not allow verification of decryption validity, which can facilitate padding attacks.

In the `Forge` (JavaScript) library, the vulnerability is

[3] Libraries that implement security techniques, such as obfuscation or additional key validation, tend to incur higher computational costs. In interpreted languages, risks related to memory management or random number generation are generally mitigated through wrappers that encapsulate native libraries.

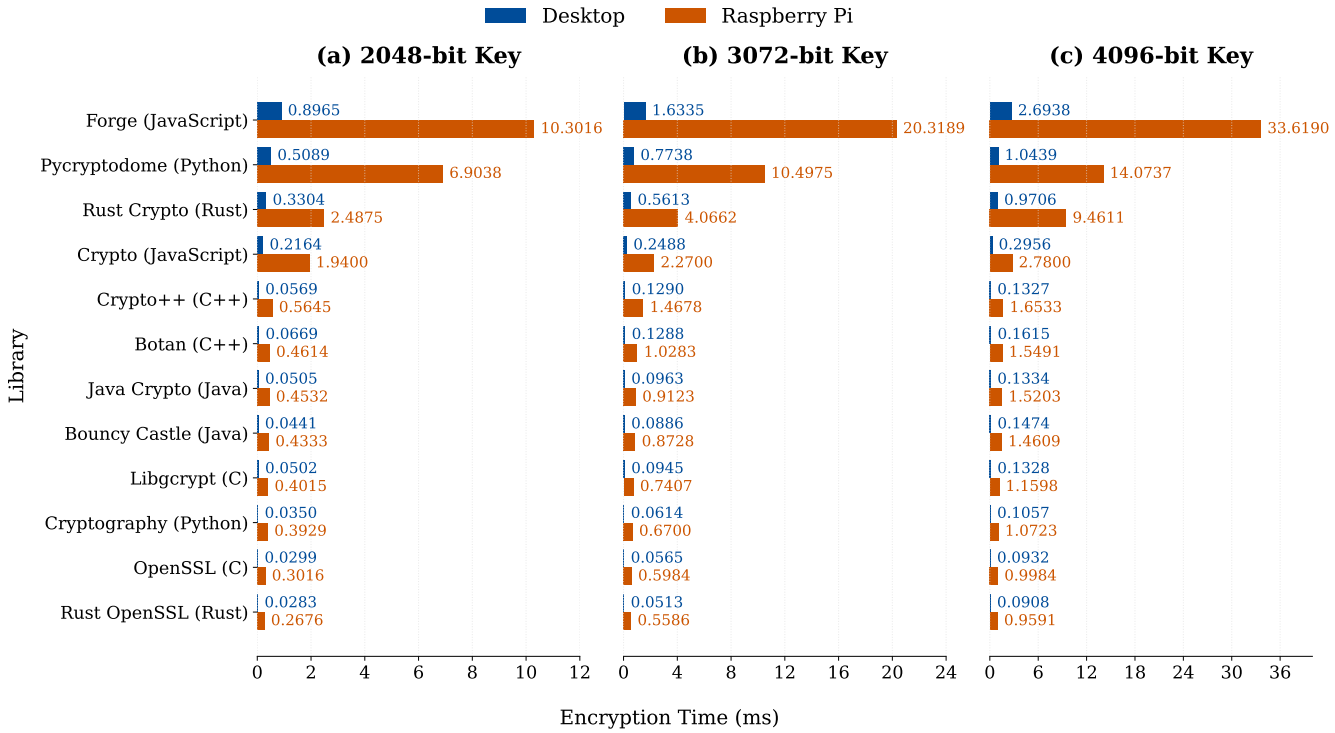

Figure 2. Classification of the encryption process

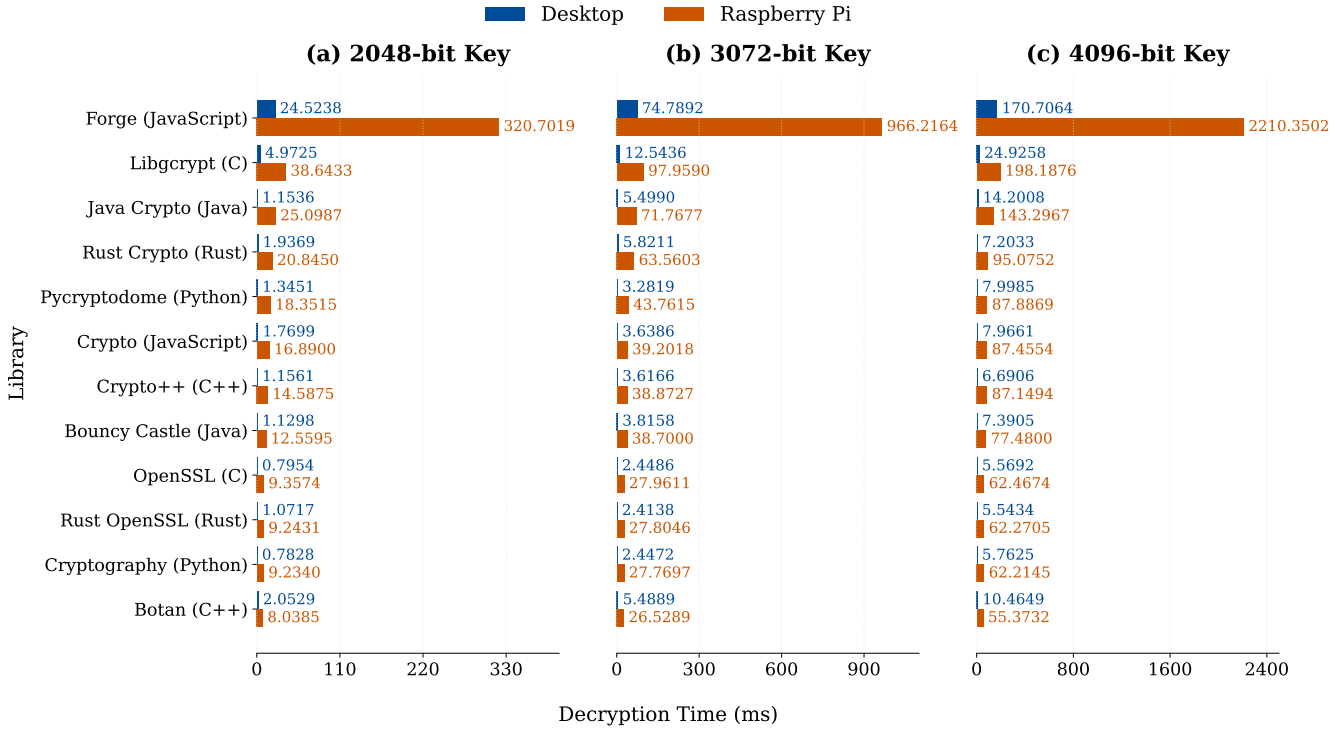

Figure 3. Classification of the decryption process

associated with random number generation, as the JavaScript environment does not, by default, provide cryptographically secure sources of randomness. This problem can lead to predictable key generation. `Forge` implements a solution based on the Fortuna algorithm[4].

Table 3 summarizes the data structures used in the encryption and decryption operations.

Table 3. Relationship of implemented data structure names by libraries

Library	Encryption	Decryption
OpenSSL	`EVP_PKEY_CTX`	`EVP_PKEY_CTX`
Libgcrypt	`gcry_sexp_t`	`gcry_sexp_t`
Botan	`PK_Encryptor_EME`	`PK_Decryptor_EME`
Crypto++	`RSA::PublicKey`	`RSA::PrivateKey`
Bouncy Castle	`RSAKeyParameters`	`RSAPrivateCrtKeyParameters`
Java Crypto	`PublicKey`	`PrivateKey`
Crypto	`Buffer`	`Buffer`
Forge	`Buffer`	`Buffer`
Cryptography	`RSAPublicKey`	`RSAPrivateKey`
Pycryptodome	`RsaKey`	`RsaKey`
Rust OpenSSL	`pkey::Private`	`pkey::Private`
Rust Crypto	`RsaPublicKey`	`RsaPrivateKey`

In general, implementations aim to balance performance and security. The use of native libraries, such as `OpenSSL`, ensures optimization and reliability. In contrast, interpreted languages, such as Python and JavaScript, introduce specific vulnerabilities in memory management and random number generation. These differences reflect the direct impact of design choices on the robustness and security of cryptographic implementations.

5.3 Memory Dump Analysis

In the final stage of our analysis, we used the `gcore` tool to generate memory dumps of the running processes.

`OpenSSL` presented private key fragments in memory, in byte format, distributed in distinct blocks. In `Libgcrypt`, due to the use of S-Expressions, it was necessary to manually extract the key components (n, e, d), convert them from hexadecimal to bytes, and search for both the original and inverted forms. We located the public exponent e in both forms.

In the `Botan` and `Crypto++` libraries, we identified fragments in bytes and sections in PEM format. `Bouncy Castle` revealed few traces, making recovery impossible. `Java Crypto` exhibited fragments similar to those of `OpenSSL`.

In the JavaScript libraries `Forge` and `Crypto`, the key was entirely located in memory, in PEM and byte formats. Figure 4 presents the search for a section of the PEM key in the `Forge` dump.

```
$ strings target.24348 | grep "
    MIIEvQIBADANBgkqhkiG9w0BAQEFA"
MIIEvQIBADANBgkqhkiG9w0BAQEFA
```

Figure 4. Search for a section of the private key, in PEM format, in the memory dump generated by the `Forge` library

We detected no recoverable key traces in `Cryptography` and `Rust OpenSSL`. `PyCryptodome` and `Rust Crypto` contained fragments in bytes.

Although `OpenSSL` presents traces, derived implementations such as `Cryptography` and `Rust OpenSSL` seem to mitigate this vulnerability. The full recovery observed in JavaScript libraries suggests a potential vulnerability related to the language runtime, even though it is not explicitly documented. In this case, an attacker with access to the memory dump can decrypt communications and forge digital signatures, directly compromising the confidentiality and authenticity of the data.

When we reduced the block size, we observed that libraries like `OpenSSL` and `Bouncy Castle` store key sections at consecutive offsets, suggesting contiguous blocks. Even so, dispersed parts indicate fragmentation, corruption, or the use of multiple storage formats as a strategy to hinder recovery.

5.4 Quality of Documentation

We identified challenges with the RSA implementation in specific libraries. For example, the `Crypto++` official website remained inactive for much of our research period, and the repository received few updates and contributions. This scenario suggests a possible lack of continuous maintenance by the community, which could lead to long-term vulnerabilities and increased exposure to attacks.

To obtain more detailed security information for RSA implementations, we accessed the websites' version histories. This situation also occurred for the `Bouncy Castle` and `Forge` libraries, which, with the update and reformulation of their documentation, many pages became invalid.

In other cases, we found undocumented changes in the data structures used in RSA implementations or the absence of practical examples of cryptographic algorithms, as in `Bouncy Castle`.

Finally, the library documentation for `Rust Crypto` is in progress. Thus, to obtain more details on implementation and security of the algorithms, the repository issues were consulted.

6 Conclusion

This work aimed to evaluate different implementation approaches for the RSA algorithm.

The analysis of average execution times was conducted in two distinct environments: one with high performance and another with hardware and memory constraints. In the first, C, C++, and Python showed the lowest average times for key generation with key sizes of 2048, 3072, and 4096 bits. In the second environment, C++ and Python maintained the best performance. For encryption and decryption, the results were similar across both environments, with Rust, Python, and C standing out, mainly because they use `OpenSSL` as the foundation of their implementations.

Furthermore, the evaluated libraries demonstrated similarities in the use of standards and data structures for key storage. However, each implementation incorporates specific security mechanisms that influence both performance and code complexity. The memory analysis indicated the presence of private key fragments in different libraries, including `OpenSSL`, `Botan`, `Crypto++`, `Bouncy Castle`, and `Java Crypto`. In JavaScript implementations, we could re-

[4] A pseudorandom number generation algorithm designed for use in cryptographic operations. Documentation available at: `https://www.schneier.com/academic/fortuna/`.

cover the full private key, highlighting a critical vulnerability that can compromise encrypted data. In the other libraries, although fragments appeared, we could not reconstruct a valid key.

The results obtained provide a basis for selecting RSA implementations across various contexts, balancing performance and security. During the research, challenges arose, including a lack of documentation and detailed information about the RSA implementation. Developers should improve these aspects to allow researchers and professionals to perform more precise investigations and deploy solutions securely.

For future work, we suggest evaluating RSA across additional languages and libraries to broaden the comparative overview. Moreover, researchers can deepen the security analysis by examining sensitive variables and data structures in memory and exploring key recovery from fragments, thus mitigating potential vulnerabilities.

Declarations

Funding

Research partially funded by CNPq (proc. 405940/2022-0) and Coordination for the Improvement of Higher Education Personnel – Brazil (CAPES) – Funding Code 88887.954253/2024-00.

Authors' Contributions

ACQR contributed to the conception of this study and performed all experiments. RC contributed to the supervision, methodology, and final review. DMB contributed to the methodology and final review. ACQR is the manuscript's main contributor and writer. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no conflicts of interest.

Availability of data and materials

The developed implementations and generated scripts are available at: `https://github.com/anacarlaquallio/tcc2`.

References

Afrose, S., Rahaman, S., and Yao, D. (2019). Cryptoapi-bench: A comprehensive benchmark on java cryptographic api misuses. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 49–61. DOI: 10.1109/SecDev.2019.00017.

Alzaher, R., Hantom, W., Aldweesh, A., and Allah, N. M. (2022). Parallelizing multi-keys rsa encryption algorithm using openmp. In *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 778–782. DOI: 10.1109/CICN56167.2022.10008237.

Botan (2025). Botan: Crypto and tls for modern c++. Disponível em: `https://botan.randombit.net/`. Acesso em: 21 jan. 2025.

Bouncy Castle (2013). The legion of the bouncy castle. Disponível em: `https://www.bouncycastle.org/`. Acesso em: 12 mar. 2024.

Cargo Bench (2025). Compile and execute benchmarks. Disponível em: `https://doc.rust-lang.org/cargo/commands/cargo-bench.html`. Acesso em: 25 jan. 2025.

Crypto++ (2023). Crypto++ library. Disponível em: `https://www.cryptopp.com/`. Acesso em: 12 mar. 2024.

Crypto (2025). Crypto. Disponível em: `https://nodejs.org/api/crypto.html`. Acesso em: 21 jan. 2025.

Cryptography (2024). Pyca/cryptography. Disponível em: `https://cryptography.io/`. Acesso em: 12 mar. 2024.

Dalal, Y. M., S, S., K, A., Satheesha, T. Y., PN, A., and Somanath, S. (2024). Optimizing security: A comparative analysis of rsa, ecc, and dh algorithms. In *2024 IEEE North Karnataka Subsection Flagship International Conference (NKCon)*, pages 1–6. DOI: 10.1109/NKCon62728.2024.10775183.

Forge (2024). A native implementation of tls in javascript and tools to write crypto-based and network-heavy webapps. Disponível em: `https://github.com/digitalbazaar/forge`. Acesso em: 08 ago. 2024.

Google Benchmark (2025). A library to benchmark code snippets, similar to unit tests. Disponível em: `https://github.com/google/benchmark`. Acesso em: 25 jan. 2025.

Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. (2009). Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98. DOI: 10.1145/1506409.1506429.

Imam, R., Areeb, Q. M., Alturki, A., and Anwer, F. (2021). Systematic and critical review of rsa based public key cryptographic schemes: Past and present status. *IEEE Access*, 9:155949–155976. DOI: 10.1109/ACCESS.2021.3129224.

Islam, M., Islam, M., Islam, N., and Shabnam, B. (2018). A modified and secured rsa public key cryptosystem based on "n" prime numbers. *Journal of Computer and Communications*, 6:78–90. DOI: 10.4236/jcc.2018.63006.

Java Cryptography (2025). Java cryptography architecture standard algorithm name documentation for jdk 8. Disponível em: `https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html`. Acesso em: 21 jan. 2025.

Jintcharadze, E. and Abashidze, M. (2023). Performance and comparative analysis of elliptic curve cryptography and rsa. In *2023 IEEE East-West Design & Test Symposium (EWDTS)*, pages 1–4. DOI: 10.1109/EWDTS59469.2023.10297088.

JMH (2025). Jmh is a java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in java and other languages targeting the jvm. Disponível em: `https://github.com/openjdk/jmh`. Acesso em: 25 jan. 2025.

Libgcrypt (2023). Libgcrypt documentation. Disponível em: `https://gnupg.org/software/libgcrypt/index.html`. Acesso em: 19 mar. 2024.

Maalavika, S., Thangavel, G., and Basheer, S. (2024). Performance evaluation of rsa type of algorithm with cuckoo optimized technique. In *2024 IEEE International Conference on Computing, Power and Communication Technologies (IC2PCT)*, volume 5, pages 1362–1367. DOI: 10.1109/IC2PCT60090.2024.10486350.

OpenSSL (2024). Openssl - cryptography and ssl/tls toolkit.

Disponível em: `https://www.openssl.org/`. Acesso em: 12 mar. 2024.

PyCryptodome (2024). Pycryptodome documentation. Disponível em: `https://www.pycryptodome.org/`. Acesso em: 19 mar. 2024.

PYPL (2024). Pypl - popularity of programming language. Disponível em: `https://pypl.github.io/PYPL.html`. Acesso em: 27 jun. 2024.

Pytest (2025). Pytest: helps you write better programs. Disponível em: `https://docs.pytest.org/en/stable/`. Acesso em: 25 jan. 2025.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126. DOI: 10.1145/359340.359342.

Rosa, A., Campiolo, R., and Batista, D. (2025). Avaliação de desempenho e segurança de diferentes implementações do sistema de criptografia rsa. In *Anais Estendidos do XXV Simpósio Brasileiro de Cibersegurança*, pages 180–191, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/sbseg_estendido.2025.11863.

Rust Crypto (2024). Cryptographic algorithms written in pure rust. Disponível em: `https://github.com/RustCrypto`. Acesso em: 19 mar. 2024.

Rust OpenSSL (2025). Openssl bindings for the rust programming language. Disponível em: `https://github.com/sfackler/rust-openssl`. Acesso em: 21 jan. 2025.

Stallings, W. (2008). *Cryptography and Network Security: Principles and Practice*. Prentice Hall, Upper Saddle River, NJ, 5 edition.

TIOBE (2024). Tiobe index - the software quality company. Disponível em: `https://www.tiobe.com/tiobe-index`. Acesso em: 27 jun. 2024.