Avaliação de desempenho e segurança de diferentes implementações do sistema de criptografia RSA

Ana Carla Quallio Rosa[1], **Rodrigo Campiolo**[1], **Daniel Macêdo Batista**[2]

[1] Departamento de Ciência da Computação
Universidade Tecnológica Federal do Paraná – Campo Mourão – PR – Brasil

[2]Departamento de Ciência da Computação
Universidade de São Paulo – São Paulo – SP – Brasil

anacarlarosa@alunos.utfpr.edu.br, rcampiolo@utfpr.edu.br, batista@ime.usp.br

Abstract. *RSA remains one of the most prominent cryptographic systems for protecting systems and data. This study evaluates implementations of the RSA algorithm using different programming languages and libraries. The experiments were conducted in C, C++, Java, JavaScript, Python, and Rust. Rust, C, and Python implementations exhibited the shortest execution times for encryption and decryption. The analysis of data structures revealed similar patterns across languages. Memory dumps exposed fragments of the private key in several implementations, with full recovery possible in JavaScript. The results provide valuable insights for performance and security decisions in RSA implementations.*

Resumo. *O RSA ainda se destaca como um dos principais sistemas de criptografia para prover a proteção de sistemas e dados. Este trabalho avalia implementações do algoritmo RSA, considerando diferentes linguagens de programação e bibliotecas. Os experimentos foram conduzidos em C, C++, Java, JavaScript, Python e Rust. Implementações em Rust, C e Python apresentaram os menores tempos de execução na cifração e decifração. A análise das estruturas revelou padrões semelhantes entre as linguagens. O despejo de memória mostrou fragmentos da chave privada em diversas implementações, sendo possível recuperá-la integralmente em JavaScript. Os resultados oferecem subsídios para decisões sobre desempenho e segurança em implementações do RSA.*

1. Introdução

Diante da expansão das telecomunicações e do aumento no fluxo de troca de informações nos dispositivos eletrônicos, os sistemas de criptografia tornam-se essenciais para manter, sobretudo, a autenticidade, a confidencialidade e a integridade dos dados. Isso porque a criptografia, ao transformar dados em formatos ininteligíveis para entidades não autorizadas, assegura que apenas remetentes e destinatários legítimos possam acessar o conteúdo original [Stallings and Vieira 2008].

Nesse contexto, o algoritmo RSA (Rivest–Shamir–Adleman) emergiu como um dos sistemas de chave pública mais adotados [Imam et al. 2021]. Desde a publicação por [Rivest et al. 1978], o RSA possui aplicações em distintos campos do conhecimento científico, como a geração de assinaturas digitais, autenticação de sítios e segurança de transações.

Entretanto, diante de um cenário de constantes atualizações na área de Segurança da Informação, o algoritmo RSA original torna-se mais propenso a ataques e, a depender da implementação, ineficiente. Sob tal ótica, [Imam et al. 2021] destacam que diversos trabalhos propõem alternativas e otimizações ao RSA, todavia, ainda verifica-se a necessidade de uma taxonomia consistente para alinhar requisitos como desempenho e segurança.

O objetivo principal deste estudo, derivado de um Trabalho de Conclusão de Curso e que complementa os resultados preliminares apresentados em [Rosa and Campiolo 2024], é conduzir uma avaliação das implementações do RSA. Como objetivos específicos, o trabalho envolveu analisar o tempo de execução do RSA para diferentes APIs em uma mesma linguagem e para linguagens de programação distintas; avaliar as modificações no algoritmo adotadas por cada implementação e, por fim, examinar as semelhanças e diferenças entre as estruturas de dados utilizadas para o armazenamento de informações confidenciais. Além da introdução, este artigo possui as seções de trabalhos relacionados, materiais e métodos, resultados e discussão, e considerações.

2. Trabalhos relacionados

Desde a publicação por [Rivest et al. 1978], diversas modificações foram propostas ao sistema de criptografia RSA. [Islam et al. 2018], por exemplo, generalizaram o RSA para o uso de n números primos, aumentando a complexidade e o custo computacional. Nesse contexto, visando melhorar a eficiência, [Alzaher et al. 2022] propuseram uma paralelização de uma abordagem do RSA que utiliza duas chaves públicas para cifrar a mensagem duas vezes e, posteriormente, duas chaves privadas para decifrá-la no lado do receptor. A paralelização foi realizada com o uso da biblioteca `OpenMP`, na linguagem C++. A implementação do RSA não utilizou bibliotecas criptográficas. Os resultados experimentais indicaram que a versão paralelizada apresentou desempenho superior em comparação à versão sequencial.

Seguindo uma linha de aprimoramento do RSA por meio de algoritmos heurísticos, [Maalavika et al. 2024] implementaram uma versão por meio do *Cuckoo Search*, algoritmo de busca utilizado nesse contexto para encontrar os números primos que geram a chave pública de maneira mais rápida. Os autores compararam essa abordagem com o RSA tradicional e com a criptografia de curva elíptica (ECC), considerando o tempo necessário para quebrar chaves de 256, 512 e 1024 *bits*. Os resultados indicaram que o algoritmo proposto apresentou maior resistência à quebra das chaves. Outros estudos compararam o RSA com algoritmos como ECC e Diffie-Hellman (DH). [Jintcharadze and Abashidze 2023] e [Dalal et al. 2024] concluíram, de modo geral, que o ECC obteve melhor desempenho e eficiência em termos de tempo de execução e tamanho de chave para níveis de segurança equivalentes.

No contexto de vulnerabilidades, [Afrose et al. 2019] propuseram um *benchmark* para detectar o uso indevido de APIs criptográficas em aplicações Java e Android, embora o foco não tenha sido em implementações diretas de algoritmos.

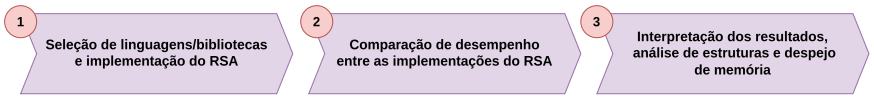
Os trabalhos relacionados mostram uma série de adaptações que visam melhorar a eficiência e segurança do RSA. No entanto, verifica-se, na literatura existente, a falta de uma abordagem que explora variações nas implementações do algoritmo e as implicações resultantes da adoção de diferentes bibliotecas no contexto de uma mesma linguagem de

programação, bem como em linguagens distintas.

3. Materiais e métodos

A Figura 1 fornece um panorama do método de pesquisa adotado.

Figura 1. Método de pesquisa



A seguir, são descritas cada etapa do método:

1. Seleção de linguagens/bibliotecas e implementação do RSA: Esta etapa compreende a escolha de linguagens de programação e de bibliotecas criptográficas para a implementação do RSA. Nesse contexto, escolheu-se cinco linguagens de programação, a saber: C, C++, Java, JavaScript, Python e Rust. Essa seleção se deu porque, segundo o Índice TIOBE [TIOBE 2024] e o Índice PYPL [PYPL 2024], tratam-se de linguagens mais utilizadas em cenários de diversos propósitos. Salienta-se que Rust foi escolhida porque trata-se de uma linguagem projetada para garantir a segurança de memória e concorrência. A seleção das bibliotecas, por sua vez, priorizou aquelas com maior uso e documentação mais abrangente;
2. Comparação de desempenho entre as implementações: Esta etapa compreende a comparação do processo de criptografia entre as bibliotecas selecionadas, abordando ainda as execuções individuais de geração de chaves, cifração e decifração em cada biblioteca por meio de *microbenchmarking*. Além disso, são consideradas distintas configurações de chaves e número de iterações;
3. Interpretação dos resultados, análise de estruturas e de despejo da memória: Esta etapa visa analisar e discutir os gráficos e tabelas obtidos. Nesse sentido, é apresentado um panorama entre as linguagens selecionadas. Além disso, são comparados padrões e estruturas de dados adotados para o armazenamento de informações confidenciais, com base na documentação de cada biblioteca. Por fim, tem-se a análise com a ferramenta `gcore`, que gera um despejo de memória de um programa em execução a partir de seu identificador de processo. O intuito dessa análise foi verificar a presença de fragmentos da chave privada que permitam sua recuperação. Para isso, foi testada a implementação do algoritmo RSA nas bibliotecas selecionadas, utilizando a chave gerada de 2048 *bits*.

3.1. Especificação de bibliotecas e ferramentas de *microbenchmarking*

Esta seção apresenta a relação entre bibliotecas criptográficas e ferramentas de *microbenchmarking* selecionadas.

Ressalta-se que algumas linguagens, como C++, Java, Python e Rust, oferecem ferramentas de *microbenchmarking*. Estas ferramentas geralmente incluem fases de *warm-up* para otimizar o código e estabilizar o ambiente antes da medição. Desse modo, os resultados apresentam um menor desvio padrão.

Entretanto, linguagens como C e JavaScript não dispõem de ferramentas específicas. Nesse caso, a medição de tempo foi realizada utilizando as bibliotecas padrão dessas

linguagens, enquanto as métricas estatísticas foram calculadas manualmente. Observou-se que, nesses casos, quanto menor o número de iterações, maior tende a ser o tempo médio e o desvio padrão. À medida que o número de iterações aumenta, esses resultados apresentam uma tendência de estabilização.

A Tabela 1 apresenta a relação entre as bibliotecas e ferramentas de *microbenchmarking* selecionadas por linguagem.

Tabela 1. Relação entre bibliotecas e ferramentas de benchmark por linguagem

Linguagem	Bibliotecas	Ferramenta
C	[OpenSSL 2024], [Libgcrypt 2023]	Biblioteca de tempo padrão
C++	[Botan 2025], [Crypto++ 2023]	[Google Benchmark 2025]
Java	[Bouncy Castle 2013], [Java Cryptography 2025]	[JMH 2025]
JavaScript	[Crypto 2025], [Forge 2024]	Biblioteca de tempo padrão
Python	[Cryptography 2024], [PyCryptodome 2024]	[Pytest 2025]
Rust	[Rust OpenSSL 2025], [Rust Crypto 2024]	[Cargo Bench 2025]

3.2. Contexto de execução dos experimentos

Os experimentos foram conduzidos em uma máquina com processador AMD Ryzen 5 (64 *bit*), 12 GB de memória RAM, SSD NVMe de 240 GB e sistema operacional Linux Mint 22.1 Xia - 64 *bits*. As execuções ocorreram na interface gráfica Xfce, considerando os processos do sistema e o terminal em operação durante os testes. Para o desenvolvimento e execução dos códigos, foram empregados os seguintes compiladores e interpretadores: `GCC` (GNU *Compiler Collection*) versão 11.4.0 para C/C++, `OpenJDK` versão 11.0.27 para Java, `Node.js` versão 20.19.0 para JavaScript, `Python` versão 3.10.1 para Python e `rustc` versão 1.86.0 para Rust.

Para garantir consistência nos resultados, foram adotados alguns padrões. Em todas as execuções, utilizou-se um arquivo de mensagem no formato *.txt* de tamanho de 90 *bytes*, correspondente ao limite máximo permitido para a cifração com uma chave RSA de 2048 *bits*, considerando o preenchimento aplicado pela biblioteca `Forge`, do JavaScript. As chaves de 2048, 3072 e 4096 *bits* foram geradas previamente com `OpenSSL` e carregadas nos processos de cifração e decifração em todas as linguagens. Os experimentos foram conduzidos com a utilização de um núcleo de processamento.

O limite máximo, em *bytes*, da mensagem que pode ser cifrada com o RSA depende tanto do tamanho da chave quanto da implementação específica das bibliotecas e das estruturas de dados utilizadas. Por exemplo, para uma chave de 2048 *bits*, o tamanho máximo de mensagem permitido é de 256 *bytes*. Já para uma chave de 4096 *bits*, o limite é de 512 *bytes*. Esse tamanho máximo considera o algoritmo de preenchimento aplicado, que ocupa parte do espaço disponível para a mensagem.

A Tabela 2 apresenta a relação entre o tamanho máximo permitido para a mensagem, em *bytes*, e o tamanho da chave, em *bits*, para cada biblioteca analisada, além dos algoritmos de preenchimento utilizados nas diferentes implementações.

A maioria das bibliotecas selecionadas aceita chaves no formato PEM (*Privacy Enhanced Mail*), com algumas exceções. A biblioteca `Crypto++`, da linguagem C++, apenas carrega chaves no formato DER (*Distinguished Encoding Rules*), o que exigiu

Tabela 2. Relação entre tamanho máximo de mensagem (*bytes*) permitido por biblioteca, chave e algoritmo de preenchimento

Biblioteca	2048 *bits*	3072 *bits*	4096 *bits*	Algoritmo de preenchimento
OpenSSL (C)	245	373	501	PKCS#1 v1.5
Libgcrypt (C)	214	318	470	Sem por padrão (OAEP aplicado)
Botan (C++)	190	318	446	OAEP
Crypto++ (C++)	214	342	470	OAEP
Bouncy Castle (Java)	192	288	384	Sem por padrão[a]
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

[a] Optou-se por não aplicar o OAEP devido à ausência de um exemplo direto na documentação para a utilização do construtor do método de preenchimento.

a conversão das chaves geradas inicialmente utilizando o `OpenSSL`. Já a biblioteca `Libgcrypt`, da linguagem C, apresenta a limitação de suportar exclusivamente chaves no formato *S-Expression*[1]. Para contornar essa limitação, foi utilizado o pacote `monkeysphere`, que expande as funcionalidades do OpenPGP (`Libgcrypt`). Esse pacote permitiu converter chaves inicialmente geradas no formato PEM para o formato GPG (GNU *Privacy Guard*). Posteriormente, um *script* em Python foi executado para extrair os dados das chaves pública e privada e convertê-los para o formato *S-Expression*, possibilitando o carregamento e o uso das chaves.

Por fim, os processos de cifração e decifração foram executados com conjuntos de 10, 100, 1000 e 10000 iterações. De maneira geral, o conjunto de 10 iterações apresentou um pico na média de repetições e no desvio padrão, considerando tanto implementações que não utilizaram uma biblioteca específica de *microbenchmarking* quanto as que utilizaram ferramentas. Com isso, os resultados finais são apresentados como a média de 5 execuções para cada conjunto, excluindo o conjunto de 10 iterações.

3.3. Análise de despejo de memória

Para a realização da análise de despejo de memória, o primeiro passo consistiu na obtenção do identificador de cada programa em execução. Em seguida, a ferramenta `gcore` foi executada com privilégios administrativos (`sudo`) para a geração do despejo de memória.

Em relação ao armazenamento da chave privada na memória, as bibliotecas aplicam manipulações, como a conversão para *bytes* e a inversão da sequência. Durante a análise, buscou-se identificar tanto trechos da chave no formato PEM — com exceção das bibliotecas `Libgcrypt`, que utiliza *S-Expression*, e `Crypto++`, que aceita somente o formato DER — quanto sequências convertidas em *bytes*, incluindo a possibilidade de inversão da sequência.

No que tange à identificação de trechos da chave privada no despejo de memória, adotaram-se duas abordagens. Na primeira, utilizou-se o comando `strings`

[1]Estruturas de dados utilizadas na implementação de algoritmos de criptografia assimétrica. Essas expressões armazenam informações como chaves, mensagens a serem cifradas, dados criptografados e assinaturas digitais.

`arquivo_dump | grep trecho_chave` para localizar diretamente fragmentos da chave. Na segunda, foi desenvolvido um *script* em Python que aplica as manipulações supracitadas e busca, no despejo de memória gerado durante a execução do código de cada biblioteca, padrões correspondentes à chave privada.

Para a busca de fragmentos em memória, a chave convertida foi subdividida em sequências de 16 *bytes*, considerando também a versão invertida. Os scripts desenvolvidos e os códigos que demonstram a utilização das bibliotecas selecionadas estão disponíveis em um repositório no GitHub[2].

4. Resultados e discussão

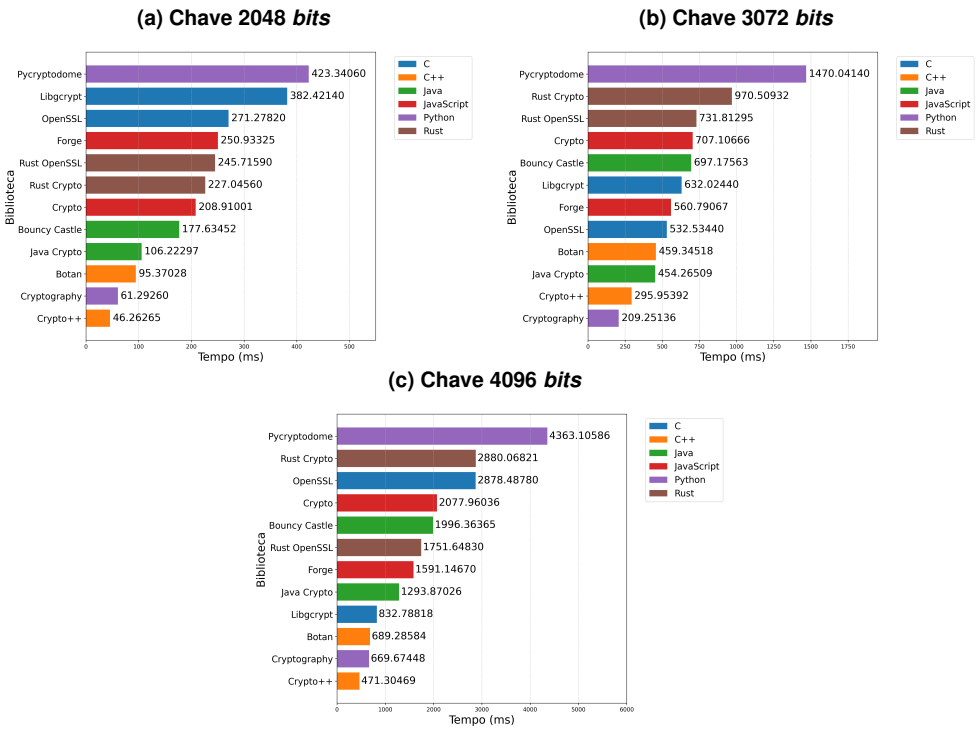
A análise inicial verificou o tempo médio de execução dos processos de criptografia para cada biblioteca selecionada. Em seguida, foram investigados os padrões utilizados na implementação do RSA e mecanismos de segurança adotados. Por fim, foi realizada uma análise com `gcore` para verificar fragmentos da chave privada na memória.

4.1. Tempo médio dos processos de criptografia

A análise de tempo médio foi dividida em três etapas principais: a geração de chaves, a cifração e a decifração.

Para facilitar a interpretação de todos os tempos médios obtidos, organizamos os dados experimentais em classificações de acordo com o processo de criptografia e o tamanho de chave utilizado. A Figura 2 apresenta a classificação dos tempos referentes à geração de chaves.

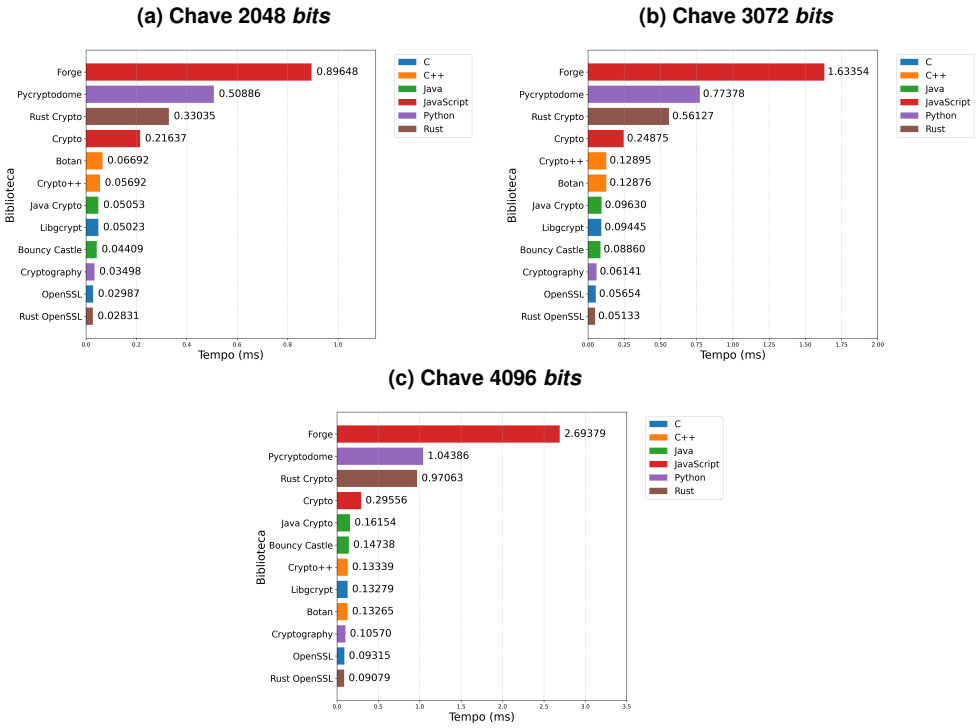
Figura 2. Classificação do processo de geração



[2]Disponível em: `https://github.com/anacarlaquallio/tcc2`. Acesso em: 22 de maio de 2025.

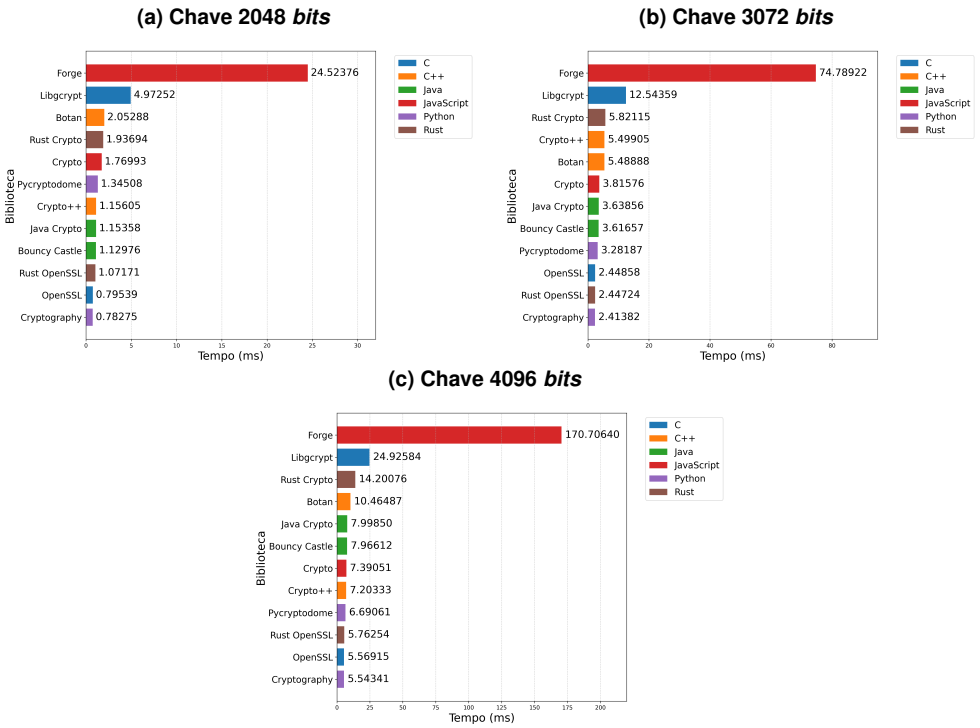
Em seguida, a Figura 3 exibe a classificação do processo de cifração.

Figura 3. Classificação do processo de cifração



Por fim, a Figura 4 ilustra os resultados obtidos no processo de decifração.

Figura 4. Classificação do processo de decifração



Com base nas Figuras 2, 3 e 4, as bibliotecas `Crypto++` (C++) e `Cryptography` (Python) apresentaram os menores tempos de geração de chaves. Já nos processos de cifração e decifração, os melhores desempenhos foram obtidos com as bibliotecas `Rust OpenSSL` (Rust), `Cryptography` (Python) e `OpenSSL` (C). Em contrapartida, a biblioteca `Forge`, do JavaScript, apresentou o pior desempenho em todos os experimentos realizados nesse contexto.

Ressalta-se que o processo de geração de chaves possui desvio padrão maior em relação à cifração e à decifração porque o tempo depende dos expoentes selecionados e operações matemáticas envolvidas, que podem diferir a cada execução. A planilha com os resultados completos está disponível no repositório mencionado.

4.2. Análise da implementação das bibliotecas selecionadas

A análise das implementações do RSA revelou que a biblioteca `OpenSSL` é utilizada como *wrapper* em Rust (`Rust OpenSSL`), Python (`Cryptography`) e JavaScript (`Crypto`). Essa integração contribui para o desempenho superior observado em Rust, C e Python nas operações criptográficas.

Em termos de padrões e estruturas, as bibliotecas geralmente seguem os protocolos RFC 8017 e 5208. O padrão X.509 é comum para chaves públicas (exceto `PyCryptodome` com PKCS #1), e o PKCS #8 para chaves privadas. A codificação ASN.1 é prevalente, com exceção da `Libgcrypt`, que utiliza *S-expression*.

As técnicas de segurança[3] incluem:

- Ofuscamento: `Libgcrypt`, `Botan` e `Rust Crypto` empregam técnicas de ofuscamento, como a introdução de fatores aleatórios, para mitigar ataques de temporização.
- Expoente Público: `Crypto++` utiliza um expoente público 17, diferente do valor padrão 65537.

Já as vulnerabilidades incluem:

- Vulnerabilidades em `Cryptography` (Python): Questões de limpeza de memória (devido a estruturas imutáveis em Python) e o uso do padrão PKCS #1 v1.5 em contextos *online*, que impede a verificação de decifração bem-sucedida e pode viabilizar ataques;
- Vulnerabilidade em JavaScript (`Forge`): Problemas na geração de números aleatórios podem levar à previsibilidade, impactando a segurança das chaves. A biblioteca implementa uma solução baseada no algoritmo Fortuna[4] [Dodis et al. 2014], mas a documentação destaca limitações que precisam ser aprimoradas para garantir conformidade com o protocolo TLS (*Transport Layer Security*);
- Vulnerabilidade em `Rust Crypto`: Suscetibilidade ao ataque Marvin[5], que possibilita a recuperação da chave privada em cenários de rede (correção em andamento, ver *issue* #19).

[3]Bibliotecas que implementam técnicas de segurança exigem maior custo computacional. Em linguagens interpretadas, riscos como gerenciamento de memória ou geração de números aleatórios tendem a ser mitigados por *wrappers*.

[4]Algoritmo de geração de números pseudoaleatórios para operações criptográficas. Documentação disponível em: `https://www.schneier.com/academic/fortuna/`. Acesso em 09 fev. 2025.

[5]O Ataque Marvin explora variações no tempo de execução para obter detalhes sobre a chave privada e assinaturas. O principal alvo deste ataque são os servidores TLS. Ao explorar a diferença no tempo de

A Tabela 3 sintetiza as estruturas de dados usadas nas operações de cifração e decifração.

Tabela 3. Relação dos nomes das estruturas de dados implementadas por bibliotecas

Biblioteca	Cifração	Decifração
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`
Rust Crypto	`RsaPublicKey`	`RsaPrivateKey`

Nota-se que algumas implementações, como `Libgcrypt` e `Crypto`, usam a mesma estrutura para ambas as operações, enquanto outras, como `Crypto++` e `Bouncy Castle`, utilizam estruturas distintas.

4.3. Análise de despejo de memória

Como etapa final da análise, utilizou-se a ferramenta `gcore` para gerar despejos de memória dos processos em execução.

A `OpenSSL` apresentou fragmentos da chave privada em memória, no formato *bytes*, distribuídos em blocos distintos. Na `Libgcrypt`, devido ao uso de *S-Expressions*, foi necessário extrair manualmente os componentes da chave (n, e, d), convertê-los de hexadecimal para *bytes* e buscar tanto a forma original quanto invertida. O expoente público e foi encontrado em ambas as formas.

Nas bibliotecas `Botan` e `Crypto++`, identificaram-se fragmentos em *bytes* e trechos em formato PEM. A `Bouncy Castle` revelou poucos vestígios, inviabilizando a recuperação. Já a `Java Crypto` exibiu fragmentos semelhantes aos da `OpenSSL`.

Nas bibliotecas JavaScript `Forge` e `Crypto`, a chave foi localizada integralmente em memória, nos formatos PEM e *bytes*. A Listagem 1 apresenta a busca de um trecho da chave PEM no despejo da `Forge`.

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

Listagem 1. Busca por um trecho da chave privada, no formato PEM, no despejo de memória gerado pela biblioteca `Forge`

Na `Cryptography` e na `Rust OpenSSL`, não foram detectados vestígios da chave. Já `PyCryptodome` e `Rust Crypto` revelaram fragmentos em *bytes*.

processamento da operação de decifração do RSA, o invasor pode obter informações sensíveis e forjar assinaturas. Descrição do ataque disponível em: `https://people.redhat.com/~hkario/marvin/`. Acesso em 09 fev. 2025.

Embora a `OpenSSL` apresente vestígios, implementações derivadas como `Cryptography` e `Rust OpenSSL` parecem mitigar essa vulnerabilidade. A recuperação integral nas bibliotecas JavaScript sugere uma possível vulnerabilidade da linguagem, ainda que não documentada. Nesse caso, um atacante com acesso ao despejo de memória pode decifrar comunicações e forjar assinaturas digitais, comprometendo diretamente a confidencialidade e a autenticidade dos dados.

Tentou-se reconstruir a chave a partir dos *bytes* identificados, convertendo-os para PEM. Nenhuma sequência gerou um arquivo válido: o cabeçalho ASN.1 (`30 82`) não foi encontrado no início dos blocos, surgindo apenas em posições intermediárias. Testes posteriores confirmaram que os arquivos reconstruídos eram distintos.

Com a redução do tamanho dos blocos analisados, observou-se que bibliotecas como `OpenSSL` e `Bouncy Castle` armazenam trechos da chave em *offsets* consecutivos, sugerindo blocos contíguos. Ainda assim, partes dispersas indicam fragmentação, corrupção ou múltiplos formatos de armazenamento como estratégia para dificultar a recuperação. O *script* utilizado encontra-se no repositório das implementações.

5. Considerações

Este trabalho teve como objetivo avaliar diferentes abordagens de implementação do algoritmo RSA. A análise dos tempos médios de execução revelou que, na geração de chaves, as linguagens C++ e Python obtiveram o menor tempo médio, considerando os tamanhos de 2048, 3072 e 4096 *bits*. Nos processos de cifração e decifração, as linguagens Rust, Python e C apresentaram o melhor desempenho, principalmente devido ao uso do `OpenSSL` como base de suas implementações. Além disso, a implementação do RSA nas linguagens selecionadas demonstrou similaridades no uso de padrões e estruturas de dados para o armazenamento das chaves. Cada biblioteca introduz mecanismos de segurança específicos, influenciando tanto o tempo médio de execução quanto a complexidade da implementação.

A análise de memória revelou fragmentos da chave em diferentes bibliotecas. No caso do JavaScript, foi possível recuperar a chave privada integralmente nas duas implementações desenvolvidas, configurando uma vulnerabilidade que pode comprometer a segurança do sistema. Em outras bibliotecas, foram realizadas tentativas de recuperação da chave, mas o resultado final não gerou uma chave válida.

Os resultados obtidos fornecem uma base para a escolha de implementações do RSA em diversos contextos, levando em consideração tanto o desempenho quanto a segurança. Durante a pesquisa, foram enfrentados desafios, como a falta de documentação e informações detalhadas sobre a implementação do RSA. Esses aspectos devem ser aprimorados pela comunidade de desenvolvedores de cada biblioteca selecionada, a fim de garantir que pesquisadores e profissionais possam realizar investigações mais precisas e desenvolver implementações seguras.

Como trabalhos futuros, sugere-se a implementação do RSA em diferentes bibliotecas, considerando cenários com restrições de memória e *hardware*, e tamanhos de chaves maiores, considerando as limitações de cada biblioteca. Por fim, a análise de segurança pode ser aprofundada para investigar a presença de variáveis e estruturas de dados sensíveis na memória, assim como tentativas de recuperação das chaves em fragmentos aleatórios, contribuindo para a mitigação de potenciais vulnerabilidades.

Agradecimentos

Pesquisa parcialmente financiada pelo CNPq (proc. 405940/2022-0) e Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 88887.954253/2024-00.

Referências

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.

Alzaher, R., Hantom, W., Aldweesh, A., and Allah, N. M. (2022). Parallelizing multikeys rsa encryption algorithm using openmp. In *2022 14th International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 778–782.

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.

Dodis, Y., Shamir, A., Stephens-Davidowitz, N., and Wichs, D. (2014). How to eat your entropy and have it too – optimal recovery strategies for compromised RNGs. Cryptology ePrint Archive, Paper 2014/167.

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.

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.

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.

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.

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.

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.

Rosa, A. and Campiolo, R. (2024). Avaliação de diferentes implementações do sistema de criptografia rsa. In *Anais Estendidos do XXIV SBSeg*, pages 347–352. SBC.

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. and Vieira, D. (2008). *Criptografia e segurança de redes: princípios e práticas*. Pearson Prentice Hall.

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