Análise comparativa de implementações padrão de algoritmos de *hashing* disponibilizadas pelo *framework* Spring Security

NICOLAS SATIL MENDES
LUCIANO GONÇALVES DE CARVALHO

Resumo

Senhas são comumente alvo de atacantes maliciosos, e, portanto, quando armazenadas, passam por técnicas e processos para aumentar o grau de segurança e diminuir o risco de um ataque bem sucedido. O *framework* Spring Security fornece implementações dos algoritmos de *hashing* BCrypt, SCrypt, PBKDF2 e Argon2. Este artigo visa comparar métricas de performance das implementações padrão desses algoritmos de *hashing* e auxiliar na escolha de um algoritmo de *hashing* dado um contexto de utilização de recursos computacionais. Através da ferramenta de benchmarking Java Microbenchmark Harness (JMH), foi examinado o consumo de memória e o tempo médio de execução do *hashing* das senhas mais comumente usadas, e, pelo software Hashcat, a resistência a ataques de dicionário. Observa-se que o algoritmo BCrypt tem o menor consumo de memória e de tempo de execução, enquanto o SCrypt tem o maior consumo de memória e a maior resistência a ataques de dicionário, e o algoritmo Argon2 tem consumo de memória maior do que BCrypt, porém, com resistência a ataques de dicionário similar. O algoritmo PBKDF2 apresentou o maior tempo de execução e a menor resistência a ataques de dicionário.

Palavras-chave*: hashing*; senhas; Spring Security; *benchmarking*; Argon2.

Comparative analysis of hashing algorithm implementations available on the Spring Security framework

Abstract

Passwords are commonly the main target of malicious attackers, and therefore, when stored, techniques and processes are applied with the goal of increasing security and decreasing the risk of a successful attack. The Spring Security framework provides implementations of the hashing algorithms BCrypt, SCrypt, PBKDF2, and Argon2. This paper aims to compare the standard implementations of these hashing algorithms, and aid in choosing a hashing algorithm in a computational resource utilization context. By using the Java Microbenchmark Harness (JMH), memory consumption and average execution time of the hashing of common passwords was analyzed, along with the usage of Hashcat to evaluate dictionary attack resistance. Results showed BCrypt has the lowest memory consumption and lowest execution time, while SCrypt has the greatest memory consumption and greatest dictionary attack resistance, and Argon2 consumed more memory than BCrypt, though with similar dictionary attack resistance. PBKDF2 showed the slowest execution time and the weakest dictionary attack resistance.

Keywords: *hashing; passwords; Spring Security; benchmarking; Argon2.*

---

1 INTRODUÇÃO

Um dos dados sensíveis mais visados por atacantes maliciosos são as senhas. Elas consistem em um conjunto de caracteres idealmente mantidos em sigilo pelo usuário, e são utilizadas para realizar o processo de autenticação, ou seja, a verificação da identidade do usuário na aplicação. Senhas passam por técnicas e processos para aumentar o grau de segurança e diminuir a chance de que ataques sejam bem-sucedidos.

Uma das técnicas de proteção de dados mais utilizadas é o *hashing*, técnica na qual um algoritmo recebe um texto como entrada, de tamanho variável, como uma senha, e tem como saída um texto de tamanho fixo, chamado de *hash*. Esse *hash* é então guardado no banco de dados da aplicação, e o processo de autenticação em aplicações web é tradicionalmente feito de modo que, em toda tentativa de autenticação de um usuário, a senha recebida passa pelo algoritmo de *hash* utilizado e o *hash* resultante é então comparado ao *hash* armazenado no banco de dados. Caso os dois *hashes* sejam iguais, a autenticação é validada. Além disso, segundo Coelho et al. (2023), qualquer alteração na entrada original gera um *hash* totalmente diferente, o que permite checar alterações no dado e impedindo (geralmente) que senhas parecidas gerem *hashes* iguais. Além da segurança, também é necessário prezar por velocidade, como demonstrado por um estudo da Google em que 53% de usuários *mobile* saem de um site que leva mais de 3 segundos para carregar (Google, 2016). Para o armazenamento de senhas, o *Open Worldwide Application* Security Project (OWASP) recomenda que elas passem por *hashing* ao invés de serem criptografadas e que a duração do cálculo do *hash* seja de menos de 1 segundo (OWASP, 2025).

É importante então que o algoritmo de *hashing* mais adequado aos recursos computacionais disponíveis seja o favorecido. Para facilitar e acelerar o desenvolvimento de sistemas, alguns frameworks de desenvolvimento web disponibilizam implementações com parâmetros padronizados de algoritmos de *hashing*, como o Spring Security, módulo de segurança do ecossistema Spring. O presente trabalho teve como objetivo comparar quatro implementações padrão de algoritmos de *hash* de senhas do Spring Security: BCryptPasswordEncoder, Argon2PasswordEncoder, Pbkdf2PasswordEncoder e SCryptPasswordEncoder. Tal comparação é feita sob a ótica de parâmetros de performance, sendo eles o tempo médio de execução e o consumo de memória, ambos medidos com o uso do software Java Microbenchmark Harness (JMH), para realizar o *hashing* de senhas contidas em duas listas de respectivamente 200 e 10000 senhas, bem como na ótica de parâmetros de segurança, sendo utilizado o software Hashcat com o objetivo de descobrir senhas a partir de *hashes* dos algoritmos citados, utilizando um ataque de dicionário. Busca-se então entender qual é o grau de uso de recursos computacionais das implementações e o grau de segurança delas a ataques de dicionário *offline*.

2 DESENVOLVIMENTO

2.1 Fundamentos e contextualização de *hashing* de senhas

Funções de *hash* são algoritmos que geram *hashes* de variados tamanhos e graus de segurança. Funções de *hash* possuem algumas propriedades. Ertaul, Kaur e Gudise (2016) disseram que é possível computar *hashes* de qualquer mensagem rapidamente; inviabilizam a recuperação do texto original de seu *hash*; qualquer alteração no texto original provoca alteração no *hash* resultante; quaisquer duas mensagens parecidas quase nunca geram o mesmo *hash*.

O *hashing* é fundamental para proteger a segurança de senhas no caso de um vazamento de dados, pois um adversário pode simplesmente acessar o banco de dados e ter acesso à informação secreta em instantes caso a senha tenha sido armazenada em texto puro ao invés de seu *hash*, comprometendo a confidencialidade do sistema em segundos (Prima, 2025).

Efetuar o *hashing* de uma informação sensível, como uma senha, é mais efetivo do que criptografá-la utilizando a criptografia tradicional pois funções de *hash* dificultam significativamente a descoberta do texto original através de seu *hash* (Sriramya; Karthika, 2015), entretanto ataques de *Rainbow Tables* e ataques de dicionário ainda podem passar pela segurança de uma função de *hash* (Manankova *et al*., 2023) especialmente se o algoritmo não

utilizar um *salt*, que é um valor gerado aleatoriamente e que uma função de *hashing* toma como um de seus argumentos e é concatenado à mensagem original. Como um *salt* é gerado a partir de bits aleatórios, um programa não pode prever qual valor será desafiado (Rosulek, 2021).

O ataque de dicionário *online* é comumente utilizado por invasores para quebrar senhas, consistindo em tentativas repetitivas de autenticação em algum serviço, geralmente um serviço web, utilizando uma lista de senhas de tamanho considerável. Tal ataque pode ser feito com o uso de 1 ou mais endereços IP, via utilização de numerosos *bots.* Embora esse seja o caso, existem técnicas de segurança que dificultam a viabilidade desse ataque, o que leva atacantes a buscarem maneiras alternativas de comprometerem a segurança dos usuários. Uma dessas técnicas é o ataque de dicionário *offline*, que, diferente do citado anteriormente, trata-se do uso de um banco de dados de *hashes* de senhas vazadas, que são submetidas a comparação com *hashes* de senhas contidas no dicionário. Quando correspondências são descobertas (o *hash* do banco vazado é igual ao *hash* calculado) os malfeitores podem então adentrar o sistema autenticados como os usuários que tiveram suas informações vazadas (Queiroz, 2022).

2.1.1 Trabalhos relacionados

Em 2015, foi conduzida por estudiosos da criptografia a *Password Hashing Competition*, com mais de 20 algoritmos submetidos. Hatzivasilis, Papaefstathiou e Manifavas (2015) realizaram um *benchmark*, tendo sido medido o tempo de execução, tamanho de código-fonte, e consumo de memória de algoritmos submetidos na competição.

Ertaul, Kaur e Gudise (2016) implementaram uma aplicação Android em que a senha de entrada do usuário passa por *hashing* e o *hash* resultante é usado como chave criptográfica para o algoritmo AES que criptografa os números de telefone salvos na lista de contato do celular. Uma análise entre os algoritmos PBKDF2, SCrypt e BCrypt foi feita observando o tempo de execução para gerar o *hash* da senha. Tal análise foi empregada variando um parâmetro dos algoritmos por vez e mantendo os outros iguais. Primeiro, variando o tamanho de chave (16, 32, e 64 bits), depois a quantidade de iterações (10, 100, e 500 iterações), e por fim o tamanho do *salt* (32, 64, e 128 bytes).

Pittalia (2019) forneceu uma análise dos algoritmos MD5, SHA-1, SHA-2, SHA-3 e Whirlpool tanto de seu funcionamento como de seus parâmetros, como tamanho de *hash* e quantidade de iterações no processo de *hashing*.

Fedorchenko *et al*. (2024) compararam implementações de algoritmos de *hash* tanto nativos como em bibliotecas da plataforma .NET, especificamente BCrypt, SCrypt, PBKDF2, Argon2, MD5 e SHA256. Foram separadas listas de senhas de tamanhos variados e calculado o tempo de *hashing* em nanosegundos. Além disso, foi utilizado o software Hashcat para empregar um ataque de dicionário a cada lista e analisar a quantidade de senhas descobertas.

2.2 Materiais e métodos

Para realizar a análise, foi empregado o desenvolvimento de uma aplicação em console Java com os frameworks Spring Boot e Spring Security. Foram escolhidas 2 listas de senhas para calcular a velocidade de *hashing*, medindo a quantidade média de tempo decorrido para realizar os cálculos. As 2 listas são:

a. lista de 200 senhas retiradas das 200 senhas mais comuns globais segundo levantamento do site da NordPass em 2024 (NordPass, 2024);

b. lista de 10.000 senhas retiradas do site da WeakPass denominada "*10_million_password_list_top_10000*" (WeakPass, 2025).

O *hashing* foi realizado pelas quatro implementações de algoritmos de *hash*, seguindo metodologia semelhante à descrita por Fedorchenko *et al*. (2024), com as medições incluindo

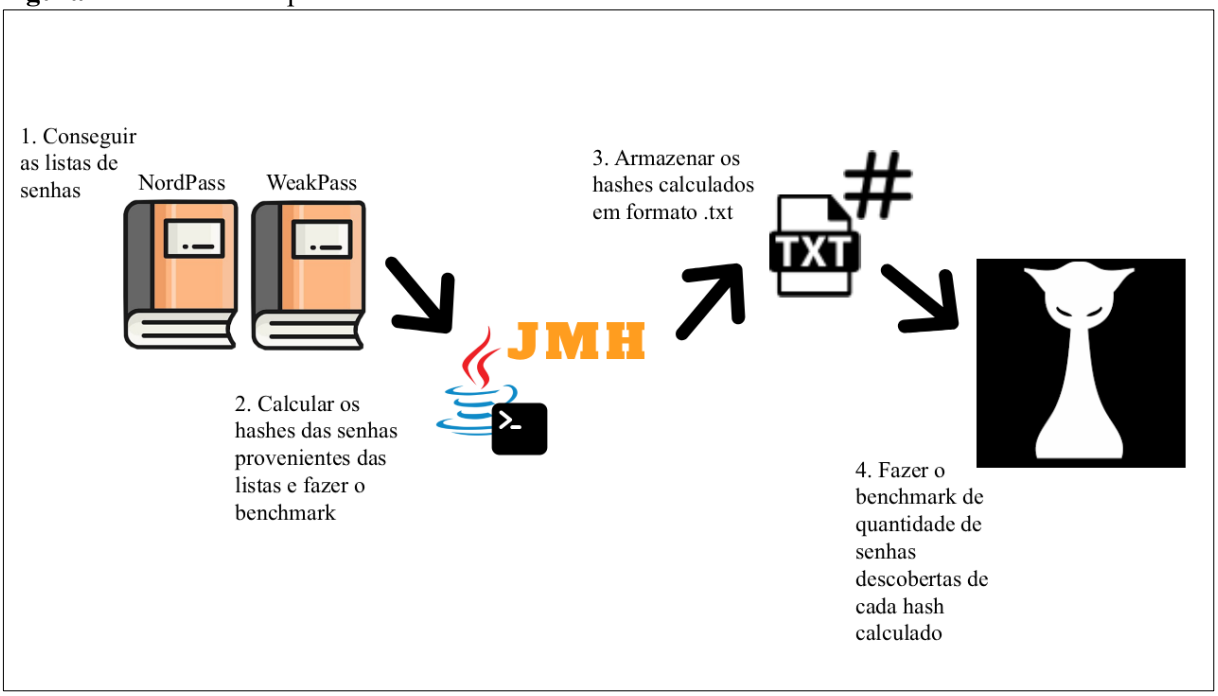
o tempo de execução médio em minutos e a alocação de memória em megabytes por operação para cada combinação de lista e algoritmo.

Para tal, foi empregada a biblioteca JMH, que utiliza iterações de aquecimento dos testes para simular de forma mais aproximada como o algoritmo se comportaria em um ambiente de produção. O valor escolhido para iterações de aquecimento e para a quantidade de iterações padrão foram 2 e 3, respectivamente.

Após a conclusão do *hashing*, os *hashes* foram coletados e submetidos a ataque de dicionário com o software Hashcat, utilizando uma lista de cerca de 14 milhões de senhas denominada "*rockyou.txt*" como dicionário, retirada também da WeakPass. Foi registrada a quantidade de senhas descobertas. O limite de tempo determinado para o ataque foi de 3 horas.

A Figura 1 mostra, de forma resumida, o procedimento executado para realização dos testes.

Figura 1 – Resumo dos procedimentos



Fonte: elaborada pelos autores, 2026.

2.2.1 Implementação do código do experimento

Existem 4 funcionalidades principais que o código necessita para realizar os testes determinados anteriormente: Calcular *hashes*, exibir os resultados, manipular arquivos de formato .txt, e realizar o *benchmark*. Tais funcionalidades foram codificadas em específicos métodos, distribuídos em 4 classes: GeradorSenhas, GerenciadorHashers, Performance e ProjetohashApplication.

A classe GerenciadorHashers possui a declaração dos 4 objetos de *encoding* de texto puro para *hash*, seus respectivos *getters* e *setters*, e 2 métodos, hashPasswords e insertPasswords. hashPasswords transforma o texto puro passado por um parâmetro utilizando um *encoder* também em um parâmetro, e insertPasswords insere senhas em texto puro vindas de um vetor, coloca seus *hashes* em uma *ArrayList*, e retorna essa *ArrayList*. Vale a pena declarar que a versão padrão dos *encoders* utilizados é a versão defaultsForSpringSecurity_v5_8. Existem outras versões padrão desses *encoders*, porém essa é a versão utilizada na versão do Spring Security investigada por este trabalho.

Quanto aos parâmetros utilizados pelos *encoders* na implementação padrão disponibilizada pelo Spring Security, temos: A versão sem parâmetros do *encoder* do BCrypt utiliza um *cost factor*, um parâmetro relacionado a resistência do *hash*, de 10, e a versão 2a. O Argon2 utiliza um *salt* de tamanho de 16 bytes, um *hash* de 32 bytes, 1 de paralelismo, um custo de memória de 16384 KiB, 2 iterações, e a versão do Argon2 usada é a argon2id. O SCrypt tem um custo de unidade central de processamento (CPU) de 65536, custo de memória de 8, paralelismo de 1, tamanho do *hash* de 32 bytes, e tamanho do *salt* de 16 bytes. O custo de CPU, custo de memória, e paralelismo são multiplicados juntos de 128 para resultar no custo estimado de memória em bytes para calcular um *hash*. Quanto ao último *encoder* analisado, o PBKDF2, ele utiliza um tamanho de *salt* de 16 bytes, e um número de iterações correspondente a 310000.

A Figura 2 exibe a declaração dos *encoders* especificados, e a Figura 3 mostra o código referente aos métodos hashPasswords e insertPasswords.

Figura 2 – Declarações dos *encoders* da classe GerenciadorHashers.

```
public class GerenciadorHashers {

    public Argon2PasswordEncoder argon2Hasher_DEFAULT = Argon2PasswordEncoder.defaultsForSpringSecurity_v5_8();

    public SCryptPasswordEncoder scryptHasher_DEFAULT = SCryptPasswordEncoder.defaultsForSpringSecurity_v5_8();

    public Pbkdf2PasswordEncoder pbkdf2Hasher_DEFAULT = Pbkdf2PasswordEncoder.defaultsForSpringSecurity_v5_8();

    public BCryptPasswordEncoder bcryptHasher_DEFAULT = new BCryptPasswordEncoder();
```

Fonte: elaborada pelos autores, 2026.

Figura 3 – Métodos hashPasswords e insertPasswords da classe GerenciadorHashers.

```
public void hashPasswords(PasswordEncoder passwordEncoder, String[] passwords) {
    for (String password : passwords) {
        passwordEncoder.encode(password);
    }
}

public ArrayList<String> insertPasswords(PasswordEncoder passwordEncoder, String[] passwords) {
    ArrayList<String> hashedPasswords = new ArrayList<>();
    for (String password : passwords) {
        hashedPasswords.add(passwordEncoder.encode(password));
    }
    return hashedPasswords;
}
```

Fonte: elaborada pelos autores, 2026.

A classe GeradorSenhas instancia um objeto de GerenciadorHashers e recebe seus *encoders*. Há uma declaração das senhas da lista da NordPass para um vetor de *strings* e mais três métodos: get10kList, generateTxts e generate2SetTxts. O método get10kList cria um vetor de *strings*, lê um arquivo .txt correspondente a lista de senhas da WeakPass, armazena cada senha lida neste vetor, e retorna o vetor. Para gerar um arquivo .txt, se usa o método generateTxts, que recebe uma *ArrayList* e o nome do .txt que se deseja criar, esse arquivo é criado dentro de uma pasta específica na pasta do Hashcat. Por fim, o método generate2SetTxts gera 2 .txts de *hashes* das senhas provenientes das listas de senhas da NordPass e WeakPass. A Figura 4 exibe a implementação do método get10kList, a Figura 5 mostra o método generateTxts, e a Figura 6 exibe o código do método generate2SetTxts.

Figura 4 – Método get10kList da classe GeradorSenhas.

```
public String[] get10kList() {
    String[] list10k = new String[10000];

    try {
        File list10kFile = new File(pathname: "C:\\hashcat-7.1.2\\experimento_benchmark_seguranca\\10_million_password_list_top_10000.txt");
        Scanner reader = new Scanner(list10kFile);
        int i = 0;
        while(reader.hasNextLine()) {
            list10k[i] = reader.nextLine();
            i++;
        }
        reader.close();
    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    }

    return list10k;
}
```

Fonte: elaborada pelos autores, 2026.

Figura 5 – Método generateTxts da classe GeradorSenhas.

```
public void generateTxts(ArrayList<String> hashList, String txtName) {
    try (FileWriter writer = new FileWriter("C:\\hashcat-7.1.2\\experimento_benchmark_seguranca\\" + txtName + ".txt")) {
        System.out.println(hashList.size());
        for (int i = 0; i < hashList.size(); i++) {
            if(i == hashList.size()-1) {
                writer.write(hashList.get(i));
            }  else {
                writer.write(hashList.get(i) + "\n");
            }
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Fonte: elaborada pelos autores, 2026.

Figura 6 – Método generate2SetTxts da classe GeradorSenhas.

```
public void generate2SetTxts() {
    // NORDPASS LIST

    ArrayList<String> bcryptHashes_NORDPASS = new ArrayList<>(gerenciadorHashers.insertPasswords(bCryptPasswordEncoder, senhasNordpass));
    ArrayList<String> scryptHashes_NORDPASS = new ArrayList<>(gerenciadorHashers.insertPasswords(sCryptPasswordEncoder, senhasNordpass));
    ArrayList<String> pbkdf2Hashes_NORDPASS = new ArrayList<>(gerenciadorHashers.insertPasswords(pbkdf2PasswordEncoder, senhasNordpass));
    ArrayList<String> argon2Hashes_NORDPASS = new ArrayList<>(gerenciadorHashers.insertPasswords(argon2PasswordEncoder, senhasNordpass));

    generateTxts(bcryptHashes_NORDPASS, txtName: "bcryptHashes_NORDPASS");
    generateTxts(scryptHashes_NORDPASS, txtName: "scryptHashes_NORDPASS");
    generateTxts(pbkdf2Hashes_NORDPASS, txtName: "pbkdf2Hashes_NORDPASS");
    generateTxts(argon2Hashes_NORDPASS, txtName: "argon2Hashes_NORDPASS");

    // 10K PASSWORDS LIST

    String[] list10k = get10kList();

    ArrayList<String> bcryptHashes_10K = new ArrayList<>(gerenciadorHashers.insertPasswords(bCryptPasswordEncoder, list10k));
    ArrayList<String> scryptHashes_10k = new ArrayList<>(gerenciadorHashers.insertPasswords(sCryptPasswordEncoder, list10k));
    ArrayList<String> pbkdf2Hashes_10k = new ArrayList<>(gerenciadorHashers.insertPasswords(pbkdf2PasswordEncoder, list10k));
    ArrayList<String> argon2Hashes_10k = new ArrayList<>(gerenciadorHashers.insertPasswords(argon2PasswordEncoder, list10k));

    generateTxts(bcryptHashes_10K, txtName: "bcryptHashes_10k");
    generateTxts(scryptHashes_10k, txtName: "scryptHashes_10k");
    generateTxts(pbkdf2Hashes_10k, txtName: "pbkdf2Hashes_10k");
    generateTxts(argon2Hashes_10k, txtName: "argon2Hashes_10k");
}
```

Fonte: elaborada pelos autores, 2026.

Para rodar os *benchmarks*, é preciso utilizar a classe Performance. Essa classe contém as *annotations* necessárias para declarar parâmetros referentes ao JMH, vetores contendo as senhas necessárias para o experimento, e a declaração do método do *benchmark* em si. A Figura 7 exibe os parâmetros utilizados pelo teste e a inicialização das listas, e a Figura 8 exibe os métodos executados pelo *benchmark*.

Figura 7 – Classe Performance.

```java
@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class Performance {

    GeradorSenhas geradorSenhas = new GeradorSenhas();
    GerenciadorHashers gerenciadorHashers = new GerenciadorHashers();
    String[] nordpassList = geradorSenhas.senhasNordpass;
    String[] weakpassList = geradorSenhas.get10kList();
```

Fonte: elaborada pelos autores, 2026.

Figura 8 – Métodos de *benchmark* da classe Performance.

```java
// NORDPASS BENCHMARK

//@Benchmark
public void hashWithBcrypt_NORDPASS() {
    gerenciadorHashers.hashPasswords(gerenciadorHashers.bcryptHasher_DEFAULT, nordpassList);
}

@Benchmark
public void hashWithScrypt_NORDPASS() {
    gerenciadorHashers.hashPasswords(gerenciadorHashers.scryptHasher_DEFAULT, nordpassList);
}

//@Benchmark
public void hashWithPbkdf2_NORDPASS() {
    gerenciadorHashers.hashPasswords(gerenciadorHashers.pbkdf2Hasher_DEFAULT, nordpassList);
}

//@Benchmark
public void hashWithArgon2_NORDPASS() {
    gerenciadorHashers.hashPasswords(gerenciadorHashers.argon2Hasher_DEFAULT, nordpassList);
}

// WEAKPASS BENCHMARK

//@Benchmark
public void hashWithBcrypt_WEAKPASS() {
    gerenciadorHashers.hashPasswords(gerenciadorHashers.bcryptHasher_DEFAULT, weakpassList);
}

@Benchmark
public void hashWithScrypt_WEAKPASS() {
    gerenciadorHashers.hashPasswords(gerenciadorHashers.scryptHasher_DEFAULT, weakpassList);
}

@Benchmark
public void hashWithPbkdf2_WEAKPASS() {
    gerenciadorHashers.hashPasswords(gerenciadorHashers.pbkdf2Hasher_DEFAULT, weakpassList);
}

//@Benchmark
public void hashWithArgon2_WEAKPASS() {
    gerenciadorHashers.hashPasswords(gerenciadorHashers.argon2Hasher_DEFAULT, weakpassList);
}
```

Fonte: elaborada pelos autores, 2026.

O método *run* provém mais configurações do *benchmark*, como a quantidade de iterações de aquecimento, iterações de medição, a medição primária (tempo médio) e

secundária (memória consumida) e a checagem dos resultados para exibição na tela do console. A Figura 9 mostra a implementação do método *run*.

Figura 9 – Método *run* da classe Performance.

```
public void run() throws RunnerException {

    Options opt = new OptionsBuilder()
            .include(Performance.class.getSimpleName())
            .forks(value: 1)
            .warmupIterations(value: 2)
            .measurementIterations(count: 3)
            .addProfiler(profiler: GCProfiler.class) // <--- allocations
            .addProfiler(profiler: "jfr")
            .build();

    Collection<RunResult> results = new Runner(opt).run();

    // Map benchmark name -> time & allocated bytes
    Map<String, String> summary = new HashMap<>();

    for (RunResult result : results) {
        String name = result.getParams().getBenchmark();
        double avgTimeMs = result.getPrimaryResult().getScore(); // average time in ms
        double allocatedBytes = result.getSecondaryResults().get(key: "gc.alloc.rate.norm").getScore();
        summary.put(name, String.format(format: "AvgTime: %.2f ms, Memory Allocated per operation: %.0f bytes", avgTimeMs, allocatedBytes));
    }

    System.out.println(x: "\n=== Hashing Benchmark Summary ===");
    summary.forEach((benchmark, stats) -> System.out.println(benchmark + " -> " + stats));

}
```

Fonte: elaborada pelos autores, 2026.

Por fim, a classe *ProjetohashApplication* é responsável por executar o programa, como exibido na Figura 10.

Figura 10 – Classe ProjetohashApplication.

```
@SpringBootApplication
public class ProjetohashApplication {

    Run | Debug
    public static void main(String[] args) throws IOException, RunnerException {
        SpringApplication.run(ProjetohashApplication.class, args);

        System.out.println(x: "-- Bem vindo ao Projeto Spring Security Hash --");

        Performance performance = new Performance();
        performance.run();


    }

}
```

Fonte: elaborada pelos autores, 2026.

2.2.2 Procedimento do *benchmark* de segurança

Após as senhas das listas de senhas terem sido submetidas ao *hashing*, a segunda etapa do experimento é utilizar o software *Hashcat* para verificar quantas senhas podem ser descobertas, utilizando um ataque de dicionário *offline*, com o dicionário sendo o notório *rockyou*.txt. O *Hashcat* tem como parâmetros de entrada principais o modo (-m), que é qual algoritmo de *hash* que será utilizado para quebrar os *hashes*, o tipo de ataque (-a), que é um ataque de dicionário puro, o arquivo de saída (as senhas quebradas) e o arquivo de entrada. As

flags --status e --status-timer respectivamente mostram o status atual da quebra de senhas, e de quanto em quantos segundos os dados do status são atualizados. A *flag* --runtime determina por quanto tempo em segundos que o *benchmark* irá ser executado, sendo no caso 3 horas. A Figura 11 exibe o comando exato para realizar o primeiro ataque contra a lista de *hashes* de BCrypt das senhas da NordPass, e a Figura 12 mostra o status inicial do ataque.

Figura 11 – Comando utilizado para realizar um ataque na lista *bcryptHashes_NORDPASS*.

```
C:\hashcat-7.1.2>hashcat -m 3200 -a 0 --status --status-timer=10 --runtime=10800 -o experimento_benchmark_seguranca/crac
ked_bcryptNordpass.txt experimento_benchmark_seguranca/bcryptHashes_NORDPASS.txt experimento_benchmark_seguranca/rockyou
.txt
```

Fonte: elaborada pelos autores, 2026.

Figura 12 – Saída de status do Hashcat.

```
* Device #01: Radeon RX 5500 XT, 3939/4080 MB, 11MCU

OpenCL API (OpenCL 2.1 AMD-APP (3652.0)) - Platform #1 [Advanced Micro Devices, Inc.]
================================================================================
* Device #02: Radeon RX 5500 XT, skipped

Minimum password length supported by kernel: 0
Maximum password length supported by kernel: 72
Minimum salt length supported by kernel: 0
Maximum salt length supported by kernel: 256

Hashes: 200 digests; 200 unique digests, 200 unique salts
Bitmaps: 16 bits, 65536 entries, 0x0000ffff mask, 262144 bytes, 5/13 rotates
Rules: 1

Optimizers applied:
* Zero-Byte

Watchdog: Temperature abort trigger set to 90c

Host memory allocated for this attack: 560 MB (4213 MB free)

Dictionary cache hit:
* Filename..: experimento_benchmark_seguranca/rockyou.txt
* Passwords.: 14344384
* Bytes.....: 139921497
* Keyspace..: 14344384

[s]tatus [p]ause [b]ypass [c]heckpoint [f]inish [q]uit =>

Session..........: hashcat
Status...........: Running
Hash.Mode........: 3200 (bcrypt $2*$, Blowfish (Unix))
Hash.Target......: experimento_benchmark_seguranca/bcryptHashes_NORDPASS.txt
Time.Started.....: Fri Apr 24 12:18:07 2026 (9 secs)
Time.Estimated...: Thu Oct 08 15:56:06 2026 (167 days, 3 hours; Runtime limited: 2 hours, 59 mins)
Kernel.Feature...: Pure Kernel (password length 0-72 bytes)
Guess.Base.......: File (experimento_benchmark_seguranca/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#01........:      197 H/s (26.74ms) @ Accel:1 Loops:32 Thr:16 Vec:1
Recovered........: 2/200 (1.00%) Digests (total), 2/200 (1.00%) Digests (new), 2/200 (1.00%) Salts
Progress.........: 1584/2868876800 (0.00%)
Rejected.........: 0/1584 (0.00%)
Restore.Point....: 0/14344384 (0.00%)
Restore.Sub.#01..: Salt:9 Amplifier:0-1 Iteration:928-960
Candidate.Engine.: Device Generator
Candidates.#01...: 123456 -> chicken
Hardware.Mon.#01.: Temp: 60c Fan: 47% Util: 99% Core:1755MHz Mem:1736MHz Bus:8

[s]tatus [p]ause [b]ypass [c]heckpoint [f]inish [q]uit =>
```

Fonte: elaborada pelos autores, 2026.

O mesmo padrão é executado para o Argon2, porém o PBKDF2 e SCrypt precisam de mais um procedimento para realizar o *benchmark* de segurança. O PBKDF2 no Spring Security por padrão é codificado em formato hexadecimal, diferente dos outros *hashes* mostrados, e o *Hashcat* demanda por codificação em *Base64* para o *benchmark*. Para realizar a conversão, foi utilizado um *script* feito em Python, cujo código é mostrado na Figura 13.

Figura 13 – *Script* de conversão de *hash* PBKDF2 em *hex* para *Base64*.

```
import base64

INPUT_FILE = "C:\hashcat-7.1.2\experimento_benchmark_seguranca\pbkdf2_10kList.txt"
OUTPUT_FILE = "C:\hashcat-7.1.2\experimento_benchmark_seguranca\pbkdf2_10kList_CONVERTED.txt"
ITERATIONS = 310000  # Spring Security v5.8 default

def spring_to_hashcat(line, iterations=ITERATIONS):
    line = line.strip()
    if not line:
        return None

    # First 16 bytes (32 hex chars) = salt
    salt_hex = line[:32]
    # Remaining 32 bytes (64 hex chars) = hash
    hash_hex = line[32:]

    # Convert hex → base64
    salt_b64 = base64.b64encode(bytes.fromhex(salt_hex)).decode()
    hash_b64 = base64.b64encode(bytes.fromhex(hash_hex)).decode()

    return f"sha256:{iterations}:{salt_b64}:{hash_b64}"

def main():
    with open(INPUT_FILE, "r") as infile, open(OUTPUT_FILE, "w") as outfile:
        for line in infile:
            converted = spring_to_hashcat(line)
            if converted:
                outfile.write(converted + "\n")

if __name__ == "__main__":
    main()
```

Fonte: elaborada pelos autores, 2026.

No caso do SCrypt, o *hash* vindo do Spring Security não segue o padrão de *string* de *hash* do *Hashcat*, sendo esse "SCRYPT:N:r:p:salt:hash", então outro *script* Python foi feito para realizar a conversão de formatos. O Spring Security transforma os parâmetros do SCrypt em hexadecimal e os junta no início da *string*, como pode ser visto no *hash* de exemplo "100801hMZXku+TDzgLKzR6o+9IfA==$q6jo0rRxXTWBF2BJMREs8OB8k3G51HeB9CUTE6CV5zk=". O número 10 pode ser lido como 0x10, que corresponde a 16, sendo 2^{16} o custo de CPU (65536), 0x08 que corresponde a 8 (custo de memória) e 0x01 (paralelismo). Após realizar a transformação para um formato interpretável pelo *Hashcat*, foi necessário usar a *flag* --self-test-disable para impedir um erro de teste inicial. Vale ressaltar que desabilitar o *self test* causa uma pequena diminuição de performance do *benchmark*, porém essa diminuição é considerada negligível.

A Figura 14 mostra o código do *script* Python usado para converter o formato de *hash* vindo do Spring Security para um formato interpretável pelo *Hashcat*.

Figura 14 – *Script* de adequação e *parsing* do SCrypt.

```
def parse_spring_scrypt(hash_str):
    if not hash_str.startswith("$"):
        raise ValueError("Formato inválido")

    parts = hash_str.strip().split("$")

    # ['', params, salt, hash]
    if len(parts) != 4:
        raise ValueError("Formato inesperado")

    params_hex = parts[1]
    salt_b64 = parts[2]
    hash_b64 = parts[3]

    params = int(params_hex, 16)

    log2_n = (params >> 16) & 0xFFFF
    r = (params >> 8) & 0xFF
    p = params & 0xFF

    N = 2 ** log2_n

    return f"SCRYPT:{N}:{r}:{p}:{salt_b64}:{hash_b64}"


def convert_file():
    input_file = "scryptHashes_NORDPASS.txt"
    output_file = "scryptHashesCONVERTED2_NORDPASS.txt"

    total = 0
    success = 0

    with open(input_file, "r", encoding="utf-8") as infile, \
         open(output_file, "w", encoding="utf-8") as outfile:

        for line in infile:
            total += 1
            line = line.strip()

            if not line:
                continue

            try:
                # já está no formato hashcat → só copia
                if line.startswith("SCRYPT:"):
                    outfile.write(line + "\n")
                    success += 1
                    continue

                converted = parse_spring_scrypt(line)
                outfile.write(converted + "\n")
                success += 1

    print("\n=== RESUMO ===")
    print(f"Total: {total}")
    print(f"Convertidos: {success}")

# ===== EXECUÇÃO =====
if __name__ == "__main__":
    convert_file()
```

Fonte: elaborada pelos autores, 2026.

2.2.3 Hardware e software utilizados

O Quadro 1 exibe o hardware utilizado para o *benchmark*, e o Quadro 2 mostra o versionamento do software relevante utilizado, em um sistema operacional Windows 11.

Quadro 1 – Configuração do hardware usado.

Componentes	Especificações
CPU	Intel Core i5-10600k 4.10GHz
Memória RAM	8GB Kingston Fury Beast DDR4 3200MHz KFC3200C16D4/8GX e 8GB T-FORCE TeamGroup DDR4 2666MHz TEAMGROUP-UD4-2666
Placa-mãe	ASUS H410M-E
GPU	Radeon RX 5500XT 4GB

Fonte: elaborada pelos autores, 2025.

Quadro 2 – Configuração do software usado.

Software	Versão
JDK	OpenJDK 21.0.7
Spring Boot	3.5.4
Spring Security	6.5.2
Hashcat	7.1.2
JMH	1.37

Fonte: elaborada pelos autores, 2025.

2.3 Resultados e discussões

Inicialmente, foram coletados dados referentes às 2 listas de senhas, seguindo a métrica de tempo de execução e alocação total de memória durante o tempo decorrido. A Tabela 1 exibe os resultados do *benchmark* de tempo de execução médio em minutos do *hashing* da lista de senhas retiradas da NordPass de 200 senhas e a lista da WeakPass de 10000 senhas.

Tabela 1 – Resultado do teste de tempo de execução médio das listas da NordPass e WeakPass em minutos

Tempo de execução médio (min)		
Algoritmo	NordPass	WeakPass
BCrypt	2,05 min	103,08 min
SCrypt	7,91 min	380,97 min

PBKDF2	22,35 min	1118,5 min
Argon2	2,79 min	143,94 min

Fonte: elaborada pelos autores, 2025.

Os resultados demonstram uma clara distinção de velocidade entre os algoritmos. Enquanto o BCrypt demonstra ser o mais rápido com seu *cost factor* padrão de 10, o algoritmo PBKDF2 é cerca de 10 a 11 vezes mais lento. O SCrypt é o penúltimo, performando 3,70 a 3,80 vezes mais devagar, e Argon2 é 40% mais lento que BCrypt.

Juntamente com o *benchmark* de tempo de execução, também foram coletados dados referentes à alocação de memória. A Tabela 2 exibe a quantidade de memória alocada em megabytes ao decorrer de todo o processo de *hashing* da lista de senhas retirada da NordPass e da lista de senhas da WeakPass.

Tabela 2 – Resultado da medição de memória das listas da NordPass e WeakPass em megabytes.

Alocação de memória total (mb)		
Algoritmo	NordPass	WeakPass
BCrypt	383,24 mb	19257,96 mb
SCrypt	13358,74 mb	671293,14 mb
PBKDF2	2961,67 mb	148827,24 mb
Argon2	3461,20 mb	173929,29 mb

Fonte: elaborada pelos autores, 2025.

Na comparação de consumo de memória, o algoritmo menos intensivo no uso de memória é o BCrypt, seguido de PBKDF2, Argon2, e por fim, SCrypt. Os dois últimos algoritmos citados fazem um grande uso de memória, primariamente com o objetivo de diminuir a efetividade de ataques utilizando certos tipos de hardware como GPUs (*Graphic Processing Units*), e ASICs (*Application Specific Integrated Circuits*) projetados para calcular *hashes* rapidamente.

Após os *benchmarks* de performance, foram realizados testes no software Hashcat. Os *hashes* da lista da NordPass e da WeakPass de cada algoritmo foram submetidos a um ataque de dicionário *offline*. A Tabela 3 mostra os resultados da lista da NordPass e da WeakPass.

Tabela 3 – Resultado do ataque de dicionário aos *hashes* da lista da NordPass.

Quantidade de senhas recuperadas		
Algoritmo	NordPass	WeakPass
BCrypt	174	214
SCrypt	155	21
PBKDF2	186	768
Argon2	174	477

Fonte: elaborada pelos autores, 2025.

Tanto BCrypt como Argon2 performaram igualmente na lista da NordPass, enquanto SCrypt consistentemente performou como o algoritmo mais resistente entre as duas listas, tendo sido descobertas a menor quantidade de senhas. PBKDF2 demonstrou ser o menos resistente.

Dentre os 4 algoritmos, BCrypt demonstrou ter o menor consumo de memória e tempo de execução médio, mostrando que essa implementação é adequada para aplicações em dispositivos com baixos recursos computacionais, como sistemas embarcados ou computadores de placa única.

Por outro lado, o SCrypt apresentou o maior consumo de memória e maior resistência a ataques de dicionário, que, quando combinado com o recurso que mitiga a eficácia da utilização de GPUs e FPGAs para quebrar o *hash*, aparenta ser o algoritmo mais apto para aplicações em dispositivos com recursos computacionais de média a larga escala, como servidores e computadores pessoais.

Vale ressaltar que Argon2 possui recursos semelhantes aos do SCrypt, porém, tem um tempo médio de execução de *hashing* e consumo de memória significativamente menores, que, mesmo com uma inferior resistência a ataques, pode tornar-se uma opção viável em aplicações nas quais não se há certeza sobre os recursos computacionais disponíveis ou há uma preferência em uma implementação consistentemente performática.

Existem 2 ressalvas a serem feitas sobre as implementações padrão dos algoritmos SCrypt e Argon2. Ambos os algoritmos estão utilizando a biblioteca Bouncy Castle do Java, que, de acordo com a documentação do Spring (Spring, 2026) não explora a *feature* de paralelismo dos algoritmos, fragilizando a potência da implementação e causando uma assimetria de cálculo de *hashes* entre defensor e atacante, tanto que ambas as implementações têm o parâmetro de paralelismo igual a 1. Além disso, o SCrypt é baseado em Salsa20, que, também segundo a documentação, performa pobremente em Java.

De forma geral, as implementações padrão recomendadas são dos algoritmos SCrypt e BCrypt. Embora o BCrypt seja o algoritmo mais antigo, o parâmetro de *cost factor* determinado em sua implementação padrão ainda permanece confiável, sendo demonstrado pela sua velocidade e consumo de memória adequados. O SCrypt é uma alternativa de alto consumo que obriga atacantes a alocarem mais recursos computacionais para "quebrarem" os *hashes*, enquanto entrega velocidade muito superior à pior implementação estudada, a do PBKDF2. Mesmo sendo o mais lento da seleção de algoritmos, *hashes* calculados com ele foram quebrados em velocidade muito maior, provavelmente devido a certos otimizadores que o Hashcat utilizou, notavelmente "Zero-byte" e "Slow-Hash-SIMD-LOOP". Portanto, a implementação padrão do PBKDF2 apresentou desempenho inferior quando comparado aos demais algoritmos. Argon2 é considerado de forma geral o melhor algoritmo fora do Spring, porém sua implementação no contexto do Spring Security e o uso da biblioteca Bouncy Castle parecem afetar sua performance ainda mais que a do SCrypt, tornando-o uma boa, porém aparentemente menos performática, escolha de algoritmo.

3 CONSIDERAÇÕES FINAIS

Senhas são textos privados para o uso de sistemas e aplicações tanto na web quanto fora dela. Para a proteção de senhas, são muitas vezes utilizados algoritmos de *hashing* para ocultá-las e preservar a confidencialidade do usuário-alvo. É de suma importância então investigar os melhores algoritmos de *hashing* e suas implementações sob o pretexto de buscar a ideal performance para os sistemas desenvolvidos. Para tal fim, foram estudadas implementações de algoritmos de *hashing* no *framework* Spring Security.

Os resultados demonstram as diferenças características da implementação dos algoritmos BCrypt, SCrypt, PBKDF2 e Argon2 pelo Spring Security, e revelam uma grande discrepância na segurança, consumo de memória e tempo de execução de cada algoritmo

implementado. É observável a necessidade de análise do algoritmo de *hashing* adequado para cada contexto em que ele será utilizado.

O BCrypt, com o seu baixo consumo de memória e tempo de execução, demonstrou ser o mais adequado em aplicações com baixos recursos computacionais, enquanto o SCrypt, com o seu alto consumo de memória e alta resistência a ataques de dicionário, indica ser o algoritmo mais apto para aplicações com altos recursos computacionais. Argon2 apresentou um consumo de memória e tempo de execução significativamente menores quando comparado ao SCrypt, porém, ainda manteve uma alta taxa de resistência a ataques de dicionário, o que o torna uma alternativa adequada para aplicações com recursos computacionais médios, incertos ou híbridos. A implementação do PBKDF2 teve a pior performance observada.

A comparação demonstrada neste artigo auxilia na diferenciação da performance das implementações padrão dos quatro algoritmos oferecidos pelo Spring Security, fornecendo insights para desenvolvedores que buscam saber qual é a melhor opção levando em consideração a utilização de recursos computacionais.

Propomos então que futuramente, mais estudos sejam feitos, utilizando hardware comumente encontrado em servidores, levando em consideração que este trabalho foi feito utilizando um computador pessoal, o que pode não refletir a realidade de performance das implementações observadas em contextos reais de aplicações empresariais. Seria também válido realizar testes com outros softwares de quebra de *hash*, como *John the Ripper*. Vale lembrar que uma limitação do estudo atual é de que foram comparadas implementações padrão, parametrizadas de acordo com o que desenvolvedores do Spring Security determinaram. O estudo não buscou equalizar os parâmetros de nenhuma forma, sendo essa prática uma interessante direção futura para estudos posteriores.

REFERÊNCIAS

COELHO, G.; EMANUEL, V.; PEREIRA, V.; SANTOS, F. da S.; COSTA, L. M. da S. S. **Criptografia de dados em nuvem: técnicas e algoritmos mais utilizados.** *In*: Seminário de Pesquisa e Iniciação Científica – Nova UBM, 6., 2023, Barra Mansa. Anais eletrônicos [...]. Barra Mansa, v. 6, n. 1, p. 155-168, 2023. Disponível em: https://revista.ubm.br/index.php/engenhariaseexatas/article/view/2311/635. Acesso em: 8 ago. 2025.

ERTAUL, L.; KAUR, M.; GUDISE, V. A. K. R. **Implementation and performance analysis of PBKDF2, Bcrypt, Scrypt algorithms.** *In*: International Conference on Wireless Networks (ICWN), 2016, Las Vegas. Proceedings […]. Las Vegas: CSREA Press, 2016. p. 66-72. Disponível em: https://www.worldcomp-proceedings.com/proc/p2016/ICW3865.pdf. Acesso em: 14 ago. 2025.

FEDORCHENKO, V.; YEROSHENKO, O.; SHMATKO, O.; KOLOMIITSEV, O.; OMAROV, M. Password hashing methods and algorithms on the .NET platform. **Advanced Information Systems**. Ucrânia. v. 8, n. 4, p. 82-92, 2024. Disponível em: http://ais.khpi.edu.ua/article/view/314569/305491. Acesso em: 4 set. 2025.

GOOGLE. **Increase the speed of your mobile site with this toolkit.** Disponível em: https://blog.google/products/admanager/increase-speed-of-your-mobile-site-wi/. Acesso em: 29 mai. 2026.

HATZIVASILIS, G.; PAPAEFSTATHIOU, I.; MANIFAVAS, C. Password hashing competition - survey and benchmark. **Cryptology ePrint Archive**, Paper 2015/265, 2015. Disponível em: https://eprint.iacr.org/2015/265.pdf. Acesso em: 24 ago. 2025.

MANANKOVA, O. A.; YABUKOVA, M. Z.; RAKHMATULLAEV, M.A.; BAIKENOV, A. S. Simulation of the rainbow attack on the SHA-256 hash function. **Journal of Theoretical and Applied Information Technology**. Paquistão. v. 101, n. 4, p. 1594-1603, 2023. Disponível em: https://www.jatit.org/volumes/Vol101No4/36Vol101No4.pdf. Acesso em: 11 set. 2025.

NORDPASS. **Top 200 most common passwords.** Disponível em: https://nordpass.com/most-common-passwords-list/. Acesso em: 4 set. 2025.

OWASP FOUNDATION. **Password storage cheat sheet.** Disponível em: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. Acesso em: 26 ago. 2025.

PITTALIA, P. P. A comparative study of hash algorithms in cryptography. International **Journal of Computer Science and Mobile Computing**. Suécia. v. 8, n. 6, p. 147-152, 2019. Disponível em: https://ijcsmc.com/docs/papers/June2019/V8I6201928.pdf. Acesso em: 22 ago. 2025.

PRIMA, K. W. Balancing data security: a comparative review of hashing and non-hashing approaches in data storage. **Journal of Embedded System Security and Intelligent System**. Indonesia. v. 6, n. 2, p. 268-278, 2025. Disponível em: https://journal.unm.ac.id/index.php/JESSI/article/view/9257/5561. Acesso em: 22 abr. 2026.

QUEIROZ, L. G. de O. **Funções de hash e gerenciamento de senhas.** Orientador: Ruy de Queiroz. 2022. 58 f. Trabalho de Graduação (Graduação em Engenharia da Computação) - Universidade Federal de Pernambuco, Recife, 2022. Disponível em: https://www.cin.ufpe.br/~tg/2022-1/tg_EC/TG_lgoq.pdf. Acesso em: 21 abr. 2026.

ROSULEK, M. **The joy of cryptography.** Cambridge, Massachusetts: MIT Press, 2021. E-book. Disponível em: https://joyofcryptography.com/. Acesso em: 24 ago. 2025.

SPRING. **Spring Security Reference Documentation.** Disponível em: https://docs.spring.io/spring-security/reference/. Acesso em: 2 jul. 2026.

SRIRAMYA, P.; KARTHIKA, R. A. Providing password security by salted password hashing using Bcrypt algorithm**.** ARPN **Journal of Engineering and Applied Sciences.** Paquistão. v. 10, n. 13, p. 5551-5556, 2015. Disponível em: https://www.arpnjournals.com/jeas/research_papers/rp_2015/jeas_0715_2288.pdf. Acesso em: 22 ago. 2025.

WEAKPASS. **10_million_password_list_top_10000.txt.** Disponível em: https://weakpass.com/wordlists/10_million_password_list_top_10000.txt#download. Acesso em: 4 set. 2025.