Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Segurança em APIs de Criptografia: Um Estudo sobre Usos Indevidos em Sistemas Corporativos

Joilton Almeida de Jesus

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2025

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

JJ58ss

Jesus, Joilton Almeida de
Segurança em APIs de Criptografia: Um Estudo sobre Usos Indevidos em Sistemas Corporativos / Joilton Almeida de Jesus; orientador Rodrigo Bonifácio de Almeida. Brasília, 2025.
94 p.

Dissertação(Mestrado Profissional em Computação Aplicada) Universidade de Brasília, 2025.

1. APIs de Criptografia. 2. Análise Estática de Código. 3. Correções de Vulnerabilidades no uso de APIs de Criptografia. 4. DevSecOps. 5. Segurança de Software. I. Almeida, Rodrigo Bonifácio de, orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Segurança em APIs de Criptografia: Um Estudo sobre Usos Indevidos em Sistemas Corporativos

Joilton Almeida de Jesus

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof.a Dr.a Elaine Venson
FCTE/UnB

Prof. Dr. Anderson Uchôa
UFC

Prof.a Dr.a Edna Dias Canedo
Coordenadora do Programa de Pós-graduação em Computação Aplicada

Brasília, 27 de Novembro de 2025

Dedicatória

Dedico este trabalho à memória dos meus pais, que, com simplicidade e sabedoria, me ensinaram o valor do trabalho, da honestidade e da perseverança.
Mesmo ausentes, permanecem vivos em cada conquista e em cada passo da minha caminhada.

(in memoriam)

Agradecimentos

Registro meus sinceros agradecimentos ao Prof. Dr. Rodrigo Bonifácio, meu orientador, pela orientação precisa, pelos ensinamentos compartilhados e pela confiança depositada ao longo de todo o desenvolvimento deste trabalho.

À minha esposa Cláudia e aos meus filhos Rafael e Vivian, pelo apoio, paciência e compreensão em todos os momentos em que precisei me ausentar para me dedicar aos estudos. O apoio e o carinho de vocês foram essenciais para que eu chegasse até aqui.

Aos professores do Programa de Pós-Graduação em Computação Aplicada (PPCA), pelos conhecimentos transmitidos e pela contribuição para minha formação acadêmica e profissional.

À Profa. Dra. Elaine Venson e ao Prof. Dr. Anderson Uchôa, membros das bancas de qualificação e de defesa, pelas análises criteriosas e pelas contribuições que aprimoraram significativamente este trabalho.

Aos meus colegas da SDSS/GTI/Embrapa, em especial àqueles que participaram do Grupo Focal, pela colaboração, disponibilidade e contribuições ao longo do estudo.

Ao meu supervisor, Murilo Martins, pelo apoio, incentivo e compreensão ao longo deste percurso, tornando possível conciliar as atividades profissionais com a realização deste mestrado.

Aos colegas de mestrado, pela convivência e troca de experiências que tornaram esta trajetória mais produtiva e enriquecedora.

Ao Luís Amaral e à Carine Ferreira, pela colaboração, incentivo e palavras de apoio nos momentos de maior desafio.

Ao Marcos Antônio, pela parceria desde o início do projeto até a conclusão dessa jornada.

À Ana Cláudia e à Gleice, pelo incentivo desde a época em que fui aluno especial até o ingresso e a permanência no mestrado.

E, por fim, a todos que me incentivaram a iniciar e a prosseguir neste mestrado, expresso minha gratidão e reconhecimento.

Resumo

Este trabalho investiga vulnerabilidades associadas ao uso inadequado de APIs de criptografia em sistemas corporativos desenvolvidos em Java na Empresa Brasileira de Pesquisa Agropecuária (Embrapa). A pesquisa utilizou as ferramentas de análise estática CogniCrypt e CryptoGuard para identificar usos indevidos das APIs em oito sistemas, sete webservices e dois componentes reutilizáveis da instituição. Por meio de um estudo de caso composto por duas etapas — o Estudo 1, que reúne a análise estática de código e um grupo focal com desenvolvedores, e o Estudo 2, que contempla a correção manual das vulnerabilidades e a avaliação de sugestões de correção geradas por modelos de linguagem (SLMs e LLMs) — foi possível mensurar a prevalência de vulnerabilidades nos sistemas analisados, compreender as percepções das equipes técnicas sobre os alertas emitidos pelas ferramentas, avaliar os impactos das correções manuais implementadas e examinar as sugestões de correção produzidas por SLMs/LLMs quanto à sua efetividade e aderência às recomendações de segurança. Espera-se, com os resultados obtidos, subsidiar futuras práticas de segurança em desenvolvimento de software, promovendo maior proteção e eficiência nos sistemas corporativos.

Palavras-chave: APIs de Criptografia, Análise Estática de Código, Segurança de Software.

Abstract

This work investigates vulnerabilities associated with the improper use of cryptographic APIs in Java-based corporate systems at the Brazilian Agricultural Research Corporation (Embrapa). The study employed the static analysis tools CogniCrypt and CryptoGuard to identify API misuses in eight systems, seven webservices, and two reusable components within the institution. Through a case study composed of two stages — Study 1, which combines static code analysis and a focus group with developers, and Study 2, which comprises the manual correction of vulnerabilities and the evaluation of correction suggestions generated by language models (SLMs and LLMs) — it was possible to measure the prevalence of vulnerabilities in the analyzed systems, understand the technical teams' perceptions of the warnings issued by the tools, assess the impact of the manual corrections applied to the code, and examine the correction suggestions produced by SLM/LLM regarding their effectiveness and adherence to security recommendations. The results are expected to support future secure software development practices, promoting greater protection and efficiency in corporate systems.

Keywords: Cryptographic APIs, Static Code Analysis, Software Security.

Sumário

1 Introdução **1**

1.1 Apresentação do tema e sua relevância 1
1.2 Justificativa 3
1.3 Contribuição 4
1.4 Objetivos 4
1.4.1 Objetivo Geral 4
1.4.2 Objetivos Específicos 5
1.4.3 Questões de pesquisa 5
1.5 Estrutura do Texto 5

2 Fundamentação Teórica **7**

2.1 Criptografia e APIs de Criptografia 7
2.1.1 Conceitos básicos de criptografia 8
2.1.2 Funcionamento e importância das APIs de criptografia 10
2.2 Usos Indevidos de APIs de Criptografia 11
2.3 Detalhamento das Vulnerabilidades Criptográficas 13
2.3.1 Modos de Operação de Cifra de Bloco 13
2.3.2 Derivação Segura de Chaves 14
2.3.3 Geradores de Números Aleatórios (PRNG) 14
2.3.4 Segurança em Protocolos de Transporte (SSL/TLS) 15
2.4 Análise Estática de Código 16
2.4.1 CogniCrypt 17
2.4.1.1 CogniCrypt: Análise Estática e Suporte ao Desenvolvedor . . 17
2.4.1.2 CogniCryptSAST: Expansão da Análise Estática no CogniCrypt 18
2.4.1.3 CogniCryptGen: Geração Segura de Código Criptográfico . . 19
2.4.1.4 CrySL: A Linguagem de Especificação de Segurança do CogniCrypt 20
2.4.1.5 Os Resultados e o Impacto do CogniCrypt 21
2.4.2 CryptoGuard 22

2.4.3 Comparação e Contribuições 24

3 Caracterização do Estudo de Caso 26
3.1 Objetivos, Questões e Métricas 26
3.2 Contexto da Pesquisa: Embrapa 28
3.2.1 Estrutura Organizacional 28
3.2.2 Processo de Desenvolvimento 29
3.2.3 Adoção da Tecnologia Java 29
3.3 Projetos de Software Alvo do Estudo de Caso 30
3.4 Metodologia 32
3.4.1 Estudo 1: Análise da Situação Atual 33
3.4.1.1 Seleção de sistemas e coleta de informações 33
3.4.1.2 Grupo Focal 34
3.4.2 Estudo 2: Resolução das Vulnerabilidades Criptográficas 37
3.4.2.1 Correção manual dos usos indevidos de APIs de criptografia . 37
3.4.2.2 Correção semiautomatizada com SLMs/LLMs 38

4 Resultados do Estudo 1: Análise da Situação Atual 40
4.1 Resultados da Análise Estática 41
4.1.1 Vulnerabilidades Identificadas pela Análise Estática 41
4.2 Percepções dos Desenvolvedores 42

5 Resultados do Estudo 2: Resolução das Vulnerabilidades Criptográficas 49
5.1 Resultados das correções manuais 50
5.1.1 Correção das vulnerabilidades no código-fonte 50
5.1.2 Integração dos componentes corrigidos nos sistemas 66
5.1.3 Dificuldades encontradas 66
5.1.4 Impactos das correções dos componentes nos sistemas 68
5.2 Resultados das correções semiautomatizadas com SLMs e LLMs 69
5.2.1 Modelos de linguagem utilizados 70
5.2.2 Execução dos modelos SLM 70
5.2.3 Execução dos modelos LLM 71
5.2.4 Técnica *one-shot* nos prompts 72
5.2.5 Análise Quantitativa da Efetividade 74
5.2.6 Avaliação da Suficiência de Uma Única Iteração por Método Vulnerável 76
5.2.7 Principais Erros Identificados 77
5.2.8 Impressões Gerais e Recomendações 78

6 Considerações Finais **80**
6.1 Contribuições e Conclusões Gerais 80
6.2 Trabalhos futuros 81
6.2.1 Normalização e enriquecimento dos relatórios SAST (CogniCrypt e CryptoGuard) 82
6.2.2 Uso de LLMs para mediação didática e interpretação de alertas 82
6.2.3 Integração de análise estática e dinâmica 83
6.2.4 Avaliação e adaptação leve de modelos de linguagem para o domínio criptográfico 83
6.2.5 Controle de variância e reprodutibilidade em experimentos com modelos de linguagem 84
6.2.6 Automação da triagem de alertas por prioridade de risco 85
6.3 Ameaças à Validade da Pesquisa 85
6.4 Síntese e Encerramento 87

Referências **88**

Apêndice **94**

A Volume de Alterações - Correção Manual **95**
A.1 Diferenças de Código nas Correções Manuais 95

B Volume de Alterações - Correção SLM/LLM **97**
B.1 Diferenças de Código nas Correções com SLM e LLM 97

C Código-fonte original analisado nas ferramentas de análise estática **99**
C.1 Código-fonte original do Componente 1 100
C.2 Código-fonte original do Componente 2 104

D Código-fonte corrigido manualmente **106**
D.1 Código-fonte corrigido do Componente 1 107
D.2 Código-fonte corrigido do Componente 2 116

E Código-fonte sugerido por SLM **121**
E.1 Sugestão gerada pela SLM: Qwen 3 4B 122
E.2 Sugestão gerada pela SLM: Mistral-7B 128
E.3 Sugestão gerada pela SLM: Gemma 3n E4B 135

F Código-fonte sugerido por LLM **142**
F.1 Sugestão gerada pela LLM: Gemini 2.5 Pro 143

F.2 Sugestão gerada pela LLM: Claude Sonnet 4 151
F.3 Sugestão gerada pela LLM: ChatGPT 5 164

G Prompt e insumos dos experimentos com modelos de linguagem 174

Lista de Figuras

2.1 Imagem cifrada com o modo ECB. 14

Lista de Tabelas

3.1 Demografia dos Participantes do Grupo Focal. 36

4.1 Vulnerabilidades encontradas nas análises estáticas. 42
4.2 Detalhamento das Vulnerabilidades por Software - Parte 1. 43
4.3 Detalhamento das Vulnerabilidades por Software - Parte 2 44

5.1 Resumo do Volume de Alterações — Correção manual. 66
5.2 Análise Comparativa em Níveis para SLMs. 74
5.3 Análise Comparativa em Níveis para LLMs. 74
5.4 Volume de alterações por modelo — Correções com SLMs/LLMs (Componente 1). 75
5.5 Volume de alterações por modelo — Correções com SLMs/LLMs (Componente 2). 75
5.6 Correção Efetiva por Tipo de Vulnerabilidade - SLMs. 76
5.7 Correção Efetiva por Tipo de Vulnerabilidade - LLMs. 77
5.8 Resumo dos resultados de correção com SLMs e LLMs. 78

A.1 Volume de Alterações por classe — Manual (Componente 1). 95
A.2 Volume de Alterações por classe — Manual (Componente 2). 96

B.1 Volume de alterações por classe — SLMs (Componente 1). 97
B.2 Volume de alterações por classe — LLMs (Componente 1). 98
B.3 Volume de alterações por classe — SLMs (Componente 2). 98
B.4 Volume de alterações por classe — LLMs (Componente 2). 98

Lista de Códigos

2.1 Uso indevido de API de criptografia . . . 13
2.2 Exemplo de execução do CogniCrypt . . . 18
2.3 Exemplo de relatório do CogniCrypt . . . 19
2.4 Regra CrySL para SecretKey . . . 21
2.5 Exemplo de execução do CryptoGuard . . . 23
2.6 Exemplo de relatório do CryptoGuard . . . 23

5.1 Método encrypt com algoritmo inadequado . . . 52
5.2 Trecho do método encrypt com algoritmo correto . . . 53
5.3 Modos de operação sugeridos pelo CogniCrypt . . . 53
5.4 Geração incorreta de chave e utilizando constante . . . 54
5.5 Constantes para métodos auxiliares . . . 54
5.6 Geração correta de chave . . . 55
5.7 Geração de IV . . . 55
5.8 Método encrypt corrigido completo . . . 56
5.9 Geração de Salt . . . 56
5.10 Método decrypt corrigido . . . 58
5.11 Método sslCheck() com implementação inadequada . . . 60
5.12 CogniCrypt — falha em sslCheck() (RecursoWS) . . . 60
5.13 CogniCrypt — falha em sslCheck() (ModuloLoginJAASWS) . . . 61
5.14 Método sslCheck() corrigido . . . 63
5.15 Método decrypt: Falso Positivo no CogniCrypt . . . 64

C.1 `doStartTag()` — original . . . 100
C.2 `sslCheck()` — original (`RecursoWS`) . . . 101
C.3 `encrypt()` e `decrypt()` — originais . . . 101
C.4 `randomChar()` e `randomString()` — originais . . . 102
C.5 `sslCheck()` — original (`ModuloLoginJAASWS`) . . . 102
C.6 `createEasySSLContext()` — original . . . 103
C.7 Método construtor `GerenciadorConfiancaX509` — original . . . 103

C.8 Classe `UtilHashLM` — original . . . 104
C.9 `geraSalt()` — original . . . 104
C.10 `geraSenhaAleatoria()` — original . . . 105

D.1 `doStartTag()` — correção manual . . . 107
D.2 `sslCheck()` — correção manual (`RecursoWS`) . . . 108
D.3 `encrypt()` e `decrypt()` — correção manual . . . 110
D.4 `randomChar()` e `randomString()` — correção manual . . . 111
D.5 `sslCheck()` — correção manual (`ModuloLoginJAASWS`) . . . 112
D.6 `createEasySSLContext()` — correção manual . . . 113
D.7 `FabricaSocketTLSV12` (Nova classe). . . . 114
D.8 Método construtor `GerenciadorConfiancaX509` — correção manual . . . 116
D.9 Classe `UtilHashLM` — correção manual . . . 119
D.10 `geraSalt()` — correção manual . . . 120
D.11 `geraSenhaAleatoria()` — correção manual . . . 120

E.1 `doStartTag()` — SLM (Qwen 3 4B) . . . 122
E.2 `sslCheck()` — SLM (Qwen 3 4B) — (`RecursoWS`) . . . 123
E.3 `encrypt()` e `decrypt()` — SLM (Qwen 3 4B) . . . 124
E.4 `randomChar()` e `randomString()` — SLM (Qwen 3 4B) . . . 124
E.5 `sslCheck()` — SLM (Qwen 3 4B) — (`ModuloLoginJAASWS`) . . . 125
E.6 `createEasySSLContext()` — SLM (Qwen 3 4B) . . . 126
E.7 Método construtor `GerenciadorConfiancaX509` — SLM (Qwen 3 4B) . . . 126
E.8 Classe `UtilHashLM` — SLM (Qwen 3 4B) . . . 127
E.9 `geraSalt()` — SLM (Qwen 3 4B) . . . 128
E.10 `geraSenhaAleatoria()` — SLM (Qwen 3 4B) . . . 128
E.11 `doStartTag()` — SLM (Mistral-7B) . . . 129
E.12 `sslCheck()` — SLM (Mistral-7B) — (`RecursoWS`) . . . 129
E.13 `encrypt()` e `decrypt()` — SLM (Mistral-7B) . . . 130
E.14 `randomChar()` e `randomString()` — SLM (Mistral-7B) . . . 131
E.15 `sslCheck()` — SLM (Mistral-7B) — (`ModuloLoginJAASWS`) . . . 132
E.16 `createEasySSLContext()` — SLM (Mistral-7B) . . . 132
E.17 Método construtor `GerenciadorConfiancaX509` — SLM (Mistral-7B) . . . 133
E.18 Classe `UtilHashLM` — SLM (Mistral-7B) . . . 133
E.19 `geraSalt()` — SLM (Mistral-7B) . . . 134
E.20 `geraSenhaAleatoria()` — SLM (Mistral-7B) . . . 134
E.21 `doStartTag()` — SLM (Gemma 3n E4B) . . . 135
E.22 `sslCheck()` — SLM (Gemma 3n E4B) — (`RecursoWS`) . . . 136

E.23 `encrypt()` e `decrypt()` — SLM (Gemma 3n E4B) . . . 137
E.24 `randomChar()` e `randomString()` — SLM (Gemma 3n E4B) . . . 137
E.25 `sslCheck()` — SLM (Gemma 3n E4B) — (`ModuloLoginJAASWS`) . . . 138
E.26 `createEasySSLContext()` — SLM (Gemma 3n E4B) . . . 138
E.27 Método construtor `GerenciadorConfiancaX509` — SLM (Gemma 3n E4B) 139
E.28 Classe `UtilHashLM` — SLM (Gemma 3n E4B) . . . 140
E.29 `geraSalt()` — SLM (Gemma 3n E4B) . . . 140
E.30 `geraSenhaAleatoria()` — SLM (Gemma 3n E4B) . . . 141

F.1 `doStartTag()` — LLM (Gemini 2.5 Pro) . . . 143
F.2 `sslCheck()` — LLM (Gemini 2.5 Pro) — (`RecursoWS`) . . . 144
F.3 `encrypt()` e `decrypt()` — LLM (Gemini 2.5 Pro) . . . 146
F.4 `randomChar()` e `randomString()` — LLM (Gemini 2.5 Pro) . . . 146
F.5 `sslCheck()` — LLM (Gemini 2.5 Pro) — (`ModuloLoginJAASWS`) . . . 147
F.6 `createEasySSLContext()` — LLM (Gemini 2.5 Pro) . . . 148
F.7 Método construtor `GerenciadorConfiancaX509` — LLM (Gemini 2.5 Pro) 149
F.8 Classe `UtilHashLM` — LLM (Gemini 2.5 Pro) . . . 150
F.9 `geraSalt()` — LLM (Gemini 2.5 Pro) . . . 151
F.10 `geraSenhaAleatoria()` — LLM (Gemini 2.5 Pro) . . . 151
F.11 `doStartTag()` — LLM (Claude Sonnet 4) . . . 152
F.12 `sslCheck()` — LLM (Claude Sonnet 4) — (`RecursoWS`) . . . 153
F.13 `encrypt()` e `decrypt()` — LLM (Claude Sonnet 4) . . . 156
F.14 `randomChar()` e `randomString()` — LLM (Claude Sonnet 4) . . . 156
F.15 `sslCheck()` — LLM (Claude Sonnet 4) — (`ModuloLoginJAASWS`) . . . 158
F.16 `createEasySSLContext()` — LLM (Claude Sonnet 4) . . . 159
F.17 Método construtor `GerenciadorConfiancaX509` — LLM (Claude Sonnet 4) . . . 161
F.18 Classe `UtilHashLM` — LLM (Claude Sonnet 4) . . . 163
F.19 `geraSalt()` — LLM (Claude Sonnet 4) . . . 163
F.20 `geraSenhaAleatoria()` — LLM (Claude Sonnet 4) . . . 164
F.21 `doStartTag()` — LLM (ChatGPT 5) . . . 165
F.22 `sslCheck()` — LLM (ChatGPT 5) — (`RecursoWS`) . . . 165
F.23 `encrypt()` e `decrypt()` — LLM (ChatGPT 5) . . . 167
F.24 `randomChar()` e `randomString()` — LLM (ChatGPT 5) . . . 168
F.25 `sslCheck()` — LLM (ChatGPT 5) — (`ModuloLoginJAASWS`) . . . 168
F.26 `createEasySSLContext()` — LLM (ChatGPT 5) . . . 170
F.27 Método construtor `GerenciadorConfiancaX509` — LLM (ChatGPT 5) . . . 171
F.28 Classe `UtilHashLM` — LLM (ChatGPT 5) . . . 172

F.29 `geraSalt()` — LLM (ChatGPT 5) 173
F.30 `geraSenhaAleatoria()` — LLM (ChatGPT 5) 173

G.1 Prompt completo para sslCheck 178

Lista de Abreviaturas e Siglas

ACM *Association for Computing Machinery.*

AD *Active Directory.*

AEAD *Authenticated Encryption with Associated Data.*

AES *Advanced Encryption Standard.*

API *Application Programming Interface.*

BCA Boletim de Comunicações Administrativas.

CBC *Cipher Block Chaining.*

CI/CD *Continuous Integration/Continuous Deployment.*

CN *Common Name.*

CrySL *Cryptographic Specification Language.*

CSPRNG *Cryptographically Secure Pseudo-Random Number Generator.*

CSV *Comma-Separated Values.*

CTR *Counter Mode.*

CWE *Common Weakness Enumeration.*

DES *Data Encryption Standard.*

DevSecOps *Development, Security, and Operations.*

ECB *Electronic Codebook.*

Embrapa Empresa Brasileira de Pesquisa Agropecuária.

GCM *Galois/Counter Mode*.

GDPR *General Data Protection Regulation*.

GF Grupo Focal.

GTI Gerência de Tecnologia da Informação.

HTTPS *HyperText Transfer Protocol Secure*.

IDE *Integrated Development Environment*.

IV Inicialização de Vetores.

JAAS *Java Authentication and Authorization Service*.

JCA *Java Cryptography Architecture*.

JEP *Java Enhancement Proposal*.

JKS *Java KeyStore*.

JSON *JavaScript Object Notation*.

LDAP *Lightweight Directory Access Protocol*.

LLM *Large Language Model*.

LOC *Lines of Code*.

MAPA Ministério da Agricultura e Pecuária.

MD5 *Message Digest Algorithm 5*.

MITM *Man-in-the-Middle*.

NIST *National Institute of Standards and Tecnology*.

OWASP *Open Web Application Security Project*.

PCI DSS *Payment Card Industry Data Security Standard*.

PDSE Processo de Desenvolvimento de Produtos de *Software* da Embrapa.

PRNG *Pseudo-Random Number Generator*.

PU *Paderborn University.*

REST *Representational State Transfer.*

RSA Rivest-Shamir-Adleman.

SaaS *Software-as-a-Service.*

SAN *Subject Alternative Name.*

SARIF *Static Analysis Results Interchange Format.*

SAST *Static Application Security Testing.*

SHA *Secure Hash Algorithm.*

SLM *Small Language Model.*

SSH *Secure Shell.*

SSL *Secure Sockets Layer.*

TLS *Transport Layer Security.*

TUD *Technische Universität Darmstadt.*

UA *University of Alberta.*

XOR *exclusive OR.*

Capítulo 1

Introdução

1.1 Apresentação do tema e sua relevância

O crescimento de ameaças digitais está cada vez mais se tornando uma preocupação para organizações ou indivíduos que utilizam software em geral. Isso se deve ao aumento de aplicativos que hospedam e processam informações críticas sobre os usuários, como por exemplo dados financeiros e informações médicas. A implementação de criptografia como meio de conter e combater ameaças clássicas e emergentes é vital para garantir a segurança de sistemas críticos.

Conforme observado por Egele et al. [1], sistemas que aparentemente parecem seguros podem, na realidade, ter falhas críticas devido à implementação inadequada de criptografia. Uma situação complicada pela complexidade dos sistemas que utilizam criptografia e pelas dificuldades que os desenvolvedores encontram ao lidarem com *Application Programming Interface*s (APIs) pouco documentadas. Isso é evidenciado por Nadi et al. [2] ao estudar os desafios associados à implementação de tais soluções e todas as vulnerabilidades associadas. Revisões mais recentes, como Ochoa et al. [3], ampliam esse panorama ao caracterizar sistematicamente diferentes formas de uso nocivo de APIs e suas consequências, incluindo casos relacionados a bibliotecas criptográficas, bem como técnicas de reparo disponíveis.

Nesse sentido, a criptografia fornece uma camada de segurança ao converter dados legíveis em um formato criptografado que só pode ser acessado por quem possui a chave de acesso correta. Porém, a eficácia dessa criptografia está diretamente relacionada ao uso adequado da API em cada sistema. Erros no uso desta API podem levar a diversas vulnerabilidades graves que comprometem a segurança dos dados e do ambiente interno da organização [4].

Por outro lado, com o avanço da Engenharia de Software, várias ferramentas de análise estática de código foram concebidas com o intuito de promover um desenvolvimento de

software mais seguro. Essas ferramentas permitem identificar problemas como lógicas incorretas, trechos de código não utilizados e usos indevidos de APIs de criptografia, entre outros. Apesar dos benefícios que essas ferramentas podem proporcionar e da crescente adoção nos últimos anos em fluxos automatizados de *Continuous Integration/Continuous Deployment* (Integração Contínua/Entrega Contínua – CI/CD), conforme evidenciado por estudos recentes sobre o uso prático do SonarQube [5], sua adoção ainda enfrenta alguns obstáculos. Dentre os desafios mais relevantes, destacam-se a dificuldade de integração plena com fluxos de trabalho existentes, a frequência de falsos positivos e a demanda contínua por conhecimentos especializados para a correta interpretação de seus resultados.

Esses desafios são ainda mais significativos no contexto de APIs de criptografia, uma vez que existe uma complexidade inerente nesse domínio. Pesquisas recentes indicam que, apesar das melhorias observadas nas ferramentas de análise estática, a precisão na identificação de vulnerabilidades associadas ao uso indevido de APIs de criptografia continua exigindo atenção para a redução de falsos positivos e negativos [6].

Além disso, o uso de boas práticas ao trabalhar com APIs de criptografia é essencial para a segurança de sistemas. Afinal, como destaca Schneier [4], a implementação é um ponto crítico para a segurança, podendo comprometer a proteção mesmo quando os algoritmos de criptografia são robustos. Portanto, os desenvolvedores devem estar bem informados sobre os princípios gerais de criptografia e padrões de boas práticas que os auxiliem a prevenir vulnerabilidades na implementação do código de software. Sistemas que utilizam APIs de criptografia de forma correta em seus códigos podem diminuir os riscos e proteger informações confidenciais com segurança.

Como exemplo da realidade das organizações brasileiras, a Empresa Brasileira de Pesquisa Agropecuária (Embrapa) é uma das principais empresas que, como a maioria das outras, depende fortemente de sistemas de informação para a manutenção de suas operações. Esses sistemas, desenvolvidos ao longo da história da instituição, apoiam tanto a pesquisa agropecuária quanto a alta gestão e, em diversos casos, fazem uso de APIs de criptografia para proteger informações sensíveis e garantir a integridade dos dados. Assim, a Embrapa — que engloba a própria sede e outras 43 unidades de pesquisa distribuídas em diferentes regiões do Brasil — também está sujeita aos riscos decorrentes do uso inadequado dessas APIs, reforçando a relevância do tema abordado neste estudo.

Considerando tudo o que foi relatado e as consequências do uso indevido de APIs de criptografia, bem como os benefícios de aplicação de ferramentas de análise estática para encontrar esses usos indevidos e corrigi-los, o objetivo deste estudo é investigar quais vulnerabilidades atualmente existem em alguns sistemas corporativos e o impacto de suas correções na estabilidade funcional e na estrutura do código.

1.2 Justificativa

De acordo com Parnas [7], software envelhece, o que ocorre também com tecnologia em geral. Nesse sentido, nas mais diversas instituições, públicas ou privadas, existe a necessidade de renovação em seu parque tecnológico, seja físico ou de software e os sistemas são uma categoria das que mais demonstram essa necessidade. Isso também é verdade na Embrapa visto a sua heterogeneidade de arquiteturas, *frameworks* e tecnologias correlatas no desenvolvimento de software. Algumas destas tecnologias já obsoletas. Diante desse cenário, aderir à práticas de análise estática de código para a verificação de usos inadequados de APIs de criptografia poderia ajudar a descobrir e mitigar potenciais riscos de segurança.

A dificuldade de garantir o uso correto de APIs de criptografia em sistemas, especialmente sistemas complexos e/ou sistemas legados é frequentemente associada à falta de investimento em práticas modernas de engenharia de software, incluindo a análise estática de código. Balalaie et al. [8] observam que a manutenção de sistemas pode ser trabalhosa e demorada devido à sua estrutura complexa, o que pode resultar em uma menor atenção das equipes para práticas de segurança. Anderson [9] destaca que a análise estática de código é crucial para a detecção de uso indevido de APIs criptográficas. A detecção de vulnerabilidades de software, incluindo aquelas relacionadas à criptografia, por meio de análise estática, é frequentemente reconhecida e debatida em literatura especializada, como evidenciado por Anderson em seu livro *Security Engineering'* [9]. Apesar dessa relevância já amplamente reconhecida na literatura especializada [9], e das melhorias recentes implementadas nas ferramentas de análise estática, questões como a capacitação das equipes e o tempo necessário para a realização de testes ainda são recursos escassos. Isso se deve ao tempo frequentemente gasto na manutenção de sistemas legados ou complexos. Esses aspectos restringem a adoção prática dessas ferramentas, especialmente quando se considera a persistência de falsos positivos e negativos em contextos específicos como o das APIs de criptografia [6].

Diante do cenário apresentado sobre os sistemas que são mantidos pela Embrapa na Gerência de Tecnologia da Informação (GTI), observa-se um momento oportuno para compreender vulnerabilidades existentes e associadas à usos indevidos de APIs de criptografia em sistemas utilizando-se ferramentas de análise estática. Para este estudo, serão utilizados 8 sistemas, 7 *webservices* e 2 componentes mantidos pela GTI. Espera-se que tal estudo de caso possa ser replicado para outros sistemas da Embrapa.

1.3 Contribuição

O objetivo deste trabalho é duplo: acadêmico e institucional. Sob o olhar acadêmico, o presente trabalho visa, em primeiro lugar, promover a aplicação de ferramentas de análise estática para o rastreamento do uso indevido de APIs de criptografia em sistemas de software e a descoberta dessas vulnerabilidades. Ao ampliar o conhecimento das vulnerabilidades criptográficas nos sistemas, este trabalho abre o caminho para pesquisas e desenvolvimento de práticas de codificação mais seguras. O estudo visa também obter a percepção dos desenvolvedores em relação aos usos indevidos de APIs criptográficas dos sistemas que eles mantém. Além disso, o trabalho investiga a perspectiva sobre a experiência de correção, abordando o esforço necessário para corrigir componentes reutilizados pelas aplicações, os impactos dessas correções nas aplicações e se a vulnerabilidade foi efetivamente corrigida ou se o risco precisará ser assumido.

Institucionalmente, este trabalho fornece à Embrapa e outras instituições um guia prático que não só rastreia e mitiga riscos de segurança relacionados ao uso de APIs de criptografia, mas também orienta na correção dessas vulnerabilidades. Conforme aponta McGraw [10], a análise estática de código pode identificar vulnerabilidades de segurança desde o início do desenvolvimento, muito antes do código ser utilizado em produção. Além de fornecer orientações para a detecção precoce de vulnerabilidades, este trabalho também aborda a aplicação prática dessas correções, deixando os sistemas não apenas mais seguros, mas estabelecendo um padrão para o desenvolvimento futuro de softwares livres dessas falhas.

Dado o exposto, os resultados obtidos nesta pesquisa contribuem com percepções valiosas ao processo decisório sobre a segurança dos sistemas, possibilitando aos desenvolvedores e equipes de segurança a evolução de suas práticas de codificação, a correção de vulnerabilidades decorrentes do uso indevido de APIs de criptografia em sistemas, e a redução da chance de sistemas com vulnerabilidades exploráveis [11]. A oportunidade de identificar e corrigir falhas de segurança relacionadas à utilização incorreta de APIs de criptografia representa um avanço significativo no desenvolvimento de sistemas com mais segurança e confiabilidade.

1.4 Objetivos

1.4.1 Objetivo Geral

Compreender e corrigir as vulnerabilidades existentes no uso de APIs de criptografia em um subconjunto de sistemas corporativos da Embrapa desenvolvidos em Java, detectadas por meio do uso de ferramentas de análise estática.

1.4.2 Objetivos Específicos

- Identificar vulnerabilidades por meio de ferramentas de análise estática, utilizando as ferramentas CogniCrypt e CryptoGuard.
- Avaliar a percepção de desenvolvedores sobre os *warnings* gerados.
- Implementar as correções e analisar o impacto dessas mudanças, contemplando a estabilidade dos sistemas, o esforço de codificação e a eficácia das correções manuais e semiautomatizadas sugeridas por *Small Language Model*s (Modelo de Linguagem Pequenos – SLMs) e/ou *Large Language Model*s (Modelo de Linguagem Grandes – LLMs).

1.4.3 Questões de pesquisa

(QP1) Qual a prevalência de vulnerabilidades em sistemas corporativos da Embrapa?

(QP2) Como as equipes de desenvolvimento da Embrapa percebem os *warnings* sobre uso incorreto de APIs de criptografia reportados por ferramentas como Cognicrypt e CryptoGuard?

(QP3) Qual o impacto da correção dessas vulnerabilidades na estabilidade funcional e na estrutura de código nos sistemas corporativos da Embrapa?

(QP4) Quão efetivos são os usos de SLMs ou LLMs para as correções das vulnerabilidades identificadas?

1.5 Estrutura do Texto

Este documento está organizado em 6 capítulos. No primeiro encontra-se a introdução apresentando a contextualização do problema, da instituição, a justificativa e a contribuição da proposta deste trabalho.

O capítulo 2 apresenta a fundamentação teórica do trabalho em relação ao tema, trazendo conceitos acerca de criptografia, APIs de criptografia, ferramentas de análise estática e sua utilização.

O capítulo 3 detalha o estudo de caso realizado, descrevendo os objetivos específicos, as questões de pesquisa e as métricas utilizadas. Apresenta o contexto organizacional e tecnológico da Embrapa, os softwares analisados e a metodologia empregada, que inclui análise estática dos softwares, grupo focal com desenvolvedores e a correção das vulnerabilidades identificadas.

O capítulo 4 expõe os resultados obtidos no primeiro estudo, dedicado à análise da situação atual. São apresentados os achados da análise estática conduzida com as ferramentas CogniCrypt e CryptoGuard, bem como as percepções qualitativas dos desenvolvedores discutidas no grupo focal.

O capítulo 5 apresenta os resultados do segundo estudo, voltado à resolução das vulnerabilidades criptográficas. Nesse capítulo são discutidos tanto os impactos das correções manuais aplicadas nos sistemas quanto a efetividade das correções semiautomatizadas geradas por SLMs e LLMs, incluindo suas limitações, erros recorrentes e recomendações práticas.

Por fim, o capítulo 6 reúne as conclusões gerais da pesquisa, destacando as principais contribuições acadêmicas e institucionais, as limitações do trabalho e sugestões para pesquisas futuras.

Capítulo 2

Fundamentação Teórica

Neste capítulo é apresentado embasamento teórico necessário para compreender conceitos básicos de criptografia e APIs de criptografia e a importância destes na segurança de sistemas. São introduzidos os conceitos fundamentais acerca de criptografia e a forma de funcionamento de APIs. Em seguida, é feita apresentação dos problemas e vulnerabilidades comuns associados à investigação do mau uso destas APIs, com foco especial das consequências em sistemas legados. Finalmente, é introduzido o papel da análise estática na detecção deste tipo particular de vulnerabilidade, explicando as técnicas utilizadas pelas ferramentas CogniCrypt e CryptoGuard e a comparação de seus métodos e resultados.

2.1 Criptografia e APIs de Criptografia

A crescente complexidade dos sistemas modernos, combinada à necessidade de proteger dados sensíveis, estimulou o desenvolvimento de APIs de criptografia. Essas abstrações são criadas para abstrair a complexidade dos algoritmos criptográficos, apresentando aos desenvolvedores um meio simplificado para implementar mecanismos de segurança em sistemas. Mesmo com a simplicidade adicional, um uso indevido de APIs de criptografia leva a vulnerabilidades consideráveis e colocam em risco a segurança das operações dependentes dos sistemas.

Conforme mencionado, essas APIs de criptografia, como a *Java Cryptography Architecture* (JCA), desempenham um papel significativo na criação de padrões e na facilitação da utilização de algoritmos criptográficos. Elas permitem que os desenvolvedores integrem funcionalidades criptográficas, autenticação e assinatura digital em suas aplicações, independentemente de terem total compreensão dos algoritmos em questão. No entanto, como afirmam Nadi et al. [2], a falta de conhecimento sobre a própria API e sobre as boas práticas pode levar à sua utilização inadequada, como o uso de chaves fracas ou erros

de configurações. De forma complementar, Ochoa et al. [3] destacam que tais fatores se repetem em diferentes contextos de APIs e constituem causas estruturais recorrentes de usos nocivos.

Além disso, a documentação das APIs de criptografia frequentemente deixa lacunas em termos de exemplos práticos e diretrizes claras. Isso contribui para a dificuldade enfrentada por muitos desenvolvedores ao tentar assegurar que suas implementações estejam em conformidade com os padrões de segurança. O desafio de utilizar corretamente essas APIs será explorado nas próximas subseções, com foco em suas principais vulnerabilidades e nos impactos que seu uso inadequado pode ter, especialmente em sistemas legados.

2.1.1 Conceitos básicos de criptografia

A criptografia consiste em um campo de estudo que tem se dedicado cada vez mais a proteger dados e informações no intuito de garantir confidencialidade e integridade desses dados. Como Schneier [4] enfatiza, a criptografia utiliza meios matemáticos na codificação das informações de uma maneira que, a princípio, apenas quem possui autorização consegue acessá-las. Esse processo é importante visto que cada vez mais informações sensíveis trafegam nas comunicações digitais e essas informações precisam ser protegidas.

A criptografia é normalmente dividida em simétrica e assimétrica e o conceito da diferença desses dois tipos se torna um alicerce fundamental desse campo de estudo. Stallings [12] observa que a criptografia simétrica tem dependência de uma única chave tanto para cifrar quanto para o processo de decifrar. Essa característica faz com que criar uma forma segura para a distribuição das chaves entre nós participantes do processo se torne fundamental. Um exemplo para esse tipo de criptografia é o *Advanced Encryption Standard* (AES), que é usado nas transações bancárias de forma online e para a garantia da proteção dos dados nos dispositivos móveis.

Por outro lado, a criptografia assimétrica, ou criptografia de chave pública, opera com um par de chaves, uma para cifrar e uma para decifrar. Esta implementação permite a comunicação segura sem revelar a chave privada à parte destinatária [13]. O Rivest-Shamir-Adleman (RSA) é um algoritmo de criptografia assimétrica com bastante aceitação[1]. O RSA é utilizado em vários protocolos, como *HyperText Transfer Protocol Secure* (HTTPS) que é responsável pelas conexões seguras através de navegadores web e em sistemas de autenticação, por exemplo o *Secure Shell* (SSH), para acesso remoto.

A criptografia vai além dos métodos de cifragem mencionados. Ela também fornece mecanismos de autenticação e assinatura digital. Paar et al. [14] em seu livro explicam que a autenticação é uma forma de confirmar que a informação vem de uma fonte legítima. Os autores também mencionam que as assinaturas digitais garantem a integridade das

[1] A sigla vem dos sobrenomes de seus criadores: Ronald Rivest, Adi Shamir e Leonard Adleman.

mensagens e que elas não sejam repudiadas no destino. Esses mecanismos são importantes para a confiabilidade das transações eletrônicas e para a proteção contra fraudes.

Além do JCA, foram desenvolvidas outras APIs de criptografia no intuito de reduzir a complexidade e erros comuns na utilização de algoritmos criptográficos. Um exemplo dessas APIs é o Google Tink, que é uma biblioteca projetada objetivando oferecer uma interface de alto nível para abstrair os detalhes de configuração que frequentemente levam a falhas de segurança. Como já citado, Nadi et al. [2] demonstram que, um dos desafios enfrentados pelos desenvolvedores ao utilizar APIs de criptografia em Java reside na dificuldade de compreensão da sequência correta de chamadas de métodos e parâmetros que são necessários para atingir uma implementação segura. O Google Tink surgiu com o objetivo de mitigar esse problema ao fornecer componentes criptográficos de forma padronizada para criptografia autenticada, assinatura digital e geração de chaves. Dessa forma, simplifica a utilização e reduz o risco de erros de configuração. Estudos acadêmicos, como Bonifácio et al. [15], analisam a variabilidade na especificação de uso incorreto de APIs e incluem o Google Tink em suas investigações, demonstrando sua abordagem para evitar erros comuns.

De maneira complementar, existem soluções como o WolfCrypt, uma biblioteca otimizada para desempenho, voltada para ambientes com restrições de recursos, como dispositivos embarcados. Ao contrário do Google Tink, que tem uma abordagem de alto nível, o WolfCrypt exige dos desenvolvedores um conhecimento mais profundo sobre criptografia, uma vez que essa biblioteca mantém um controle mais minucioso sobre os algoritmos e configurações. Conforme ressaltado por Nadi et al. [2], a complexidade das APIs criptográficas tradicionais pode vir a ser um obstáculo para a sua adoção adequada, resultando em implementações inseguras. Assim, embora bibliotecas como o Google Tink busquem simplificar a criptografia através de abstrações, outras como o WolfCrypt se concentram mais em situações onde o desempenho e a flexibilidade são fundamentais, demandando assim um maior cuidado na configuração de seus recursos [16].

Concluindo, a importância da criptografia fica clara na diversidade de funções que ela desempenha na segurança de informação e, portanto, na proteção de dados. Entretanto, é complexo e propenso a erros a implementação desses mecanismos, mesmo que automatizada. Para mitigar esse problema e facilitar a implementação da criptografia pelos desenvolvedores, surgiram as APIs de criptografia, como o caso do JCA. No entanto, como será discutido na próxima seção, embora forneça uma interface genérica e comum para as técnicas criptográficas, essas APIs nem sempre são simples de se utilizar ou bem documentadas, então é possível que diversas implementações ainda não sejam seguras o suficiente.

2.1.2 Funcionamento e importância das APIs de criptografia

À medida que a tecnologia evoluiu e as necessidades de proteção de informações sensíveis se tornaram mais evidentes, surgiram as APIs de criptografia em várias linguagens de programação para auxiliar os desenvolvedores na implementação de algoritmos de segurança. Essas APIs oferecem interfaces padronizadas que abstraem os detalhes internos da criptografia, permitindo, dessa forma, a aplicação de mecanismos de segurança aderente às boas práticas. Dessa forma, evitam a necessidade de reimplementar algoritmos criptográficos, reduzindo erros e facilitando a adoção de técnicas consolidadas para a proteção de dados.

No passado, os desenvolvedores precisavam implementar algoritmos de criptografia manualmente, o que poderia gerar erros e falhas de segurança. Para ajudar a simplificar e padronizar esse processo, as linguagens de programação começaram a disponibilizar bibliotecas específicas com interfaces que encapsulam algoritmos de criptografia bem usados, como AES, RSA, *Secure Hash Algorithm* (SHA) e etc. As APIs ajudam os desenvolvedores a integrar de forma mais fácil funcionalidades de segurança em seus programas, como criptografia de dados, geração de *hashing* e autenticação que garante a confidencialidade, integridade e a autenticidade das informações a serem transmitidas ou armazenadas.

Nesse contexto, segundo Ferguson et al. [17], a padronização das APIs de criptografia é essencial para garantir a segurança e a integração desses sistemas. Como mencionado antes, uma API fornece uma abstração sobre o uso de criptografia, permitindo que os desenvolvedores possam se concentrar na camada da lógica de negócio. Dessa maneira, o risco de algo dar errado em relação à segurança do dado é muito reduzido. Portanto, a padronização é uma necessidade.

Já de acordo com Gutmann [18], APIs de criptografia tornam os sistemas fáceis de atualizar e manter. Com algoritmos novos sendo desenvolvidos o tempo todo e vulnerabilidades em algoritmos existentes sendo encontradas, APIs podem ser atualizadas para suportar as mudanças sem precisar mudar o código de uma aplicação que esteja fazendo uso dela. Dessa forma, o sistema continuará protegido ao longo do tempo, apesar de novas ameaças sempre surgirem.

Também é importante notar que a criptografia em si desempenha um papel essencial na conformidade com a legislação e padrões de segurança, como o *General Data Protection Regulation* (GDPR) e o *Payment Card Industry Data Security Standard* (PCI DSS). Usando APIs que incluem algoritmos de segurança bem conhecidos e as melhores práticas, as organizações garantem que seus resultados estejam em conformidade específica com os regulamentos, evitando multas e mantendo a confiança dos clientes [19, 20].

Embora a padronização das APIs de criptografia tenha a sua contribuição para a segurança de sistemas reconhecida [17] e a sua utilização traga facilidades para a manu-

tenção ao longo do tempo [18], a sua utilização por si só não é garantia de uma correta implementação de mecanismos criptográficos. Estudos indicam que desenvolvedores com certa frequência enfrentam dificuldades na utilização dessas APIs em função de sua documentação complexa e à falta de diretrizes claras em relação à configuração de forma segura [2]. Essas dificuldades podem levar a escolhas indevidas, como o uso de algoritmos já obsoletos, a seleção de parâmetros inseguros ou a reutilização de chaves criptográficas, tendo a proteção dos dados comprometida. Além disso, estar em conformidade com padrões regulatórios não é garantia automática de uma implementação segura; boas práticas de desenvolvimento e auditorias periódicas de código são essenciais e devem continuar a serem praticadas a fim de evitar vulnerabilidades. Dessa forma, as APIs de criptografia têm a sua eficácia diretamente ligada não apenas à tecnologia utilizada, mas também na forma como empregadas e gerenciadas dentro de um sistema seguro.

Conforme observado, as APIs de criptografia são indispensáveis para o desenvolvimento de softwares seguros e trazem consigo ferramentas necessárias para a correta e eficaz implementação dos algoritmos de criptografia. Elas possibilitam que os desenvolvedores protejam dados sensíveis sem que precisem se tornar especialistas no domínio de criptografia, diminuindo assim a exposição a riscos e melhorando a segurança do software em geral. Na próxima seção, serão abordados os Usos Indevidos de APIs de Criptografia, analisando de que maneira é possível realizar implementações inseguras e, principalmente, como impedi-las de ocorrer.

2.2 Usos Indevidos de APIs de Criptografia

O uso inadequado de APIs de criptografia é um dos problemas mais comuns associados ao desenvolvimento de software que necessita de segurança para armazenar e/ou trafegar dados, e que frequentemente causa vulnerabilidades graves e compromete a segurança dos sistemas. A relativa complexidade das APIs de criptografia significa que até mesmo o desenvolvedor experiente pode cometer muitos erros ao implementar tecnologias com base nelas. Como citado anteriormente, de acordo com Nadi et al. [2], o uso de APIs de forma inadequada para criptografia é uma das ameaças de segurança mais difundidas em softwares que as utilizam, o que sugere a dificuldade de uso correto das interfaces pelos desenvolvedores. A reutilização de chaves, a escolha inadequada de algoritmos e a manipulação incorreta de dados criptográficos são comuns e levam a riscos de segurança relevantes. Esse entendimento é reforçado por Ochoa et al. [3], que sistematizaram diferentes categorias de usos nocivos de APIs, incluindo aqueles que comprometem diretamente a segurança de aplicações críticas.

Além disso, certas APIs são desenvolvidas de maneira excessivamente flexível, o que possibilita configurações que não são seguras por padrão. O estudo realizado por Nadi et al. [2] constatou que diversas bibliotecas de criptografia apresentam configurações padrões vulneráveis ou exigem expertise para utilização adequada. Essa maleabilidade, apesar de ser vantajosa para especialistas, pode causar confusão em desenvolvedores menos experientes em segurança da informação, resultando em implementações inseguras. A falta de instruções claras e exemplos práticos do uso adequado piora ainda mais a situação. Resultados semelhantes foram sintetizados por Ochoa et al. [3], que apontam configurações inseguras como uma fonte recorrente de vulnerabilidades e como um desafio persistente na prática de desenvolvimento seguro.

Frente a essa situação, é fundamental utilizar ferramentas que possibilitem aos desenvolvedores identificar e solucionar vulnerabilidades de modo preventivo, evitando assim impactos na segurança. A análise estática de código é crucial, permitindo encontrar vulnerabilidades no código sem precisar executar o programa.

O CogniCrypt, apresentado por Krüger et al. [21], e o CryptoGuard, introduzido por Rahaman et al. [22], são exemplos de ferramentas específicas para lidar com esses desafios. O CogniCrypt auxilia desenvolvedores no uso correto das APIs de criptografia desde os estágios iniciais de desenvolvimento do software, ajudando a identificar usos incorretos e recomendando boas práticas. Já o CryptoGuard faz o uso de técnicas avançadas de análise estática para identificar automaticamente usos indevidos dessas APIs em sistemas desenvolvidos na linguagem Java, evitando falhas criptográficas antes de serem implementadas. As duas ferramentas destacam a relevância da análise estática como uma estratégia proativa para a detecção de vulnerabilidades, possibilitando assim a resolução de problemas antes que possam impactar ambientes de sistemas em produção.

A análise estática de código é fundamental para identificar vulnerabilidades e mitigar falhas de segurança de sistemas durante a fase de desenvolvimento, sem precisar executar o software. Dentre as ferramentas que ajudam nesse procedimento, o CogniCrypt e o CryptoGuard se sobressaem por se concentrarem especialmente na identificação de utilizações indevidas de APIs criptográficas em Java, um ponto importante da proteção de programas de computador.

Como exemplo de uso indevido de APIs de criptografia que pode ser detectado por essas ferramentas, o trecho de código 2.1 possui uma vulnerabilidade associada ao uso inadequado de API criptográfica. Isso ocorre em razão da implementação padrão do AES sem mencionar de maneira explícita o modo de operação. Neste cenário, o modo padrão adotado é o *Electronic Codebook* (ECB), amplamente reconhecido como inseguro, uma vez que ciframentos idênticos geram textos cifrados idênticos, facilitando, dessa forma, ataques por análise de padrões. Assim, ausência de uma definição explícita de um modo

seguro, como *Cipher Block Chaining* (CBC) ou *Galois/Counter Mode* (GCM) torna o sistema vulnerável a ataques criptográficos, comprometendo assim a confidencialidade dos dados protegidos.

```
1         SecretKey secretKey = // ...
2         Cipher cipher = Cipher.getInstance("AES");
3         cipher.init(Cipher.ENCRYPT_MODE, secretKey);
4         cipher.doFinal(inputMsg);
```

Código 2.1: Uso indevido de API de criptografia. (Fonte: [2]).

Embora o trecho de código 2.1 ilustre um caso específico de configuração insegura, a compreensão profunda dos riscos criptográficos exige uma análise mais detalhada dos mecanismos envolvidos. A seção a seguir explora esses aspectos técnicos, detalhando como falhas nos modos de operação, na geração de aleatoriedade e nos protocolos de transporte comprometem a segurança dos sistemas.

2.3 Detalhamento das Vulnerabilidades Criptográficas

Para compreender a gravidade dos usos indevidos citados anteriormente, é fundamental detalhar o funcionamento técnico de algumas das principais vulnerabilidades encontradas em implementações criptográficas, bem como os conceitos teóricos que justificam as correções de segurança.

2.3.1 Modos de Operação de Cifra de Bloco

O uso de algoritmos de cifra de bloco, como o AES, requer a definição de um modo de operação. O padrão em muitas implementações antigas é o ECB, que é reconhecidamente inseguro para a grande maioria dos cenários, pois realiza o processamento de cada bloco de dados de forma independente. Essa característica faz com que blocos de texto plano idênticos gerem blocos de texto cifrado idênticos, revelando padrões nos dados originais [23].

A Figura 2.1 demonstra a fragilidade desse modo de operação. É possível ver a imagem original à esquerda e, ao centro, o resultado da cifragem utilizando tal algoritmo. Nessa imagem central, percebe-se a presença de padrões que permitem decifrar o conteúdo, exibindo os contornos do desenho original.

Figura 2.1: Imagem cifrada com o modo ECB. (Fonte: Adaptado de [24])

Em contrapartida, modos como o *Counter Mode* (CTR) transformam um valor de contador em um fluxo de chave e realizam a combinação dessa chave com o dado através de uma operação *exclusive OR* (XOR). Isso garante que blocos iguais de entrada não gerem blocos iguais na saída, evitando a quebra por análise de padrões e permitindo paralelismo no processamento [23]. A imagem à direita na Figura 2.1 demonstra o resultado visualmente seguro do uso de um modo adequado.

2.3.2 Derivação Segura de Chaves

A geração de chaves criptográficas a partir de senhas exige cuidados específicos para evitar ataques de força bruta ou dicionário. O uso de algoritmos como o `PBKDF2WithHmacSHA256` é amplamente recomendado. Esse algoritmo aplica uma função *hash* (como SHA-256) repetidamente sobre a senha combinada com um *salt* (valor aleatório) [17]. O número de iterações e o uso do *salt* aumentam exponencialmente o custo computacional para um atacante tentar descobrir a chave original [4]. Em Java, chaves secretas para algoritmos simétricos são frequentemente encapsuladas na classe `SecretKeySpec`, que adapta a chave bruta para ser consumida pelos cifradores [14].

2.3.3 Geradores de Números Aleatórios (PRNG)

A segurança de chaves, *salts* e vetores de inicialização (Inicialização de Vetores (IV)) depende da imprevisibilidade dos números gerados. Classes comuns como `java.util.Random` utilizam geradores pseudoaleatórios lineares que são determinísticos e previsíveis se o estado interno for descoberto. Para fins criptográficos, é imprescindível o uso de geradores criptograficamente seguros (*Cryptographically Secure Pseudo-Random Number Generator* (Gerador de Números Pseudoaleatórios Criptograficamente Seguro – CSPRNG)), como o `java.security.SecureRandom`. Uma implementação comum é o algoritmo `SHA1PRNG`,

que baseia sua aleatoriedade em funções hash criptográficas, garantindo aleatoriedade suficiente para impedir a previsão de valores futuros [12, 4].

2.3.4 Segurança em Protocolos de Transporte (SSL/TLS)

A segurança na camada de transporte depende de configurações rigorosas para evitar interceptações. O processo de conexão segura inicia-se com o *Handshake*, etapa onde cliente e servidor negociam a versão do protocolo (ex: TLS 1.2) e as *Ciphersuites*. As *Ciphersuites* são conjuntos de algoritmos que definem como será realizada a troca de chaves, a autenticação e a cifragem dos dados durante a sessão [17].

Configurações incorretas nesta etapa podem expor a comunicação a diversas vulnerabilidades:

- **Protocolos Obsoletos:** O uso de *Secure Sockets Layer* (SSL)v3 ou versões antigas permite ataques de *downgrade* (ou *version-rollback*), forçando a comunicação a usar algoritmos fracos [9].
- **TrustManagers Permissivos:** Implementações que não validam a cadeia de certificação ou a validade do certificado aceitam qualquer identidade fornecida pelo servidor.
- **HostnameVerifier Inseguro:** A falta de verificação do *Common Name* (CN) ou *Subject Alternative Name* (SAN) permite que um certificado válido para o site "A" seja usado para interceptar tráfego destinado ao site "B" [25].

Essas falhas facilitam ataques de *Man-in-the-Middle* (MITM), um cenário onde um atacante captura e modifica a comunicação entre duas partes, levando-as a acreditar que estão conectadas diretamente uma à outra, comprometendo a confidencialidade e a integridade dos dados [4, 25].

Diante da complexidade técnica dessas vulnerabilidades, a identificação manual torna-se propensa a falhas. Nesse contexto, Acar et al. [26] ressaltam que a utilização de métodos de análise estática de código pode auxiliar na detecção e resolução de falhas na utilização de APIs de criptografia, antes da implantação do software em ambiente de produção. Ferramentas como CogniCrypt e o CryptoGuard são citadas como casos de tecnologias que ajudam na identificação automatizada de padrões inseguros no código, aumentando a segurança dos sistemas. Portanto, a verificação automática de código é uma estratégia eficaz para reduzir erros e assegurar implementações criptográficas mais seguras.

Esse cenário evidencia a relevância da análise estática como uma abordagem proativa na detecção de vulnerabilidades, permitindo que desenvolvedores identifiquem problemas relacionados ao uso de APIs de criptografia ainda durante o ciclo de desenvolvimento.

Além de fortalecer a segurança do software, essa prática também contribui para uma maior compreensão das armadilhas comuns em implementações criptográficas. Dessa forma, ao aprofundar-se nos fundamentos e aplicações da análise estática de código, é possível avaliar como essa técnica se integra a estratégias mais amplas de segurança no desenvolvimento de sistemas.

2.4 Análise Estática de Código

A segurança do software é um tópico relevante nos sistemas modernos, especialmente no que tange ao uso correto de API de criptografia. Nesse contexto, a análise estática do código constitui uma técnica útil, uma vez que permite identificar as vulnerabilidades do software sem a necessidade de o executar. É importante considerar que algumas ferramentas, como o CogniCrypt e o CryptoGuard, foram projetadas especificamente para a descoberta de usos inseguros de criptografia em Java, possibilitando a sua correção.

A análise estática consiste na inspeção do código-fonte ou mediante a verificação do *bytecode* de uma aplicação para detectar desvio de qualidade, vulnerabilidades ou problemas de segurança em potencial. É realizada por ferramentas automatizadas que analisam o fluxo de dados e o uso de certas bibliotecas ou APIs, garantindo as práticas ideais de programação. Além da detecção de falhas de segurança como acessos à memória inválidos ou usos incorretos de APIs de criptografia, a análise estática de código também ajuda a melhorar a qualidade do código de forma geral.

É possível verificar que uma das vantagens da análise estática é a possibilidade de realizá-la em tempo de desenvolvimento, possibilitando que as vulnerabilidades sejam sanadas antes mesmo de o sistema entrar em fase de testes ou colocá-lo em produção. Também, a análise pode ser feita repetidamente à medida que o código é alterado, mantendo a integridade do sistema em todo o seu ciclo de vida.

A utilização de análise estática tem sido recomendado de forma ampla por organizações como o *Open Web Application Security Project* (OWASP) e o *National Institute of Standards and Tecnology* (NIST). A técnica de análise estática permite identificar problemas difíceis de detectar em etapas posteriores, como a execução de testes de penetração, tornando-se uma estratégia adequada para um *pipeline* de segurança *Development, Security, and Operations* (Desevolvimento, Segurança e Operações – DevSecOps) [27, 28].

Após a relevância e os benefícios da análise estática de código para segurança de software terem sido discutidos, serão apresentadas algumas ferramentas específicas que operacionalizam essa técnica. Este trabalho considera particularmente as ferramentas CogniCrypt [21] e CryptoGuard [22], escolhidas por adotarem abordagens distintas para a detecção de usos incorretos de APIs de criptografia na linguagem de programação Java.

O CogniCrypt utiliza o componente CryptoAnalysis para realizar a análise estática tendo como base regras definidas na linguagem *Cryptographic Specification Language* (Linguagem de Especificação Criptográfica – CrySL). O CryptoGuard, de forma semelhante, também utiliza regras pré-definidas, aplicadas diretamente sobre o código-fonte através de uma análise estática leve (*lightweight*) e rápida, com o objetivo de identificar padrões conhecidos de uso indevido de APIs de criptografia [22].

2.4.1 CogniCrypt

O CogniCrypt foi oficialmente apresentado em 2017 por um grupo de pesquisadores das universidades *Technische Universität Darmstadt* (TUD), *Paderborn University* (PU) e *University of Alberta* (UA) [21]. O CogniCrypt foi criado em resposta a um desafio enfrentado por muitos desenvolvedores ao utilizar APIs de criptografia, especialmente diante de um risco elevado de comprometimento da segurança dos sistemas devido a erros no uso dessas APIs [21]. A ferramenta se propõe a facilitar o uso apropriado da criptografia, auxiliando o desenvolvedor na utilização correta das APIs, mesmo sem conhecimento prévio sobre o assunto.

2.4.1.1 CogniCrypt: Análise Estática e Suporte ao Desenvolvedor

Em sua essência, o CogniCrypt é uma ferramenta de análise estática que identifica usos incorretos de APIs de criptografia em código Java. Ele atua como um *plugin* da *Integrated Development Environment* (Ambiente de Desenvolvimento Integrado – IDE) Eclipse que fornece *feedback* em tempo real para o programador enquanto ele escreve o código. Ao detectar alguma forma de uso incorreto de uma API criptográfica, a ferramenta sugere uma opção de código recomendada para sanar o problema em potencial [21, 29]. Dessa maneira, a combinação de análise estática com o suporte ao desenvolvimento facilita a utilização de APIs de criptografia de forma correta, o que reduz os seus riscos no âmbito da segurança.

Contudo, considerando que os softwares analisados neste estudo já estão finalizados e em produção, verificou-se que o uso do CogniCrypt via linha de comando é mais apropriado do que a sua utilização na IDE Eclipse. Dessa forma, optou-se por essa abordagem para realizar a análise estática de código dos softwares Java selecionados e obter relatórios sobre os usos indevidos de API de criptografia. Essa escolha eliminou a necessidade de suporte em tempo real na IDE Eclipse e permitiu a execução independente de outros programas.

2.4.1.2 CogniCryptSAST: Expansão da Análise Estática no CogniCrypt

O CogniCryptSAST é um componente avançado de análise estática que substituiu o módulo de análise original do CogniCrypt, sendo desenvolvido para ampliar sua capacidade de verificar usos incorretos de APIs de criptografia. Apresentado como parte de uma evolução do projeto CogniCrypt, o CogniCryptSAST foi projetado para resolver limitações observadas em abordagens anteriores de análise estática, como a dependência do ambiente de desenvolvimento Eclipse e o foco restrito em análises em tempo real. Ele utiliza o CrySL como base para especificar formalmente as regras de uso seguro de APIs criptográficas, permitindo, segundo seus autores, detectar violações com maior precisão e flexibilidade [29].

Diferentemente do CogniCrypt, que é fortemente integrado ao Eclipse, o CogniCryptSAST pode ser executado de forma independente via via linha de comando [29]. Essa característica elimina a necessidade de suporte em tempo real dentro de um IDE, permitindo sua aplicação em auditorias de segurança e verificação de conformidade, especialmente em sistemas legados ou projetos em produção. Essa abordagem facilita sua integração em *pipelines* de DevSecOps ou ferramentas de análise contínua, ampliando seu uso em processos de desenvolvimento modernos.

O trecho de código 2.2 ilustra uma execução do CogniCrypt via linha de comando para a realização de análise estática no Componente 1, um dos softwares selecionados para testes nesta pesquisa. Neste caso optou-se por utilizar o formato *JavaScript Object Notation* (JSON) para o relatório de saída ao informar o valor *Static Analysis Results Interchange Format* (SARIF) no parâmetro `--reportFormat`. Existe também a opção de saída do relatório em formato *Comma-Separated Values* (CSV). Nesse caso, o valor do parâmetro `--reportFormat` passado seria CSV.

```
$ java -jar HeadlessJavaScanner-4.0.1-jar-with-dependencies.jar --rulesDir
↪ Crypto-API-Rules-3.1.2-jca/JavaCryptographicArchitecture/src --appPath
↪ componente1.jar --reportPath . --reportFormat "SARIF"
```

Código 2.2: Exemplo de execução do CogniCrypt. (Fonte: Própria)

A ferramenta opera avaliando o código-fonte com base em regras definidas em CrySL, as quais especificam o uso correto das classes e métodos de APIs de criptografia. Essas regras incluem restrições de parâmetros, sequências de chamadas de métodos e requisitos para objetos. Por exemplo, o CogniCryptSAST pode identificar problemas como o uso de algoritmos inseguros (como *Data Encryption Standard* (DES)) ou a ausência de inicialização adequada de parâmetros criptográficos. Ao encontrar uma violação, a ferramenta

gera relatórios detalhados, que destacam os problemas identificados e oferecem subsídios para sua resolução [29].

O trecho de código 2.3 demonstra um relatório do CogniCrypt, resultado de sua execução também no Componente 1. Na imagem, é possível ver o uso indevido de API de criptografia no código ao usar o modo de operação ECB no algoritmo de criptografia AES. Como mencionado anteriormente, trata-se de um modo já comprovado como inseguro, sendo vulnerável a ataques por análise de padrões, ataques de rearranjo ou ataques de injeção de dados.

```
{
  "sarifVersion": "2.0.0",
  "runs": [{
    "results": [{
      "locations": [{
        "physicalLocation": {
          "fileLocation": { "uri":
          ↪  "org/instituicao/componente1/seguranca/criptografia/BaseCripto.java" }
        },
        "fullyQualifiedLogicalName":
        ↪  "org::instituicao::componente1::seguranca::criptografia::BaseCripto::encrypt"
      }],
      "ruleId": "ConstraintError",
      "message": {
        "text": "First parameter (with value \"AES/ECB/PKCS5Padding\") should be any of AES/{CBC, GCM,
        ↪  PCBC, CTR, CTS, CFB, OFB}.",
        "richText": "ConstraintError violating CrySL rule for javax.crypto.Cipher."
      }
    }]
  }]
}
```

Código 2.3: Exemplo de relatório do CogniCrypt. (Fonte: Própria)

Krüger et al. discute no artigo "*CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs*", que o CogniCryptSAST é altamente preciso na detecção de violações de segurança em grande escala. Quando aplicado a mais de 200.000 artefatos Java da biblioteca do repositório Maven Central, a ferramenta detectou pelo menos uma violação de segurança em 63% dos artefatos. Além disso, considerando 10.000 aplicações Android analisadas, a ferramenta identificou problemas em 95% dos casos. Isso sugere que a ferramenta é capaz de detectar vulnerabilidades que outras deixam passar despercebido. Toda essa precisão, segundo o artigo, está diretamente ligada à sua rigorosa abordagem ao utilizar listas brancas, onde apenas formas de uso seguras e explicitas de APIs de criptografia são permitidas [29].

2.4.1.3 CogniCryptGen: Geração Segura de Código Criptográfico

Como parte do CogniCrypt, foi apresentado o CogniCryptGen - um componente para geração segura de código criptográfico [29, 30]. Essa ferramenta foi desenvolvida para

ampliar o suporte oferecido aos desenvolvedores, indo além da detecção de usos indevidos de APIs de criptografia em código já existente, ajudando os desenvolvedores a criar software criptográfico seguro a partir do início. Para alcançar esse objetivo, o CogniCryptGen usa modelos de código, que são uma espécie de estrutura sobre o qual o código-fonte dos softwares é baseado, e regras formais especificadas na linguagem CrySL, que garantem a segurança do código gerado.

Com o CogniCryptGen, o desenvolvedor inicia o processo ao selecionar o tipo de tarefa criptográfica desejada - seja ela a cifragem de arquivos ou a gestão segura de senhas - utilizando um dos modelos já preparados oferecidos pela ferramenta. Com essa escolha, o CogniCryptGen gera automaticamente o código necessário, combinando os *templates* com as regras CrySL, que asseguram que as melhores práticas de segurança sejam seguidas. De acordo com os autores, essa automação na geração de código facilita a implementação, reduz significativamente os erros e isenta o desenvolvedor das complexidades inerentes ao manuseio de APIs de criptografia, ao mesmo tempo que garante a aplicação dos algoritmos apropriados.

O objetivo do CogniCryptGen é simplificar o processo de desenvolvimento, permitindo que desenvolvedores criem código criptograficamente correto sem a necessidade de conhecimentos profundos sobre cada API. O CogniCryptGen, então, permite a geração de código seguro para tarefas criptográficas comuns de forma automática, reduzindo a curva de aprendizado para o desenvolvedor.

2.4.1.4 CrySL: A Linguagem de Especificação de Segurança do CogniCrypt

A base para análise estática do CogniCrypt e para a geração de código seguro do CogniCryptGen é a linguagem CrySL [29]. A linguagem CrySL permite especificar formalmente como as APIs criptográficas devem ser usadas de maneira segura. Isso significa que a linguagem permite definir como métodos, classes e parâmetros de uma API criptográfica devem ser chamados para garantir que não haja nenhum problema de segurança. Como destacado por Krüger et al. [29], com a CrySL, é possível definir, de forma clara e objetiva, as sequências de chamadas de método, as restrições de parâmetros, e as relações entre objetos que são relevantes para o uso seguro de APIs criptográficas.

A linguagem CrySL também permite definir as garantias e requerimentos de uso, que são utilizados pelo CogniCrypt para identificar usos incorretos ou gerar código adequado. Por exemplo, uma regra CrySL pode especificar que o método `Cipher.init()` deve ser chamado somente após a inicialização do objeto da classe `SecureRandom`. Ou que para a configuração de um objeto da classe `PBEKeySpec` é preciso a utilização de um salt aleatório. Em resumo, a CrySL funciona como um contrato de uso para as APIs de criptografia.

O trecho de código 2.4 ilustra uma regra CrySL, no caso a `SecretKey.crysl`.

```
SPEC javax.crypto.SecretKey

OBJECTS
        byte[] keyMaterial;

EVENTS
        ge1: keyMaterial = getEncoded();
        GetEnc := ge1;

        d1: destroy();
        Destroy := d1;

ORDER
        GetEnc*, Destroy?

REQUIRES
        generatedKey[this, _];

ENSURES
        preparedKeyMaterial[keyMaterial] after GetEnc;

NEGATES
        generatedKey[this, _] after Destroy;
```

Código 2.4: Regra CrySL para SecretKey. (Fonte: [31])

A linguagem CrySL serve como um elo entre os especialistas em criptografia, que desenvolvem as regras, e os desenvolvedores que precisam utilizar APIs de criptografia corretamente. Com a CrySL, é possível definir e validar formalmente o uso das APIs de criptografia, ao mesmo tempo que possibilita a geração de código seguro pelo CogniCryptGen. As regras definidas em CrySL são utilizadas tanto para verificar o uso correto de APIs, na análise estática do CogniCrypt, quanto para gerar código seguro pelo CogniCryptGen.

2.4.1.5 Os Resultados e o Impacto do CogniCrypt

O ecossistema CogniCrypt, mais precisamente a sua ferramenta de análise estática, CogniCryptSAST tem, desde a sua introdução, exibido resultados promissores na prevenção de vulnerabilidades, ajudando os desenvolvedores a utilizar de maneira segura as APIs criptográficas [21, 29]. De acordo com estudos empíricos, o CogniCryptSAST oferece boa precisão na detecção de problemas de segurança crítica. Isso inclui erros como o uso de algoritmos inseguros, por exemplo, DES ou ECB [21, 29]. O CogniCrypt, conforme os autores, mostrou-se confiável e eficaz tanto ajudando desenvolvedores a gerar código seguro com o CogniCryptGen, quanto detectar deficiência em códigos já implementados [30].

Como observado, a evolução do CogniCrypt pode resumir-se a sua arquitetura extensível com base na linguagem CrySL e capacidade de aplicar a análise e a geração de código em diferentes contextos de desenvolvimento de software [29, 30]. O CogniCrypt tem se mostrado uma ferramenta útil na detecção de vulnerabilidades e no apoio aos desenvol-

vedores para seguirem boas práticas ao utilizar APIs de criptografia [29, 30]. Por fim, as regras CrySL são uma base consistente para a análise e geração de código voltado à segurança, promovendo a conformidade com as diretrizes de segurança recomendadas [29]. O projeto CogniCrypt também apresentou bons resultados na detecção de vulnerabilidades criptográficas, mantendo a taxa de falsos positivos relativamente controlada em comparação com outras ferramentas de análise estática [21, 29].

Além do CogniCrypt, outras ferramentas de análise estática têm sido desenvolvidas para abordar desafios similares na segurança de software. Uma dessas ferramentas é o CryptoGuard, que oferece uma abordagem diferente para resolver problemas de implementação criptográfica.

2.4.2 CryptoGuard

O desenvolvimento do CryptoGuard começou como resposta às crescentes preocupações em torno da segurança no uso de criptografia em aplicações Java. Desenvolvido por Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, dentre outros pesquisadores da Virginia Tech e da Universidade do Texas em Dallas, uma de suas apresentações aconteceu em 2019, durante a Conferência de Segurança e Comunicações da *Association for Computing Machinery* (ACM) [22]. O CryptoGuard foi concebido a partir da demanda não atendida por uma ferramenta que pudesse detectar corretamente vulnerabilidades na utilização de APIs de criptografia em grandes bases de código, de modo a auxiliar desenvolvedores a evitar erros comuns no uso de APIs de criptografia. Além disso, sua criação representa um esforço contínuo no desenvolvimento de técnicas de análise estática que se mostrem cada vez mais precisas e menos custosas em relação à taxa de falsos positivos.

A ferramenta CryptoGuard consegue detectar várias formas de problemas associadas à má utilização das APIs criptográficas do Java, como geração de chaves criptográficas fracas, uso de algoritmos de criptografia mais antigos ou inseguros, reutilização de IV em um processo criptográfico, entre outros.

O CryptoGuard tem como característica importante, como apontam os autores em [22], a capacidade de realizar uma análise de fluxo de dados interprocedural e sensível ao contexto. Isso quer dizer que ele tem a capacidade de realizar o rastreamento dos dados para verificar como eles são manipulados em diferentes partes do código, até mesmo através de chamadas de métodos. Além disso, o CryptoGuard, ainda de acordo com Rahaman et al. [22], faz uso de técnicas avançadas no sentido de analisar fluxos de controle, possibilitando que as suas detecções sejam robustas, evitando sempre os resultados considerados falsos positivos.

Outra característica importante do CryptoGuard é que ele foi desenvolvido para a análise de grandes projetos de software. Por ser capaz de lidar com volumes expressivos

de código e de detectar vulnerabilidades com um bom nível de acerto, considera-se que o CryptoGuard pode ser integrado com relativa facilidade a sistemas já em produção. Seu desempenho na detecção de vulnerabilidades criptográficas, conforme [22], torna o CryptoGuard uma ferramenta útil para instituições que utilizam ou pretendem utilizar APIs de criptografia em seus dados críticos e/ou sensíveis e que buscam manter suas implementações em nível de segurança minimamente adequado.

O trecho de código 2.5 ilustra uma execução do CryptoGuard via linha de comando para a realização de análise estática no Componente 1, um dos softwares selecionados para testes neste trabalho.

```
$ java -jar cryptoguard.jar -in jar -s ../projects/jars/componente1.jar
```

Código 2.5: Exemplo de execução do CryptoGuard. (Fonte: Própria)

Por fim, o CryptoGuard auxilia as equipes de desenvolvimento ao fornecer relatórios com dados detalhados dos usos indevidos de APIs de criptografia, categorizando esses tipos de vulnerabilidades e indicando onde se localizam no código do software. Essa característica facilita o trabalho das equipes, uma vez que os programadores podem concentrar seus esforços nas atividades de correção das falhas de segurança, demandando menos tempo na etapa de identificação. Dessa forma, é possível observar um ganho na produtividade.

O trecho de código 2.6 ilustra um relatório do CryptoGuard referente à sua execução no Componente 1. Na imagem é possível ver um uso indevido de API de criptografia no código ao usar o modo de operação ECB no algoritmo de criptografia AES. Esse modo de operação não é mais recomendado, uma vez que já foi comprovado como inseguro por ser vulnerável a ataques por análise de padrões, ataques de rearranjo ou ataques de injeção de dados.

```
{
  "projectName": "CryptoGuard",
  "Issues": [{
    "Message": "Found: \"AES/ECB/PKCS5Padding\"",
    "RuleDesc": "Found broken crypto schemes",
    "CWEId": 256,
    "Severity": "1",
    "_FullPath": "componente1.jar/org/instituicao/componente1/seguranca/criptografia/BaseCripto.class"
  }]
}
```

Código 2.6: Exemplo de relatório do CryptoGuard. (Fonte: Própria)

Conforme o campo de análise estática relacionada à segurança em APIs criptográficas evolui, a comparação entre diferentes ferramentas torna-se relevante para compreender

suas vantagens, limitações e contribuições específicas. Nesse sentido, a subseção a seguir apresenta uma análise comparativa entre o CogniCrypt e o CryptoGuard, apontando suas respectivas abordagens e resultados, bem como as contribuições que surgem desse confronto de ideias e soluções.

2.4.3 Comparação e Contribuições

Tanto o CogniCrypt quanto o CryptoGuard têm como objetivo a segurança no uso de criptografia. Porém eles têm uma abordagem distinta. O CryptoGuard, como é possível observar em seu artigo de apresetação [22], tem tem como foco a detecção precisa de vulnerabilidades através de técnicas avançadas de análise estática, visando a identificação de problemas tais como a reutilização de chaves, algoritmos obsoletos e a falta de verificação de integridade. Essas análises são baseadas em *slicing*, e análise de fluxo de dados para detectar vulnerabilidades semânticas.

Já o CogniCrypt, em especial o componente CogniCryptGen, como afirmam os autores, vai além da detecção de problemas, fornecendo suporte ao desenvolvedor nas ações de correção e também na prevenção dessas falhas. Isso é possível com o auxílio da linguagem CrySL, que garante que o código gerado seja seguro e esteja de acordo com o especificado. Quando utilizado integrado à IDE Eclipse, o CogniCryptGen atua de forma proativa, fornecendo alertas ao desenvolvedor durante a codificação do software e oferecendo sugestões de correção [21, 29, 30].

Ao mesmo tempo, o CogniCrypt utiliza o CogniCryptSAST para garantir que o código, mesmo com outras dependências, seja analisado e seguro. O CryptoGuard torna-se uma escolha acertada em auditorias de segurança em código legado ou em sistemas já em produção. O CogniCrypt também é uma excelente escolha, podendo ser utilizado tanto em ações de prevenção de usos indevidos de criptografia quanto em auditorias de segurança, com o diferencial de ter a linguagem CrySL como base para garantir a segurança do código [30].

De forma prática, o CryptoGuard é mais utilizado em auditorias de segurança de softwares em corporações, pois foca na descoberta de problemas complexos e na verificação da conformidade de softwares já existentes. Enquanto isso, o CogniCrypt, com o seu componente CogniCryptGen, torna-se um aliado do desenvolvedor ao ser utilizado de forma contínua durante a construção do software, gerando código seguro de acordo com regras predefinidas em CrySL e auxiliando o desenvolvedor no uso correto de APIs.

No entanto, nada impede que se possa utilizar as duas ferramentas de forma combinada, utilizando o CogniCrypt durante o desenvolvimento de software e o CryptoGuard para uma análise de código mais detalhada antes de liberar a versão final do software. Os desenvolvedores que desejam integrar segurança em seus softwares, devem analisar

as necessidades do seu projeto e escolher a ferramenta apropriada, sendo que todas elas oferecem contribuições válidas.

Capítulo 3

Caracterização do Estudo de Caso

3.1 Objetivos, Questões e Métricas

Conforme apresentado na introdução, esta pesquisa envolve a condução de um estudo de caso que tem como principal **objetivo** identificar e corrigir usos incorretos de APIs de criptografia em um subconjunto de sistemas, *webservices* e componentes reutilizáveis corporativos da Embrapa e desenvolvidos na linguagem de programação Java.

Este estudo foi motivado pela observação da crescente necessidade de garantia de segurança e a integridade dos dados em aplicações, tanto modernas quanto legadas, onde o papel da criptografia se mostra fundamental para a proteção contra ataques cibernéticos e acessos não autorizados. Além disso, sob a perspectiva acadêmica, existe uma lacuna na literatura relacionada à condução de estudos empíricos em colaboração com a indústria e que têm, como principal objetivo, a identificação de usos incorretos de APIs de criptografia usando ferramentas de análise estática como o CogniCrypt [30] e o CryptoGuard [22].

Para atingir esse objetivo, formulou-se quatro **questões de pesquisa** que, ao serem respondidas, permitirão uma compreensão mais aprofundada sobre a segurança no uso de APIs de criptografia na Embrapa, bem como possibilitará uma discussão sobre o uso das ferramentas CogniCrypt e CryptoGuard na avaliação da segurança em sistemas reais além de viabilizar uma análise do impacto para corrigir usos indevidos de APIs de criptografia. As questões de pesquisa são:

(QP1) Qual a prevalência de vulnerabilidades em sistemas corporativos da Embrapa?

(QP2) Como as equipes de desenvolvimento da Embrapa percebem os *warnings* sobre uso incorreto de APIs de criptografia reportados por ferramentas como Cognicrypt e CryptoGuard?

(QP3) Qual o impacto da correção dessas vulnerabilidades na estabilidade funcional e na estrutura de código nos sistemas corporativos da Embrapa?

(QP4) Quão efetivos são os usos de SLMs ou LLMs para as correções das vulnerabilidades identificadas?

A resposta a essas questões possibilita uma avaliação detalhada dos pontos fracos e fortes nas práticas de desenvolvimento observadas, auxiliando na proposição de estratégias para melhorar a segurança dos sistemas.

Foram utilizadas as duas ferramentas de análise estáticas CognicyCrypt e CryptoGuard para a análise do código de oito sistemas, sete *webservices* e dois componentes reutilizáveis da Embrapa, verificando o código de forma completa.

Considerando as **métricas** utilizadas nos estudos, as ferramentas de análise estática foram usadas para estimar a **prevalência** de vulnerabilidades em sistemas e apoiar na resposta da primeira questão de pesquisa (QP1). Tal prevalência correspondeu à quantidade de *warnings* gerados pelas ferramentas; onde um *warning* representou um problema de segurança iminente com relação a um uso indevido de criptografia.

Para responder à terceira questão de pesquisa (QP3), o impacto da correção foi avaliado sob duas dimensões complementares: a estabilidade funcional e o esforço de alteração no código. A estabilidade foi verificada por meio da execução de testes funcionais de homologação pelos usuários, assegurando que as funcionalidades dos sistemas não fossem comprometidas. Já o impacto estrutural foi avaliado por meio da métrica de **Volume de Alterações no Código-Fonte (Code Churn)**. Essa estratégia, posteriormente também aplicada às correções geradas por SLMs e LLMs, possibilita realizar a medição do esforço de desenvolvimento necessário para cada correção de vulnerabilidade, permitindo uma análise perceptível do impacto das correções no código-fonte.

Para responder à quarta questão de pesquisa (QP4), foram utilizados modelos de linguagem SLMs e LLMs para gerar propostas de correção às vulnerabilidades identificadas pelas ferramentas de análise estática. Em seguida, as correções sugeridas foram avaliadas quanto à sua efetividade por meio de duas métricas de percentual: a correção aparente, que analisa a eliminação dos *warning* nas ferramentas *Static Application Security Testing* (Teste de Segurança de Aplicação Estático – SAST), e a correção funcional, que confirma se o código corrigido permanece operacional e livre de erros de compilação. Essa avaliação em dois níveis permitiu identificar os benefícios e desafios desse método em comparação com as correções manuais, focando em compreender se os SLMs e os LLMs poderiam ser usados de forma eficaz para realizar as mesmas correções que foram feitas manualmente.

As métricas de prevalência, volume de alterações no código-fonte e os percentuais de correção são de grande importância para este trabalho. A prevalência oferece uma visão quantificada dos erros detectados, validando o uso das ferramentas CogniCrypt e CryptoGuard. O volume de alterações no código-fonte serve como um indicador concreto do esforço de desenvolvimento, tanto nas correções manuais quanto nas correções produzidas

por SLMs e LLMs. Finalmente, a análise dos percentuais de correção aparente e correção funcional possibilita uma avaliação aprofundada da real capacidade dos modelos de linguagem, diferenciando a mera conformidade sintática de uma solução verdadeiramente eficaz e aplicável. Essa distinção é de grande relevância para determinar a efetividade prática da abordagem semiautomatizada investigada neste estudo.

A segunda questão de pesquisa (QP2) é qualitativa em sua essência. Para respondê-la, foi conduzido um grupo focal cujas discussões foram examinadas por meio de uma análise baseada em tópicos, organizada de forma temática, a partir das falas dos participantes. Esse procedimento possibilitou identificar percepções recorrentes, divergências e sugestões sobre a utilização das ferramentas SAST, permitindo estruturar os dados qualitativos em categorias analíticas relevantes para a pesquisa [32].

3.2 Contexto da Pesquisa: Embrapa

Para situar adequadamente o contexto em que este projeto foi realizado, é essencial entender a estrutura e as características da Embrapa, uma entidade que exerce função estratégica no avanço tecnológico do agronegócio do Brasil. A compreensão do ambiente organizacional e tecnológico da Embrapa é fundamental para avaliar o impacto e a importância das questões de segurança discutidas nesta pesquisa.

A Empresa Brasileira de Pesquisa Agropecuária (Embrapa) é uma empresa pública, vinculada ao Ministério da Agricultura e Pecuária (MAPA), que foi criada em 1973 para desenvolver a base tecnológica de um modelo de agricultura e pecuária genuinamente tropical. A iniciativa tem o desafio constante de garantir ao Brasil segurança alimentar e posição de destaque no mercado internacional de alimentos, fibras e energia.

3.2.1 Estrutura Organizacional

A sede da Embrapa fica situada em Brasília e é responsável por planejar, supervisionar, coordenar e controlar as atividades relacionadas à execução de pesquisa agropecuária e à formulação de políticas agrícolas. Esse trabalho é realizado por meio de Unidades Administrativas, que dão suporte à Diretoria-Executiva da Empresa [33].

No momento, a Embrapa conta com 43 Unidades Descentralizadas, também conhecidas como Centros de Pesquisa. As Unidades estão localizadas em diversos Estados do Brasil e trabalham em um modelo de administração que combina pesquisa, inovação e transferência de tecnologia em várias áreas do agronegócio.

3.2.2 Processo de Desenvolvimento

A Embrapa possibilita que suas 43 unidades desenvolvam seus softwares de forma independente, todos personalizados para o uso específico de cada unidade em suas linhas de pesquisa. As tecnologias empregadas em cada projeto também são decididas de modo independente, garantindo soluções totalmente adaptadas a cada área de pesquisa da empresa. Além disso, a grande maioria dos softwares são disponibilizados em servidores locais não tendo interferência da Sede em Brasília. Entretanto, para realizar integração com dados institucionais, as unidades precisam acessar sistemas e/ou bases de dados hospedados na sede.

No âmbito corporativo, os softwares administrativos e institucionais da Embrapa são desenvolvidos em sua sede, mais especificamente na GTI. Essa centralização possibilita a padronização dos processos e a execução de soluções tecnológicas que estão de acordo com as necessidades da entidade. Esses sistemas, bem como suas bases, também são hospedados no *datacenter* da GTI, o que permite a alta disponibilidade, segurança e integridade dos dados. Por meio da infraestrutura sólida do *datacenter*, os softwares administrativos e institucionais são mantidos operacionais, desempenhando um papel fundamental nas operações do dia a dia, com eficácia e efetividade.

Para assegurar a qualidade desse desenvolvimento centralizado, o desenvolvimento de software realizado pela equipe da Embrapa na GTI, na sede da empresa, segue orientação do Processo de Desenvolvimento de Produtos de *Software* da Embrapa (PDSE) [34]. Esse processo é composto de três fases: planejamento, construção e encerramento. Na fase de planejamento são realizadas as análises de requisitos. Na fase de construção é onde o software tem o seu desenvolvimento, podendo ter uma ou mais versões em ambiente de desenvolvimento, homologação e produção. A fase de encerramento acontece quando não se tem mais versões para homologação e todas as iterações foram desenvolvidas [34].

3.2.3 Adoção da Tecnologia Java

No contexto desse desenvolvimento corporativo centralizado, a Embrapa identificou a necessidade crescente de projetos desenvolvidos em Java. Esses projetos vêm sendo construídos ao longo dos anos, com o intuito de atender às demandas institucionais e garantir a eficiência das operações da empresa.

Diante dessa necessidade, em 2010, a GTI decidiu construir uma arquitetura que seria um componente reutilizável em muitos projetos para assegurar um nível mais alto de uniformidade, manutenção e reutilização. Essa arquitetura precisava suportar várias aplicações. Na época, as decisões relacionadas à segurança eram adequadas naquele momento, mas existe a possibilidade de que tais decisões não sejam mais consideradas robustas, pos-

sibilitando assim um aumento no risco relacionado à segurança, visto que o uso adequado de APIs de criptografia torna-se cada vez mais uma questão séria.

Tendo estabelecido o contexto organizacional e as características do desenvolvimento de software na Embrapa, é fundamental definir como os resultados deste trabalho serão mensurados e avaliados de forma objetiva. A escolha e aplicação adequada de métricas é crucial para validar cientificamente as descobertas e garantir a reprodutibilidade do estudo.

3.3 Projetos de Software Alvo do Estudo de Caso

Os projetos de software usados no estudo abrangem diferentes áreas e funcionalidades, desde a gestão de ativos tecnológicos até serviços de consulta e integração via APIs. Essa seção descreve cada um destes projetos, incluindo informações sobre seu domínio de aplicação, a quantidade de linhas de código e se faz uso de ferramentas de análise estática. A análise aqui apresentada oferece um panorama dos recursos e da complexidade de cada software, fornecendo informações importantes para a compreensão da infraestrutura tecnológica da Embrapa. Por questões de confidencialidade, o nome dos sistemas não serão expostos.

A seleção dos sistemas, *webservices* e componentes como projetos de software alvo deste estudo foi fundamentada principalmente na relevância dos mesmos para a Embrapa, tanto por resolver um problema corporativo relevante para a Embrapa (no caso dos sistemas) quanto por prover APIs ou bibliotecas que são reusadas por vários sistemas (caso dos web-services e componentes escolhidos). Esta escolha criteriosa foi motivada pela expectativa de que os resultados do estudo pudessem trazer contribuições substanciais para aprimorar as práticas de desenvolvimento de softwares na instituição, promovendo melhorias nos processos e na qualidade das soluções tecnológicas.

O Sistema 1 foi desenvolvido pela Embrapa e tem como objetivo principal o cadastramento e a manutenção do acervo de ativos tecnológicos. Um ativo tecnológico pode ou não ser um produto de tecnologia da informação, se for um sistema voltado para a pesquisa agropecuária. É possível, também, que o ativo tecnológico seja o produto de uma pesquisa voltada para o ramo agropecuário. Tais ativos podem ser tanto produzidos pela Embrapa quanto desenvolvidos em parceria com outras instituições. O Sistema 1 permite ainda o cadastramento dos ativos tecnológicos desenvolvidos por outras instituições, aqueles utilizados em ações de transferência de tecnologia pela empresa, e conta com 65.292 linhas de código Java de um total de 100.371 linhas de código.

O Sistema 2 realiza o gerenciamento de carteiras de projetos de pesquisa da Embrapa de âmbito nacional. Ele aborda a elaboração de chamadas para submissão de projetos, permite a colaboração entre os pesquisadores na elaboração dessas submissões seguindo regras específicas, além de lidar com a avaliação das submissões e o acompanhamento dos projetos aprovados durante sua execução até a conclusão. Esse sistema apresenta 319.630 linhas de código em Java, totalizando 452.353 linhas de código.

O Sistema 3 realiza a gestão de desempenho institucional, programático e de equipes e é utilizado por todas as unidades da Embrapa na gestão estratégica e na gestão de pessoas. Ele abrange 315.782 linhas desenvolvidas em Java, em um total de 442.039 linhas de código.

O Sistema 4 realiza a gestão do processo de progressão na carreira da Embrapa, possibilitando ao empregado progredir no plano de cargos e salários após a avaliação de desempenho anual. Ao todo, esse sistema abrange 20.985 linhas de código, sendo 15.902 escritas em Java.

O Sistema 5 realiza a gestão de viagens (diárias e passagens) à trabalho para os empregados e colaboradores eventuais da Embrapa, contando ao todo com 1.082.782 linhas de código, dentre as quais 55.457 foram desenvolvidas em Java.

O Sistema 6 sustenta a elaboração e publicação do Boletim de Comunicações Administrativas (BCA), permitindo que os responsáveis pelo Conselho de Administração, Diretoria-Executiva, Presidente, Chefes das Unidades Centrais e Descentralizadas possam acompanhar o processo de solicitação e aprovação dos atos administrativos. Esse sistema compreende 122.786 linhas de código no total, das quais 77.745 foram desenvolvidas em Java.

Webservices. Um total de sete serviços Web que oferecem APIs para os sistemas foram avaliados. O Webservice 1 (WS01) oferece uma API para a funcionalidade de assinar normas internas, portarias, deliberações e outros documentos expedidos pela diretoria e chefes de unidades da Embrapa. Possui 701 linhas de código Java (de um total de 1.018 linhas de código). O Webservice 2 (WS02) oferece uma API para a funcionalidade de consulta de remuneração de empregados. Possui 1.570 linhas de código Java (de um total de 2.019 linhas de código).

O Webservice (WS03) oferece uma API para a funcionalidade de consulta de informações relacionadas a empregados, unidades da Embrapa, projetos, colaboradores externos entre outras informações. Possui 4.258 linhas de código Java (de um total

de 4.839 linhas de código). O Webservice 4 (WS04) disponibiliza uma API para a funcionalidade de chatbot para auxílio aos usuários da *intranet*, possibilitando o esclarecimento de dúvidas e a abertura de solicitações para suporte da equipe técnica de atendimento ao usuário. Possui 681 linhas de código Java (de um total de 1.098 linhas de código). O Webservice 5 (WS05) disponibiliza uma API para as funcionalidades de obtenção e atualização de informações cadastrais de empregados da Embrapa. Este serviço possui 8.814 linhas de código Java, de um total de 9.392 linhas de código.

Finalmente, o Webservice 6 (WS06) e o Webservice 7 (WS07) disponibilizam APIs para a funcionalidade de autenticação de usuários internos ou externos no ambiente corporativo da Embrapa, gerando um *token* de segurança contendo as informações relevantes do usuário (WS06) e para a funcionalidade de integração do Sistema 6 com a intranet da Embrapa, possibilitando o *download* de arquivos de normas e deliberações de diretores e chefes de unidade através do próprio site da intranet (WS07). O WS06 possui 2.309 linhas de código Java (de um total de 2.759 linhas de código), enquanto que o WS07 possui 715 linhas de código Java (de um total de 1.011).

O Componente 1 provê uma arquitetura básica para o desenvolvimento de sistemas em Java, oferecendo desde a modularização de menus e telas de temas diversos, até serviços de autenticação e autorização de usuários e integração com banco de dados, além de outros recursos. É utilizado como uma biblioteca dentro dos sistemas que o adotam, totalizando 14.155 linhas de código, todas escritas em Java.

O Componente 2 provê autenticação e autorização de usuários, no serviço de *Lightweight Directory Access Protocol* (LDAP) e Microsoft *Active Directory* (AD). É utilizado como uma biblioteca dentro dos sistemas que o utilizam. Sua construção é anterior à do Componente 1 e, ao todo, possui 2.069 linhas de código, todas desenvolvidas em Java.

O processo de desenvolvimento de software na GTI não faz uso de ferramentas de análise estática.

3.4 Metodologia

A análise de segurança de APIs criptográficas foi realizada em dois estudos distintos, cada um relacionado a uma determinada fase do trabalho e respondendo a questões de pesquisa específicas, seguindo metodologias apropriadas para cada objetivo.

3.4.1 Estudo 1: Análise da Situação Atual

Este estudo inicial teve como objetivo diagnosticar o estado atual do uso de APIs de criptografia na organização. Para isso, combinou uma abordagem quantitativa — baseada na análise estática de 17 softwares com as ferramentas CogniCrypt e CryptoGuard, voltada à Questão de Pesquisa 1 (QP1) — e uma abordagem qualitativa — centrada em um grupo focal com desenvolvedores, destinado a compreender suas percepções sobre os alertas e a integração de ferramentas de análise estática no processo de desenvolvimento, atendendo à Questão de Pesquisa 2 (QP2).

3.4.1.1 Seleção de sistemas e coleta de informações

A análise destes softwares selecionados constituiu a primeira etapa do Estudo 1, direcionada à Questão de Pesquisa 1 (QP1). Esta etapa envolveu a análise estática de 17 softwares previamente escolhidos, empregando as ferramentas CogniCrypt e CryptoGuard, amplamente reconhecidas por sua eficácia na detecção de chamadas incorretas de APIs criptográficas e utilizadas tanto no ambiente acadêmico quanto industrial.

A ferramenta CryptoGuard foi utilizada para identificar falhas de implementação em código Java, como o uso inadequado de API de criptografia, geração de chaves e IVs, que frequentemente resultam em vulnerabilidades críticas [22]. A utilização do CryptoGuard permitiu a verificação de padrões mal utilizados e a geração de relatórios detalhados, fornecendo *insights* precisos sobre a conformidade de cada software com práticas seguras de desenvolvimento.

De forma complementar, o CogniCrypt ofereceu recursos adicionais de análise. Apesar de o CogniCrypt possuir um *plugin* para o Eclipse, sua execução foi via linha de comando para a realização da análise estática, não existindo uma dependência da IDE. O propósito do uso da ferramenta foi indicar possíveis usos incorretos de APIs de criptografia, estabelecendo, assim, uma verificação ao uso de padrões seguros de desenvolvimento. As conclusões da respectiva análise foram relatadas em formato de relatório SARIF. A ferramenta é capaz de detectar vários tipos de usos incorretos de APIs de criptografia, como o uso de algoritmos ou modos de operação considerados inseguros. A análise detalhada realizada utilizando o CogniCrypt possibilitou a investigação de segurança dos softwares estudados a um nível mais minucioso e auxiliou na descoberta de falhas relacionadas ao uso incorreto de APIs de criptografia.

Tanto o CogniCrypt quanto o CryptoGuard foram executados nos arquivos .jar desses componentes e/ou sistemas. Os dados obtidos da execução das ferramentas foram tabulados em planilhas e foi utilizada estatística descritiva (prevalência) para responder à Questão de Pesquisa 1 (QP1).

Com base nos resultados da primeira etapa do Estudo 1, referentes a uma ampla análise dos sistemas e componentes da Embrapa e ao uso de ferramentas de análise estática de código, tornou-se evidente a necessidade de uma pesquisa adicional para obter a percepção dos desenvolvedores em relação a essas ferramentas. Nesse sentido, uma segunda etapa do estudo 1 foi conduzida por meio da metodologia de grupo focal, buscando-se entender como as ferramentas SAST podem ser integradas de maneira eficaz nos processos de desenvolvimento e avaliar as reações desses desenvolvedores às notificações produzidas pelas ferramentas CogniCrypt e CryptoGuard nos softwares analisados [35].

3.4.1.2 Grupo Focal

A resposta para a Questão de Pesquisa 2 (QP2) veio à partir da metodologia de Grupo Focal (GF), que foi empregada na segunda etapa do Estudo 1. Essa etapa buscou obter dados sobre a integração de ferramentas SAST pelos desenvolvedores de software nos processos de desenvolvimento, observando também suas reações aos *warnings* emitidos por ferramentas de análise estática, como CogniCrypt e CryptoGuard, nos sistemas em que trabalham, bem como identificar fatores técnicos (por exemplo, dependências entre módulos e componentes reutilizáveis) e fatores organizacionais (como decisões de gestão tecnológica e a incorporação de sistemas desenvolvidos por terceiros) que influenciam tanto a descoberta quanto a correção desses usos indevidos.

Um Grupo Focal é um método qualitativo comumente adotado para a coleta de dados em muitas configurações, que variam de trabalhos acadêmicos a estudos aplicados que buscam se envolver ativamente para resolver problemas reais. Consiste em reunir participantes com características ou experiências afins para discutir tópicos específicos entre si sob condições moderadas, coletando dados qualitativos valiosos. A dinâmica entre os participantes cria um ambiente propício para a livre troca de opiniões e a coleta cooperativa de informações que seriam muito mais difíceis, ou simplesmente impossíveis, de coletar por meio de outros métodos, como questionários de escolha múltipla ou entrevistas isoladas [36].

O método de GF tem se mostrado útil na engenharia de software para obter *feedback* qualitativo e explorar problemas complexos. Kontio et al. afirmam que o GF é um instrumento viável em diversas fases de pesquisa de campo empírica. Sua aplicação abrange desde a coleta inicial de requisitos até a validação de conceitos e a investigação de processos [37]. O diálogo intermediado contribui significativamente para a obtenção do conhecimento sobre práticas e perspectivas dos desenvolvedores em relação a ferramentas e processos técnicos.

Além de possibilitar uma visão mais aprofundada das relações grupais e dos fatores sociais que influenciam a adoção de tecnologias, o GF — conforme salientado por Bar-

bour — oferece subsídios para a coleta de dados robustos e para a investigação de nuances culturais e organizacionais que poderiam passar despercebidas em abordagens mais restritas [38]. Para que as conclusões sejam ainda mais robustas, cada etapa do processo deve ser planejada e executada com atenção detalhada: a seleção de participantes, a condução das sessões e a análise dos dados precisam de um cuidado especial para fortalecer a confiabilidade do estudo.

Seguindo esse mesmo rigor metodológico, a sessão de GF realizada na Embrapa teve a duração de 1 hora e 12 minutos, tendo apresentação em slides e gravação do áudio da reunião por meio da plataforma Microsoft Teams. Como parte essencial da preparação para essa sessão, executaram-se as ferramentas CogniCrypt e CryptoGuard em um conjunto de projetos Java (Caracterizados na seção *Projetos de Software Alvo do Estudo de Caso.*) que os participantes desenvolvem ou mantêm, o que permitiu identificar e analisar previamente alertas de usos indevidos de API de criptografia nos sistemas objetos desse estudo. Somente após essa avaliação detalhada dos avisos gerados pelas duas ferramentas é que a sessão foi agendada, assegurando o refinamento de materiais relevantes para discussão e melhor direcionando o foco das questões tratadas durante o encontro. Esses procedimentos visaram aumentar a precisão das informações coletadas e embasar as reflexões dos participantes com exemplos concretos de uso de ferramentas SAST, ao mesmo tempo em que reforçaram a confiabilidade dos achados ao alinhar os dados empíricos ao propósito do estudo.

A sessão de GF foi estruturada em três etapas distintas. Na primeira etapa, o foco recaiu exclusivamente na coleta de informações demográficas. Buscou-se obter detalhes sobre os participantes, como sua experiência em desenvolvimento de software e segurança de software, funções desempenhadas na instituição e familiaridade com ferramentas SAST. Além disso, foram explorados aspectos dos projetos selecionados para o estudo, incluindo o domínio do projeto, número de contribuidores e tamanho (em termos de linhas de código). Na segunda etapa, foram coletadas percepções sobre a utilização de ferramentas SAST nos fluxos de trabalho de desenvolvimento de software na Embrapa, bem como as opiniões dos participantes sobre as ferramentas com as quais já estavam familiarizados. Na etapa final, concentraram-se os esforços nas percepções dos participantes em relação aos alertas gerados por ambas as ferramentas, CogniCrypt e CryptoGuard, para os sistemas analisados.

A tabela 3.1 apresenta os dados demográficos dos participantes do GF contendo suas experiências como profissionais de desenvolvimento e a experiência prévia com ferramentas SAST[1].

[1]Os nomes dos participantes foram ocultados para preservar sua privacidade e confidencialidade.

Tabela 3.1: Demografia dos Participantes do Grupo Focal. (Fonte: Própria)

Participante	**Experiência (anos)**	**Função**	**Ferramentas SAST**
P1	20	Desenvolvedor	SonarQube
P2	22	Desenvolvedor	Nenhuma
P3	20	Desenvolvedor	SonarQube
P4	17	Desenvolvedor	SonarQube
P5	26	Desenvolvedor	SonarQube
P6	15	Desenvolvedor	Nenhuma

Na terceira etapa, foi seguido um protocolo de entrevista semiestruturada, incorporando elaborações espontâneas para diversas perguntas. Essa abordagem foi idealizada com o objetivo de incentivar os participantes a fornecer respostas detalhadas para questões que dependem do contexto específico de cada situação. Abaixo, são apresentados alguns exemplos de perguntas formuladas durante essa etapa da sessão de GF:

- O uso da função de *hash Message Digest Algorithm 5* (MD5) nesse contexto pode representar uma potencial ameaça à segurança?
- Em sua perspectiva, é crucial que a equipe aborde e corrija esse problema? Por favor, descreva o processo de tomada de decisão.
- Em que medida as mensagens de alerta geradas por essas ferramentas fornecem clareza? Há algum detalhe essencial que se acredita estar ausente nesses alertas?
- Quais sugestões podem ser indicadas para aprimorar a eficácia dessas ferramentas SAST e elevar sua qualidade geral?

No que se refere à condução da sessão, o Grupo Focal foi realizado de forma presencial, conduzido pelo autor deste trabalho, que atuou como moderador. A plataforma Microsoft Teams foi utilizada apenas para a gravação da apresentação em slides e do áudio da reunião, servindo exclusivamente como apoio de registro. Ao todo, sete desenvolvedores mantenedores diretos dos sistemas analisados foram convidados, dos quais seis participaram da sessão. Os convites foram feitos de forma individualizada e pessoalmente a cada desenvolvedor, ocasião em que se explicou o propósito do Grupo Focal e os tópicos que seriam discutidos. Todos manifestaram interesse em participar, e apenas um informou, nesse momento, que não poderia comparecer devido a compromissos pessoais.

No decorrer da sessão, foram apresentados os relatórios completos gerados pelas ferramentas CogniCrypt e CryptoGuard. Ao visualizar os alertas, cada participante se identificava como mantenedor do sistema correspondente, conferindo autenticidade às discussões.

Além disso, as perguntas do protocolo metodológico desta pesquisa foram aplicadas de forma estruturada para cada alerta, assegurando consistência e comparabilidade entre as respostas, sem prejuízo de explorações contextuais espontâneas.

Em decorrência das discussões no Grupo Focal, os participantes analisaram em detalhes os usos inadequados de APIs de criptografia identificados pelas ferramentas e fizeram uma avaliação criteriosa do impacto de manter o código vulnerável em produção, bem como os possíveis efeitos de sua correção. A conclusão da equipe foi pela necessidade de implementar as correções das vulnerabilidades identificadas, decisão que levou ao desenvolvimento do segundo estudo desta dissertação, voltado ao processo de correção desses usos inadequados e sua documentação.

3.4.2 Estudo 2: Resolução das Vulnerabilidades Criptográficas

O segundo estudo investigou a resolução das vulnerabilidades criptográficas por duas estratégias conduzidas separadamente, mas analisadas sob a mesma linha de base e critérios comuns: (i) correção manual das ocorrências reportadas; e (ii) geração de propostas de correção por modelos de linguagem (SLMs/LLMs), utilizando a técnica de *one-shot prompting* (vide Seção 5.2.4). A inclusão da estratégia (ii) visa verificar a capacidade dos modelos de produzir correções que suprimam os alertas nas ferramentas SAST e preservem a compilação e o funcionamento dos sistemas, permitindo comparação direta com a linha de base manual.

Para avaliar o impacto das intervenções realizadas no código-fonte durante este estudo, tanto no caso das correções manuais quanto das correções semiautomatizadas com SLMs/LLMs, foi empregada a métrica *`Volume de Alterações`* no Código-Fonte (Code Churn). Essa métrica, amplamente utilizada na literatura como indicador do esforço de manutenção [39, 40], corresponde à soma absoluta do número de *Lines of Code* (Linhas de Código – LOC) inseridas e excluídas durante a implementação de cada correção.

A utilização do Volume de Alterações possibilita uma análise mensurável do impacto das modificações, permitindo relacionar o tamanho e a complexidade dos ajustes ao volume de código efetivamente trabalhado.

3.4.2.1 Correção manual dos usos indevidos de APIs de criptografia

Ao término da sessão de GF, os participantes perceberam a necessidade de corrigir os usos indevidos de APIs de criptografia encontrados e discutidos durante a reunião e, por conseguinte, concordaram em iniciar esse processo. Em virtude dessa decisão, um segundo estudo foi iniciado com o intuito de analisar o impacto das correções aplicadas nas vulnerabilidades encontradas. O objetivo principal desse estudo foi entender como as

mudanças introduzidas afetaram a segurança e a estabilidade dos projetos corporativos da Embrapa, garantindo que as adaptações não comprometessem o funcionamento dos sistemas, mas que melhorassem sua robustez contra possíveis ameaças.

A primeira etapa do segundo estudo respondeu à Questão de Pesquisa 3 (QP3) ao aplicar correções manuais nos maus usos de API de criptografia encontrados pelas ferramentas de análise estática e avaliar os impactos dessas correções no uso dos softwares.

A metodologia envolveu as seguintes etapas:

1. **Planejamento das Correções:** Com base nos resultados e nas discussões do Estudo 1, foi elaborado um plano detalhado para a correção manual de cada tipo de vulnerabilidade (*Common Weakness Enumeration* (CWE)) identificada em cada software afetado. Esse planejamento considerou o escopo das modificações necessárias, preservando a lógica original das aplicações e a compatibilidade com os sistemas existentes.

2. **Implementação das Correções Manuais:** As alterações foram aplicadas diretamente nos trechos de código vulneráveis, respeitando a lógica original das aplicações e preservando sua integridade funcional. Nesse processo, foram seguidas as especificidades de cada ferramenta SAST, considerando tanto as mensagens geradas pelo CryptoGuard quanto as regras CrySL utilizadas pelo CogniCrypt. Além disso, foi dada prioridade aos componentes reutilizáveis, como o Componente 1, em virtude do seu caráter reutilizável e do potencial de disseminação de vulnerabilidades em diversos sistemas corporativos.

3. **Avaliação dos Impactos:** Os impactos das correções foram avaliados tanto em termos de segurança — verificando a eliminação dos alertas reportados pelas ferramentas de análise estática — quanto em termos de estabilidade e desempenho dos sistemas corporativos. Adicionalmente, a contagem de linhas de código modificadas foi utilizada como métrica para mensurar o esforço de manutenção necessário em cada caso, permitindo relacionar o impacto das mudanças ao tamanho e à complexidade dos ajustes realizados.

3.4.2.2 Correção semiautomatizada com SLMs/LLMs

Visando responder à Questão de Pesquisa 4 (QP4), esta segunda etapa do Estudo 2 investiga a viabilidade e a efetividade do uso de modelos de linguagem pequenos (SLMs) e grandes (LLMs) na proposição de correções para as vulnerabilidades identificadas no Estudo 1. A metodologia desta fase compreendeu:

1. **Geração de Correções Automatizadas:** Utilização de SLMs e LLMs selecionados para produzir propostas de código corrigido, a partir do código vulnerável e dos relatórios fornecidos pelas ferramentas SAST. A técnica de *one-shot prompting* foi adotada para padronizar as interações, fornecendo ao modelo instruções detalhadas e um único exemplo de tarefa.

2. **Execução Controlada dos Modelos:** Os SLMs (Qwen 3 4B, Mistral-7B, Gemma 3n E4B) foram executados localmente em ambiente dedicado, garantindo reprodutibilidade dos experimentos. Já os LLMs (Gemini 2.5 Pro, Claude Sonnet 4, ChatGPT 5) foram acessados por meio de serviços *Software-as-a-Service* (SaaS), refletindo cenários práticos de uso em ambientes corporativos.

3. **Critérios de Avaliação:** As propostas de correção geradas foram analisadas em dois níveis:

   (a) Correção aparente, referente à eliminação dos avisos reportados pelas ferramentas SAST.

   (b) Correção funcional, que exige a compilação bem-sucedida e manutenção da lógica do sistema.

   Essa dupla avaliação permitiu diferenciar soluções apenas sintaticamente conformes daquelas efetivamente aplicáveis.

4. **Análise de Suficiência e Limitações:** Investigou-se se uma única iteração por método vulnerável seria suficiente, além de documentar erros recorrentes, limitações práticas e padrões de falha observados nos resultados.

5. **Síntese Comparativa:** Os resultados obtidos foram comparados entre modelos e consolidados em tabelas, possibilitando avaliar a efetividade relativa de SLMs e LLMs e identificar recomendações práticas para a adoção dessas abordagens no ciclo de desenvolvimento seguro.

Capítulo 4

Resultados do Estudo 1: Análise da Situação Atual

Este capítulo expõe os resultados encontrados a partir da aplicação da metodologia detalhada no Capítulo 3, com o propósito de responder às questões de pesquisa formuladas em relação ao uso de APIs de criptografia na Embrapa. Conforme estabelecido, a pesquisa foi realizada em dois estudos principais. Neste primeiro estudo foi realizada uma análise estática de código para identificar a prevalência de vulnerabilidades nos softwares testados (com o objetivo de responder à QP1) e a condução de um grupo focal para compreender a visão dos desenvolvedores em relação aos alertas gerados pelas ferramentas SAST (com o intuito de responder à QP2).

As informações obtidas em cada etapa foram processadas e examinadas para oferecer perspectivas sobre as práticas atuais e os desafios associados à segurança criptográfica no contexto do estudo. A seção inicial apresenta os resultados da análise estática realizada com as ferramentas CogniCrypt e CryptoGuard nos artefatos de software selecionados. Com a análise minuciosa dos resultados, foi possível categorizar e documentar os principais tipos de vulnerabilidades encontradas, proporcionando uma percepção mais estruturada dos problemas de implementação criptográfica que precisam ser abordados.

Em seguida, na segunda seção, são apresentadas as análises qualitativas das percepções dos desenvolvedores, obtidas e examinadas a partir das discussões realizadas no grupo focal. Esses achados buscam esclarecer de que maneira as equipes compreendem os alertas de segurança e a inclusão de ferramentas SAST em seus processos de trabalho.

A apresentação desses resultados nas próximas seções tem como objetivo proporcionar uma visão clara da situação encontrada na Embrapa e fornecer fundamentação empírica para as discussões e conclusões desse trabalho.

4.1 Resultados da Análise Estática

Os resultados obtidos pelas duas ferramentas de análise estática, CogniCrypt e CryptoGuard, apontaram para a predominância de problemas associados ao mau uso de criptografia. Esse resultado se encaixa nos achados da literatura sobre o tema, que destaca os riscos de segurança diante do uso inadequado desse recurso quando mal implementado [2, 3]. A aplicação das metodologias foi de extrema importância para a aquisição de um conjunto de recomendações práticas que foram direcionadas aos desenvolvedores no intuito de aprimorar a proteção das aplicações contra as vulnerabilidades apontadas.

4.1.1 Vulnerabilidades Identificadas pela Análise Estática

Nesta subseção, são apresentados os principais usos indevidos de APIs criptográficas identificados durante a análise estática do código-fonte utilizando as ferramentas CogniCrypt e CryptoGuard. Os *misuses* detectados foram classificados e consolidados em uma tabela comparativa, permitindo uma visão abrangente das vulnerabilidades encontradas por cada ferramenta e seus potenciais impactos na segurança dos softwares [35].

Dentre os softwares analisados neste estudo, os seguintes tiveram algum tipo de apontamento das ferramentas de análise estática: Componente 1, Componente 2, Sistema 2, Sistema 3 e Sistema 6, todos eles utilizando APIs de criptografia. Para cada sistema, foram exibidas as ocorrências de *warnings*. Essa abordagem permite uma avaliação completa do impacto que os erros de segurança podem ter no ambiente de produção e a dificuldade em corrigi-los.

A Tabela 4.1 detalha os usos indevidos de APIs criptográficas identificados em cada software analisado. Isso inclui a categorização de cada uso indevido, qual ferramenta de análise estática – CogniCrypt e CryptoGuard – encontrou a vulnerabilidade e a descrição do uso indevido de APIs criptográficas.

Já as Tabelas 4.2 e 4.3 detalham cada vulnerabilidade encontrada por software, informando qual foi afetado, a quantidade de ocorrências encontradas e o CWE de cada vulnerabilidade. O CWE é utilizado como referência para realizar uma categorização e descrever cada tipo de vulnerabilidade identificada. Este catálogo fornece uma classificação de reconhecimento amplo, que padroniza a identificação de fraquezas estruturais em projetos de software, facilitando, dessa forma, a priorização e mitigação de problemas de segurança [41].

A identificação dos usos indevidos de APIs de criptografia, de acordo com o CWE [41], neste trabalho, permitiu classificar cada um deles, de forma a associar a tipos de fraquezas bem definidos. Dessa forma, o CWE contribuiu para contextualizar os resultados obtidos pelas ferramentas de análise estática - CogniCrypt e CryptoGuard -, fornecendo uma vi-

Tabela 4.1: Vulnerabilidades encontradas nas análises estáticas. (Fonte: Própria)

Tipo de Uso Indevido	**Ferramenta**	**Descrição**
Uso de algoritmo inseguro	CogniCrypt, CryptoGuard	AES/ECB/PKCS5Padding, DES/ECB/NoPadding
Geração incorreta de chave	CogniCrypt	Segundo parâmetro não gerado corretamente
Preparação incorreta de material de chave	CogniCrypt	Primeiro parâmetro não gerado corretamente
Uso de protocolo SSL obsoleto	CogniCrypt	Uso de 'SSL' em vez de TLSv1.2 ou TLSv1.3
Geração incorreta de TrustManagers	CogniCrypt	Segundo parâmetro não gerado corretamente
Uso de PRNG não confiável	CryptoGuard	Uso de java.util.Random
Uso de chave constante no código	CryptoGuard	Constantes usadas como chaves
Uso de HostNameVerifier não confiável	CryptoGuard	HostNameVerifier que aceita todos os hostnames
Uso de TrustManager não confiável	CryptoGuard	TrustManager que aceita todos os certificados
Uso de protocolo HTTP inseguro	CryptoGuard	Uso de HTTP em vez de HTTPS

são mais clara do potencial impacto de cada vulnerabilidade, orientando ações corretivas eficazes. De acordo com a MITRE [41], o objetivo principal do CWE é ajudar no entendimento e na padronização de vulnerabilidades de software, estando, dessa forma, de acordo com a necessidade de mapear e corrigir falhas em sistemas que utilizam APIs de criptografia.

4.2 Percepções dos Desenvolvedores

Diversas preocupações organizacionais emergiram durante a análise. Por exemplo, os participantes do GF, com diferentes níveis de familiaridade com ferramentas de análise de programas, observaram que, embora a segurança seja amplamente reconhecida como um aspecto crítico para a empresa, ela ainda não recebe a mesma ênfase prática nas rotinas de desenvolvimento. Na instituição, há uma forte confiança nas equipes de operações, refletindo a percepção de que medidas como políticas de *firewall* já representam uma camada importante de proteção [35]. Essa visão, embora legítima, indica oportunidade de ampliar a integração entre operações e desenvolvimento na abordagem da segurança.

Tabela 4.2: Detalhamento das Vulnerabilidades por Software - Parte 1. (Fonte: Própria)

Ref	Uso Indevido	CWE	Software Afetado	Quant.
(AES/ECB)	Uso de algoritmo inseguro	CWE-327	Componente 1	2
			Componente 2	1
			Sistema 2	2
			Sistema 3	2
			Sistema 6	2
(KeyGen)	Geração incorreta de chave	CWE-320	Componente 1	2
			Componente 2	1
			Sistema 2	2
			Sistema 3	2
			Sistema 6	2
(KeyMat)	Preparação incorreta de material de chave	CWE-321	Componente 1	2
			Componente 2	1
			Sistema 2	2
			Sistema 3	2
			Sistema 6	2
(SSL)	Uso de protocolo SSL obsoleto	CWE-326	Componente 1	3
			Componente 2	0
			Sistema 2	0
			Sistema 3	0
			Sistema 6	2
(TrustM)	Geração incorreta de TrustManagers	CWE-295	Componente 1	1
			Componente 2	0
			Sistema 2	0
			Sistema 3	0
			Sistema 6	2

Além disso, as discussões no GF evidenciaram que fatores estruturais, como a forte dependência entre módulos e componentes arquiteturais, a coexistência de código interno com sistemas desenvolvidos por terceiros e decisões de gestão tecnológica que condicionam ou postergam atualizações estruturais, funcionam como fatores complicadores tanto para a identificação quanto para a correção das vulnerabilidades criptográficas.

Esses fatores, somados às percepções gerais sobre segurança discutidas anteriormente, reforçaram uma compreensão compartilhada entre os participantes. Esta compreensão sobre segurança foi expressa pelo participante P1:

Tabela 4.3: Detalhamento das Vulnerabilidades por Software - Parte 2. (Fonte: Própria)

Ref	Uso Indevido	CWE	Software Afetado	Quant.
(**PRNG**)	Uso de PRNG não confiável	CWE-338	Componente 1	2
			Componente 2	1
			Sistema 2	1
			Sistema 3	0
			Sistema 6	0
(ConstKey)	Uso de chave constante no código	CWE-547	Componente 1	16
			Componente 2	0
			Sistema 2	0
			Sistema 3	0
			Sistema 6	0
(HostNameVer)	Uso de HostNameVerifier não confiável	CWE-601	Componente 1	2
			Componente 2	0
			Sistema 2	0
			Sistema 3	0
			Sistema 6	1
(TrustManAll)	Uso de TrustManager não confiável	CWE-349	Componente 1	4
			Componente 2	0
			Sistema 2	0
			Sistema 3	0
			Sistema 6	4
(HTTP)	Uso de protocolo HTTP inseguro	CWE-319	Componente 1	0
			Componente 2	1
			Sistema 2	0
			Sistema 3	0
			Sistema 6	0

> *"Nosso objetivo é entregar o que o cliente solicitou. Assim, outras preocupações (como segurança) podem ser, de alguma forma, negligenciadas. Acreditamos que a segurança deve ser delegada à configuração do servidor de aplicação, à VPN... e garantida pela infraestrutura. Nós nos concentramos apenas no desenvolvimento de funcionalidades."*

Por essa razão, o foco principal das equipes de desenvolvimento tende a concentrar-se nas demandas específicas do domínio, o que, em alguns casos, reduz o espaço dedicado aos requisitos não funcionais, como a segurança.

Além disso, os desenvolvedores assumem que as decisões tomadas sobre a segurança nos componentes arquiteturais, juntamente com as políticas de segurança implementadas na infraestrutura, são adequadas — sugerindo uma necessidade de maior sensibilização sobre segurança e uma compreensão mais ampla das diferentes formas de ameaça, tanto internas quanto externas. Isso é particularmente preocupante quando se considera que uma vulnerabilidade em um componente arquitetural ou reutilizável pode propagar um problema de segurança por vários sistemas. De fato, algumas vulnerabilidades relatadas pelo CogniCrypt e pelo CryptoGuard foram encontradas em componentes arquiteturais. Esta abordagem sobre segurança foi evidenciada na fala do participante P2:

> *"Nós nos concentramos muito nos requisitos (funcionais), atendendo à área de negócios, e sempre acreditamos que a arquitetura nos protege, acabando por não dar toda essa atenção [à segurança]."*

Para um melhor entendimento das discussões realizadas, algumas conclusões parciais serão apresentadas ao longo desta subseção.

Conclusão 1: Observou-se entre os participantes do GF uma elevada confiança na arquitetura institucional, o que, em alguns casos, reduz a atenção dedicada à conscientização sobre segurança.

P1: *"Nós nos concentramos apenas no desenvolvimento de funcionalidades."*

De fato, os participantes reconheceram que algumas das vulnerabilidades relatadas pelo CogniCrypt e pelo CryptoGuard estão em componentes arquiteturais desenvolvidos há mais de 10 anos. Na época, essas decisões relacionadas à criptografia eram consideradas adequadas. No entanto, devido às dependências desses componentes, atualizá-los tornou-se uma tarefa desafiadora, arriscada e custosa, cuja execução tende a ser postergada diante de outras prioridades institucionais. Durante a sessão de Grupo Focal, os participantes demonstraram uma preocupação geral com o impacto de alterar um componente arquitetural para corrigir uma vulnerabilidade. O participante P3 detalhou esta complexidade técnica:

> *"[...] se você quiser atualizar o componente arquitetural de criptografia, também precisaria atualizar outras dependências, incluindo a versão do Java que usamos aqui. É um efeito bola de neve, onde um problema leva a outro, e, antes que você perceba, está lidando com um ambiente significativamente desatualizado."*

Além da elevada confiança na robustez da infraestrutura, os participantes tiveram uma experiência frustrante ao utilizar o *SonarQube*, o que os motivou a remover ferramentas de análise estática de seus *pipelines*, evidenciando um desafio na adoção dessas ferramentas.

Além disso, segundo os participantes, existe uma literatura mais ampla sobre testes de software, mas a literatura sobre análise estática é escassa. Por essa razão, os potenciais benefícios do uso de ferramentas de análise estática ainda não são plenamente percebidos pelas equipes. No entanto, ao serem apresentadas as vulnerabilidades encontradas nos componentes arquiteturais, os participantes reconheceram a necessidade de dar maior atenção à segurança de software. Como existia um planejamento para uma eventual transição das plataformas de desenvolvimento (de Java para .NET), consideraram esse o momento ideal para iniciar tal esforço. Esta consciência sobre a importância da segurança foi expressa pelo participante P2:

> *"[...] Acredito que, à medida que as coisas progridem, é realmente importante que os desenvolvedores comecem a ter essa perspectiva de segurança durante o desenvolvimento, que é o que as novas tendências estão mostrando, a importância [...]."*

Uma das vulnerabilidades relatadas pelo CogniCrypt e pelo CryptoGuard estava em uma biblioteca de classes que implementa criptografia (Componente 1) e é amplamente utilizada para criptografar as senhas de usuários externos. Esse trecho vulnerável de código na biblioteca instancia um cifrador AES usando o modo ECB, que não é recomendado há vários anos e está relacionado a uma CWE[1] específica.

Ao apresentar essa vulnerabilidade aos participantes, iniciou-se uma discussão extensa para tentar identificar quais sistemas reutilizam esse componente arquitetural. Infelizmente, os resultados das ferramentas apenas informam que houve um mau uso de uma API de criptografia, mas não revelam se o código vulnerável é acessível a partir dos pontos de entrada do sistema (limitação das ferramentas de análise estática).

Não houve consenso na discussão, e o entendimento final foi de que muitos sistemas podem importar essa biblioteca interna de criptografia, mas apenas alguns efetivamente chamam o método com o código vulnerável. No final, os participantes perceberam que o impacto de corrigir essa vulnerabilidade poderia ser pequeno e concordaram em realizar a correção. Esta percepção sobre o uso limitado da biblioteca foi detalhada pelo participante P4:

> *"[...] É porque [a biblioteca é importada neste sistema], mas esse código não é usado. De fato, (um sistema específico) é o único sistema que o utiliza, e para definir as credenciais para pesquisadores ad hoc [...] por exemplo, essas ferramentas apontam coisas no código da biblioteca que não estão sendo usadas por este sistema específico. Você tem essa classe aqui, só como exemplo, que está disponível na biblioteca, mas, no momento, este programa específico não a utiliza. Essas ferramentas chegam a esse nível (de detalhamento)?"*

[1]CWE-327: Uso de um algoritmo criptográfico quebrado ou arriscado. Disponível em: `https://cwe.mitre.org/data/definitions/327.html`. Acesso em: 01 jan. 2025.

Conclusão 2: CogniCrypt e CryptoGuard deveriam ser aprimorados para indicar se o mau uso de uma API de criptografia provém de uma biblioteca ou do sistema em análise; e se o código vulnerável é acessível ou não.

Após a apresentação de outros casos de mau uso da classe JCA *Cipher* em componentes arquiteturais, os participantes também concordaram que deveriam examinar os casos específicos após a sessão de Grupo Focal e decidir se corrigiriam ou não. Os desenvolvedores apresentaram diferentes níveis de familiaridade com a biblioteca JCA, como ilustrado na fala do participante P1:

> *"[...] (para este caso específico, para corrigir os mau usos), seria necessário desenvolver um novo código ou apenas alterar as configurações do cifrador (de DES para outra coisa)? [...] Certo. Parece bom, vamos analisar isso depois."*

Além disso, os participantes manifestaram incerteza quanto à possibilidade de exploração prática dos usos indevidos de APIs de criptografia identificados pelo CogniCrypt e pelo CryptoGuard. Isso ocorreu porque alguns dos alertas vinham de código de teste ou utilizavam a classe *java.util.Random* em contextos não sensíveis.

Por fim, ao final do GF, os participantes começaram a discutir resultados acionáveis e demonstraram interesse em incluir ferramentas de análise estática em seus pipelines, avaliar o impacto de corrigir os maus usos das APIs de criptografia discutidos durante a sessão (e eventualmente corrigi-los), e remover código legado que incluía esses maus usos. Este momento de reflexão sobre as ações necessárias gerou uma discussão construtiva entre os participantes P5, P6 e P2:

> *P5: "Devemos alterar isso (código inseguro) imediatamente. Está nas bibliotecas arquiteturais?"*
>
> *P6: "Esses alertas vêm de uma biblioteca, que é importada por cada aplicação, certo? Então, precisaríamos alterar a biblioteca uma vez e depois atualizar as aplicações. Passar para cada desenvolvedor que precisar."*
>
> *P2: "Suspeito que esse código esteja obsoleto. Provavelmente poderíamos simplesmente excluir esse pacote de criptografia. Não acho que o usamos para nada."*

Algumas ideias sobre como integrar essas ferramentas também surgiram. Por exemplo, os participantes consideraram mais relevante executar as análises em sistemas com interfaces externas e, talvez, definir um uso periódico das ferramentas, em vez de executá-las durante os ciclos de integração contínua. Esta sugestão de abordagem pragmática foi discutida pelos participantes P6 e P2:

> *P6: "[...] devemos focar em aplicações que têm mais acesso público geral, correto?"*
>
> *P2: "[...] Também acho que seria uma boa prática (o uso de ferramentas de análise estática) para nós. Quero dizer, se não pudermos executar um analisador de código estático de segurança o tempo todo — realizar uma varredura periódica em nosso código para detectar falhas de segurança."*

Essa percepção parece ter sido motivada por uma experiência malsucedida ao utilizar o SonarQube, como explicado pelo participante P2:

> *"Como o SonarQube estava causando problemas em nosso pipeline, podemos executar essas ferramentas em nossas máquinas locais de desenvolvimento. Antes de enviar o código para nosso (sistema de gerenciamento de configuração), podemos identificar as falhas, corrigi-las e evitar implantar código com vulnerabilidades em produção."*

Além disso, alguns participantes consideraram o uso de ferramentas de análise estática importante para a transferência de conhecimento. Esta visão sobre o potencial educativo das ferramentas foi articulada pelo participante P2:

> *"Essas ferramentas podem nos ajudar a nos familiarizar mais com questões de segurança, que às vezes são negligenciadas. Essas ferramentas gradualmente revelam onde os erros ocorrem, certo? Então, acredito que seja uma boa prática para reconsiderarmos nosso processo de desenvolvimento e adicionarmos essa camada de segurança."*

Conclusão 3: Os desenvolvedores veem valor em integrar ferramentas SAST em seus processos de desenvolvimento.

Capítulo 5

Resultados do Estudo 2: Resolução das Vulnerabilidades Criptográficas

Este capítulo complementa a análise apresentada anteriormente, aprofundando-se nos processos de correção das vulnerabilidades identificadas. Inicialmente, são detalhados os resultados da correção manual das vulnerabilidades no código-fonte, destacando as dificuldades técnicas e operacionais enfrentadas durante esse processo, bem como os impactos que essas correções tiveram sobre os sistemas avaliados. Essa etapa busca responder à questão de pesquisa QP3, que investiga especificamente o impacto decorrente da correção dessas vulnerabilidades nos sistemas corporativos da Embrapa.

É reservada ao Componente 1 uma atenção especial, devido à sua ampla utilização por múltiplos sistemas e ao elevado número de vulnerabilidades detectadas nesse componente, características essas que o tornam crítico do ponto de vista da segurança, podendo funcionar como um vetor potencial para disseminação de vulnerabilidades dentro da organização.

Em seguida, o capítulo apresenta os resultados obtidos com as correções semiautomatizadas realizadas por meio de modelos de linguagem (SLMs e LLMs).Essa fase responde à questão de pesquisa QP4, dedicada à avaliação da efetividade desses modelos na correção das vulnerabilidades identificadas. Esta seção detalha a metodologia aplicada e examina questões importantes, como a suficiência de uma única iteração para a correção efetiva das vulnerabilidades identificadas, os principais erros observados durante as tentativas automatizadas e as impressões gerais resultantes do uso dessas tecnologias. Ao final, são oferecidas recomendações práticas que visam maximizar a eficácia da adoção dessas ferramentas no contexto da segurança criptográfica praticada na Embrapa, fundamentadas nos resultados conclusivos obtidos.

5.1 Resultados das correções manuais

Ao término da sessão de GF, os participantes perceberam a necessidade de corrigir os usos indevidos de APIs de criptografia encontrados e discutidos durante a reunião e, por conseguinte, concordaram em iniciar esse processo. Em virtude dessa decisão, um estudo foi iniciado com o intuito de analisar o impacto das correções aplicadas nas vulnerabilidades encontradas. O objetivo principal desse estudo é entender como as mudanças introduzidas afetam a segurança e a estabilidade dos projetos corporativos da Embrapa, garantindo que as adaptações não comprometam o funcionamento dos sistemas, mas que melhorem sua robustez contra possíveis ameaças.

Esse estudo responde a Questão de Pesquisa 3 (QP3) ao aplicar as correções manuais dos maus usos de API de criptografia encontrados pelas ferramentas de análise estática e avaliar os impactos dessas correções no uso dos softwares.

5.1.1 Correção das vulnerabilidades no código-fonte

Como é possível observar nas tabelas 4.2 e 4.3, foram encontradas 67 vulnerabilidades nos usos de APIs criptográficas, no total. Para realizar as correções desses usos inadequados, elaborou-se um planejamento que compreendeu a análise do escopo das modificações de código necessárias para cada tipo de CWE apontado e para cada software que apresentou o referido uso indevido, a fim de apoiar e organizar essa atividade [35].

O planejamento das correções foi elaborado com base nas experiências dos participantes do Grupo Focal como desenvolvedores e/ou mantenedores dos sistemas, componentes e webservices objetos de estudo desta pesquisa. Como observado anteriormente, verificou-se que os participantes não possuíam experiência prévia com o uso das ferramentas CogniCrypt e CryptoGuard, e apenas uma baixa experiência prévia com o SonarQube. Esse fator influenciou de maneira significativa o processo, desde a etapa de planejamento inicial até a execução das correções.

Um dos principais desafios esteve relacionado à interpretação dos relatórios gerados tanto pelo CogniCrypt quanto pelo CryptoGuard. Apesar de a premissa dessas ferramentas ser a geração de relatórios de fácil compreensão para auxiliar os responsáveis pelos softwares, observou-se, na prática, dificuldades para identificar o local exato do código onde a correção deveria ser aplicada, bem como a sequência de etapas necessárias para efetuar as devidas correções.

Uma dificuldade em particular foi observada nas correções dos usos inadequados apontados nos relatórios do CogniCrypt. Essa ferramenta espera que uma sequência correta de implementações de código seja seguida para atender às regras CrySL. O conhecimento

dessas regras deve ser observado para que as correções sejam realizadas com qualidade aceitável, atendendo tanto às exigências da ferramenta quanto aos requisitos de segurança.

Dos 67 usos indevidos de APIs de criptografia encontrados nos softwares selecionados para esta pesquisa, 34 encontravam-se no Componente 1. Este componente atua como uma arquitetura comum a vários sistemas e *webservices*, conforme descrito anteriormente na seção *Projetos de Software Alvo do Estudo de Caso*. Em uma análise preliminar, realizada antes da implementação das correções, constatou-se que a intervenção no componente em questão – seguida de sua substituição nos sistemas dependentes – possibilitaria que as inconsistências identificadas nos demais sistemas fossem resolvidas, senão integralmente, em grande parte mediante atualização do componente. Dessa forma, o Componente 1 foi selecionado como ponto inicial dos ajustes de código para correção das vulnerabilidades encontradas pelas ferramentas SAST.

Concluídos os ajustes necessários no Componente 1, todos os softwares que o utilizam como uma biblioteca interna foram atualizados para a versão corrigida, possibilitando, dessa forma, que o impacto gerado por essas mudanças no código fosse aferido.

Nesse processo, cada sistema que utiliza o Componente 1 foi testado novamente com as ferramentas SAST - CogniCrypt e CryptoGuard - não somente para verificar se os usos indevidos de APIs de criptografia, apontados pelas ferramentas SAST na primeira análise, foram eliminados, mas para verificar também se a simples substituição do Componente 1 corrigiu, de fato, o problema ou se outras intervenções no código de cada sistema seriam necessárias.

Com isso, foi possível verificar se as falhas foram eliminadas de fato, garantindo-se que as mudanças no código do componente não afetassem de forma negativa os softwares que o utilizam, gerando impactos nas suas funcionalidades.

Na adequação das implementações de APIs de criptografia consideradas inadequadas, adotou-se como premissa a manutenção da compatibilidade com os sistemas legados, de modo a evitar que eventuais incompatibilidades tornassem as correções tecnicamente inviáveis ou levassem à perpetuação das vulnerabilidades de segurança.

Nesse contexto, para corrigir o uso de algoritmo inseguro (vide "AES/ECB" na Tabela 4.2) - associado à CWE-327 - optou-se por substituí-lo por outro mais seguro, compatível com a versão do Java utilizada pela maioria dos sistemas. O algoritmo inseguro `AES/ECB/PKCS5Padding` era utilizado em dois métodos de uma classe do Componente 1, responsáveis por cifrar e decifrar dados recebidos em texto pleno.

Embora o modo GCM seja atualmente considerado a alternativa mais recomendada por combinar confidencialidade e integridade, sua adoção não foi viável no contexto deste estudo. Isso se deve ao fato de que a maioria dos sistemas avaliados executa sobre a versão Java 1.7, a qual não oferece suporte estável ao GCM. Diante dessa limitação, optou-se

pela utilização do modo CTR, que, apesar de não prover autenticação dos dados como o GCM, representa uma alternativa substancialmente mais segura em comparação ao ECB e mantém plena compatibilidade com o ambiente legado analisado. Essa decisão permitiu aplicar as correções necessárias sem comprometer a estabilidade e a interoperabilidade dos sistemas corporativos [17, 12, 14, 23].

O trecho de código 5.1 demonstra o método original de cifragem de dados implementado no Componente 1[1]. É possível observar que o método utiliza o algoritmo AES no modo ECB. Conforme detalhado na fundamentação teórica (Seção 2.3), este modo é inseguro pois processa blocos de forma independente, revelando padrões do texto original.

```
1     public static String encrypt(String strToEncrypt) {
2         try {
3             Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
4             final SecretKeySpec secretKey = new SecretKeySpec(key, "AES");
5             cipher.init(Cipher.ENCRYPT_MODE, secretKey);
6             final String encryptedString =
              ↪  Base64.encodeBase64String(cipher.doFinal(strToEncrypt.getBytes()));
7             return encryptedString;
8         } catch (Exception e) {
9             log.error("Erro durante a criptografia", e);
10        }
11        return null;
12
13    }
```

Código 5.1: Método encrypt com algoritmo inadequado. (Fonte: Própria)

Ainda é possível observar no trecho de código 5.1, a falta de implementação de boas práticas de segurança, como o uso de IV, tornando o processo da criptografia, ao utilizar o método `encrypt` nessa versão do código, ainda mais previsível. Isso ocorre porque, a cada vez que a criptografia é realizada com a mesma chave, serão gerados os mesmos blocos de saída para os mesmos blocos de entrada.

Já o trecho de código 5.2 mostra trecho do método `encrypt` já com as correções sugeridas pelas ferramentas de análise estática CryptoGuard e CogniCrypt. É possível observar a alteração modo de operação do algoritmo de criptografia AES de ECB para CTR. Esse é um dos modos de operação sugeridos no relatório do CogniCrypt, como é possível ver no trecho de código 5.3.

Essa implementação materializa a correção da vulnerabilidade, uma vez que o modo CTR, diferentemente do ECB, elimina a previsibilidade de padrões e viabiliza o processamento paralelo. Esses conceitos, detalhados na Seção 2.3, fundamentam a segurança obtida. A Figura 2.1, apresentada no Capítulo 2, ilustra visualmente essa eficácia ao comparar os resultados da cifragem em ambos os modos.

[1]Por razões de segurança e confidencialidade, os nomes de pacotes, classes e métodos apresentados neste e nos demais trechos de código foram anonimizados, preservando-se a estrutura lógica original.

```
1       Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
2       IvParameterSpec ivSpec = new IvParameterSpec(iv);
3       cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);
```

Código 5.2: Trecho do método encrypt com algoritmo correto. (Fonte: Própria)

```
{
    "fileLocation": {
        "uri": "org/instituicao/componente1/seguranca/BaseCripto.java"
    },
    "fullyQualifiedLogicalName": "org::instituicao::componente1::seguranca::BaseCripto::encrypt",
    "ruleId": "ConstraintError",
    "message": {
        "text": "First parameter (\"AES/ECB/PKCS5Padding\") should be AES/{CBC, GCM, PCBC, CTR, CTS,
        ↪  CFB, OFB}.",
        "richText" : "ConstraintError violating CrySL rule for javax.crypto.Cipher."
    }
}
```

Código 5.3: Modos de operação sugeridos pelo CogniCrypt. (Fonte: Própria)

Após a realização do ajuste do modo de operação no método `encrypt` do Componente 1, executou-se novamente as ferramentas de análise estática CogniCrypt e CryptoGuard para validar os ajustes e verificar se o uso indevido reportado como **Uso de algoritmo inseguro** ainda ocorria ou se teria sido sanado. Como resultado, as duas ferramentas não mais indicaram esse uso inadequado em seus relatórios. Para que esse resultado fosse possível, ajustou-se também o modo de operação no método `decrypt`.

Em seguida, as correções concentraram-se no uso inadequado apontado nos relatórios das ferramentas SAST como Geração incorreta de chave (vide "KeyGen" na Tabela 4.2) - CWE-320, ainda no Componente 1. Para esta correção, seguir as regras CrySL foi essencial, uma vez que não seguir as orientações lá descritas pode vir a acrescentar mais apontamentos nos relatórios do CogniCrypt. Ao revisar o relatório do CogniCrypt e confrontar com as regras CrySL, observou-se que a vulnerabilidade: Uso de chave constante no código (vide "ConstKey" na Tabela 4.3) - CWE-547 - poderia, também, ser corrigida em conjunto.

Dessa forma, a atividade de correção dessas duas vulnerabilidades relatadas pelas ferramentas SAST foi realizada ainda buscando-se o mínimo de impacto nos sistemas que utilizam o Componente 1.

É possível observar no trecho de código 5.4, com vulnerabilidades reportadas pelas ferramentas SAST, que a chave para criptografia é gerada de forma insegura como constante no código-fonte. Além disso, a classe `SecretKeySpec` espera que mais parâmetros sejam informados, como o tamanho da chave em bits, um número gerado aleatoriamente e uma senha forte para que a criptografia seja realizada de forma segura. Porém, esses

dados não estão sendo informados no construtor da classe.

```
1  private static byte[] key = { 0x74, 0x68, 0x69, 0x73, 0x49, 0x73, 0x41, 0x53, 0x65, 0x63, 0x72, 0x65,
   ↪  0x74, 0x4b, 0x65, 0x79 };
2
3  public static String encrypt(String strToEncrypt) {
4      try {
5          Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
6          final SecretKeySpec secretKey = new SecretKeySpec(key, "AES");
7          cipher.init(Cipher.ENCRYPT_MODE, secretKey);
8          //...
```

Código 5.4: Geração incorreta de chave e utilizando constante. (Fonte: Própria)

Como ilustrado no trecho de código 5.4, a forma de geração de chaves originalmente utilizava valores constantes no código, prática essa inadequada, como apontado pelas ferramentas SAST utilizadas nesta pesquisa. Para corrigir esse problema, implementou-se um método privado auxiliar para gerar uma chave segura a partir de uma senha fornecida como um *array* de *char* (`char[]`) e um número aleatório (*salt*).

```
1      private static final int IV_SIZE = 16;
2      private static final int SALT_SIZE = 16;
3      private static final int KEY_LENGTH = 128; // Tamanho da chave em bits
4      private static final int ITERATIONS = 65536; // Iterações PBKDF2
```

Código 5.5: Constantes para métodos auxiliares. (Fonte: Própria)

No trecho de código 5.5, são apresentadas as constantes utilizadas como base para os métodos auxiliares, incluindo o valor padrão para iterações (`ITERATIONS`) e o tamanho da chave (`KEY_LENGTH`). Para obter esses parâmetros de senha e *salt*, foram criados métodos específicos que geram esses valores de forma segura com a utilização de `SecureRandom` [14] para garantir, assim, a aleatoriedade e, consequentemente, a segurança do processo.

Além desses dois parâmetros (senha e *salt*), dois outros são utilizados na geração da chave:

- **Key Length:** O tamanho da chave em bits (neste caso, 128 bits);
- **Iterations:** o número de iterações do PBKDF2 (neste caso, 65536) [17].

Conforme demonstrado no trecho de código 5.6, a derivação da chave faz o uso do algoritmo `PBKDF2WithHmacSHA256` [17], algoritmo esse amplamente adotado para realizar a derivação de chaves a partir de senhas fornecidas. Esse procedimento faz a utilização repetida de uma função *hash* (neste caso, `SHA-256` em conjunto a `HMAC`), o que dificulta ataques de força bruta ou de dicionário, principalmente quando se faz uso em conjunto

com um *salt* aleatório e um número elevado de iterações [4]. Na implementação de forma correta, o método `deriveKey` recebe como parâmetro a senha, o *salt*, o número de iterações e o tamanho da chave e retorna uma instância de `SecretKeySpec`[2] do tipo `AES` [14].

```
// Método para gerar uma chave segura a partir da senha fornecida e do salt.
private static SecretKeySpec deriveKey(char[] password, byte[] salt) throws NoSuchAlgorithmException,
    InvalidKeySpecException {
    PBEKeySpec spec = new PBEKeySpec(password, salt, ITERATIONS, KEY_LENGTH);
    SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
    byte[] keyBytes = factory.generateSecret(spec).getEncoded();
    spec.clearPassword();
    return new SecretKeySpec(keyBytes, "AES");
}
```

Código 5.6: Geração correta de chave. (Fonte: Própria)

Além disso, como um reforço adicional à segurança, foi desenvolvido um método para a geração de IV (Trecho de código 5.7), utilizando `SecureRandom` [14]. Utilizou-se o algoritmo `SHA1PRNG`, um CSPRNG adequado para evitar a previsibilidade de valores, mitigando os riscos associados a geradores lineares comuns (ver Seção 2.3).

```
// Método para gerar um IV seguro aleatório.
private static byte[] generateIv() throws NoSuchAlgorithmException {
    byte[] iv = new byte[IV_SIZE];
    SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
    secureRandom.nextBytes(iv);
    return iv;
}
```

Código 5.7: Geração de IV. (Fonte: Própria)

O tamanho do IV (IV_SIZE) foi definido como 16 bytes (128 bits), seguindo, dessa forma, a recomendação para a criação de cifras com AES no modo que exige IV (AES em modo CTR ou GCM) [17]. Essa implementação reforça a segurança do esquema de criptografia, uma vez que, para cada operação de criptografia de dados, passa a ter um IV aleatório único [4].

O trecho de código 5.8 apresenta a implementação completa do método `encrypt`, que realiza a cifragem de uma string que é passada como parâmetro, utilizando o algoritmo de criptografia AES no modo CTR. O processo de criptografia é iniciado com a obtenção da senha no formato `char[]` através do método auxiliar `getPassword()`, garantindo que a chave seja utilizada de forma segura. Após isso, a senha é validada, lançando uma exceção caso não tenha sido fornecida. Em seguida, dois valores aleatórios são gerados: o

[2] `SecretKeySpec` é uma classe da API de segurança do Java (JCA) utilizada para representar chaves secretas em algoritmos simétricos. Aqui, ela é empregada para encapsular a chave derivada e torná-la compatível com implementações de cifradores como o AES, conforme descrito por Paar e Pelzl [14].

```java
// Método de criptografia (mantendo a assinatura original).
public static String encrypt(String strToEncrypt) {
    try {
        // Obtém a senha como um char[].
        char[] password = getPassword();
        if (password == null || password.length == 0) {
            throw new IllegalStateException("Senha para criptografia não fornecida.");
        }

        // Gera salt e IV aleatórios.
        byte[] salt = generateSalt();
        byte[] iv = generateIv();

        // Deriva a chave usando a senha e o salt.
        SecretKeySpec secretKey = deriveKey(password, salt);

        // Limpa a senha do array após uso.
        java.util.Arrays.fill(password, '\0');

        Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
        IvParameterSpec ivSpec = new IvParameterSpec(iv);
        cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);

        byte[] encrypted = cipher.doFinal(strToEncrypt.getBytes("UTF-8"));

        // Retorna o salt, o IV e o texto criptografado concatenados em Base64.
        return Base64.encodeBase64String(salt) + ":" +
               Base64.encodeBase64String(iv) + ":" +
               Base64.encodeBase64String(encrypted);
    } catch (Exception e) {
        // Tratamento de excessão
        log.error("Erro durante a criptografia.", e);
    }
    return null;
}
```

Código 5.8: Método encrypt corrigido completo. (Fonte: Própria)

salt (`generateSalt()` observado no trecho de código 5.9) e o vetor de inicialização (IV) (`generateIv()`), valores esses fundamentais para tornar todo o esquema de segurança criptográfico forte.

```java
// Método para gerar um salt seguro aleatório.
private static byte[] generateSalt() throws NoSuchAlgorithmException {
    byte[] salt = new byte[SALT_SIZE];
    SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
    secureRandom.nextBytes(salt);
    return salt;
}
```

Código 5.9: Geração de Salt. (Fonte: Própria)

Na sequência, a chave de criptografia é derivada a partir da senha e do *salt* através do método `deriveKey(password, salt)`, que utiliza PBKDF2 como forma de garantir a robustez da chave gerada. Para evitar exposição desnecessária da senha em memória, o conteúdo do `array` da senha é sobrescrito com `'\0'` logo depois da derivação da chave. É realizada, então, a configuração do algoritmo de criptografia utilizando

`Cipher.getInstance("AES/CTR/NoPadding")`, que define o uso de AES no modo CTR sem a necessidade de preenchimento adicional (*padding*). O IV gerado no método auxiliar é encapsulado em um objeto do tipo `IvParameterSpec`, e o método `cipher.init(Cipher .ENCRYPT_MODE, secretKey, ivSpec)` inicializa o processo de criptografia.

O texto de entrada do método `encrypt(strToEncrypt)` é então convertido para bytes no formato UFT-8 e passado para o método `cipher.doFinal()`, que realiza a execução da operação de cifragem. Os dados resultantes dessa operação incluem o texto criprtografado e o IV utilizado e que foi obtido com `cipher.getIV()`. Ao final, a saída do método é uma string contendo os valores do *salt*, do IV e do texto criptografado, todos estes codificados em Base64 e concatenados com `":"` como delimitador de valor. Em caso de erro em todo o processo, uma exceção é capturada e é gerado um log para verificação do erro.

Como é possível observar, a diferença entre o método `encrypt` original e o que passou por alterações para corrigir os maus usos de APIs de criptografia apontados pelas ferramentas SAST utilizadas nesta pesquisa (Trechos de código 5.1 e 5.8, respectivamente), é considerável. O método final aumentou de tamanho em relação a linhas de código, além de ter gerado quatro novos métodos de apoio. Porém, apesar de gerar mais código, o novo método `encrypt` tornou-se mais seguro e, ao ser analisado novamente pelas ferramentas CryptoGuard e CogniCrypt, não gerou mais alertas de segurança.

Para reverter o processo de cifragem, o método `decrypt`, demonstrado no trecho de código 5.10, é utilizado. Esse método segue, de forma geral, a mesma estrutura do processo de cifragem do método `encrypt`, porém com algumas diferenças que são fundamentais.

O processo de decifragem inicia obtendo-se a senha, armazenada em um `char[]` pelo método `getPassword()`. Da mesma forma como foi realizado no método `encrypt`, a senha sofre uma validação para garantir que não esteja vazia. Logo após, a string passada como parâmetro `strToDecrypt`, que contém os dados que serão decifrados, é divida em três partes utilizando-se `split(":")`. Essa etapa de separação em três partes é importante, visto que permite extrair de forma correta o valor do *salt*, do IV e do texto cifrado original. O parâmetro recebido no método `decrypt` é validado e, caso não possua três partes, uma exceção é lançada para informar que o formato de entrada do método está inválido.

A diferença principal entre os métodos `encrypt` e `decrypt` está na forma como usam o IV e como processam os dados. Enquanto no método `encrypt` o *salt* e o IV têm seus valores gerados aleatoriamente com o uso de `SecureRandom`, o método `decrypt` extrai esses valores da string de entrada. Isso é realizado dessa forma porque esses valores precisam ser exatamente os mesmos que foram utilizados no processo de cifragem, caso contrário, a reversibilidade do processo não pode ser realizada.

Outra diferença importante entre os métodos está na ordem em que realizam as operações internas. O método `encrypt` deriva a chave primeiro e só depois os dados são

```java
// Método de descriptografia (mantendo a assinatura original).
public static String decrypt(String strToDecrypt) {
    try {
        // Obtém a senha como um char[].
        char[] password = getPassword();
        if (password == null || password.length == 0) {
            throw new IllegalStateException("Senha para criptografia não fornecida.");
        }

        // Separa o salt, o IV e o texto criptografado.
        String[] parts = strToDecrypt.split(":");
        if (parts.length != 3) {
            throw new IllegalArgumentException("O texto a ser descriptografado está em um formato
            ↪  inválido.");
        }
        byte[] salt = Base64.decodeBase64(parts[0]);
        byte[] iv = Base64.decodeBase64(parts[1]);
        byte[] encryptedText = Base64.decodeBase64(parts[2]);

        // Deriva a chave usando a senha e o salt.
        SecretKeySpec secretKey = deriveKey(password, salt);

        // Limpa a senha do array após uso.
        java.util.Arrays.fill(password, '\0');

        Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
        IvParameterSpec ivSpec = new IvParameterSpec(iv);
        cipher.init(Cipher.DECRYPT_MODE, secretKey, ivSpec);

        byte[] decrypted = cipher.doFinal(encryptedText);

        return new String(decrypted, "UTF-8");
    } catch (Exception e) {
        // Tratamento de exceção
        log.error("Erro durante a decifragem.", e);
    }
    return null;
}
```

Código 5.10: Método decrypt corrigido. (Fonte: Própria)

passados para o `Cipher` processar. Já o método `decrypt`, antes de iniciar a derivação da chave, necessita separar os três componentes que integram a string cifrada (*salt*, IV e texto cifrado), lançando uma exceção caso o formato não seja válido.

Após a separação dos valores com `split(":")`, o *salt*, o IV e o texto cifrado são convertidos de `Base64` para `arrays` de bytes, utilizando-se para isso `Base64.decodeBase64()`. Após, a chave para a decifragem é derivada utilizando-se o mesmo método `deriveKey()` empregado no processo de cifragem, com os valores da senha e do *salt*. Com este passo, garante-se que a chave resultante seja idêntica à chave que foi utilizada na cifragem original.

Como um reforço de segurança adicional, o `array` utilizado para armazenar a senha é sobrescrito com `'\0'` após a derivação da chave, prevenindo, dessa forma, a exposição da senha em memória. Logo em seguida, configura-se o objeto `Cipher` com o algoritmo `AES/CTR/NoPadding`, garantindo, dessa forma, que o mesmo modo de operação utilizado para a cifragem dos dados seja utilizado no processo reverso. Na etapa seguinte, o vetor

de inicialização (IV) extraído da string passada por parâmetro, é encapsulado em um objeto do tipo `IvParameterSpec`, e o objeto `cipher` é, então, inicializado em modo de decifragem com o código `cipher.init(Cipher.DECRYPT_MODE, secretKey, ivSpec)`.

Após a inicialização do `cipher`, o texto a ser decifrado é passado para o método `cipher.doFinal()`, que executa de fato a operação de decifragem, o que resulta no texto original, porém em formato de *bytes*. Esse texto em *bytes* ainda tem que passar por mais uma etapa para que seja convertida de volta para uma string com a utilização da codificação UTF-8 e só então, enviado ao chamador do método `decrypt`. Caso alguma falha ocorra em algum momento do processo, captura-se uma exceção e registra-se um log para análise.

Com os ajustes nos métodos `decrypt` e `encrypt`, mais duas vulnerabilidades foram sanadas: Preparação incorreta de material de chave (vide "KeyMat" na Tabela 4.2) - CWE-321 e Constantes usadas como chaves (vide "ConstKey" na Tabela 4.3) - CWE-547. Dessa forma, toda a classe `BaseCriptografia.java` do Componente 1 foi corrigida. Ao analisar novamente o componente com as ferramentas CogniCrypt e CryptoGuard, constatou-se a eliminação dos alertas de segurança anteriormente indicados por essas ferramentas.

Na sequência, deu-se início à correção da vulnerabilidade relacionada ao Uso de protocolo SSL obsoleto (CWE-326), conforme indicado em "SSL" na Tabela 4.2). Essa vulnerabilidades foram identificadas em duas classes do Componente 1.

As classes `RecursoWS` e `ModuloLoginJAASWS` possuem ambas o método `sslCheck()`, com praticamente as mesmas implementações e que, sua principal função é realizar a configuração de conexões seguras em nível de transporte utilizando SSL/*Transport Layer Security* (TLS).

Apesar de possuírem implementações parecidas, cada uma dessas classes têm um papel distinto a desempenhar: a `RecursoWS` atua no consumo de recursos e serviços *Representational State Transfer* (REST), enquanto a classe `ModuloLoginJAASWS` tem a sua atuação como responsável pela lógica de autenticação no contexto *Java Authentication and Authorization Service* (JAAS). Mesmo com atuações distintas, torna-se necessário o ajuste na forma como o método `sslCheck()` das duas classes estabelece o protocolo de segurança.

O trecho de código 5.11 demonstra o código original do método `sslCheck()` na classe `RecursoWS`, cuja implementação é idêntica à do método homônimo na classe ModuloLoginJAASWS. Diante da verificação dessa duplicidade, cogitou-se unificar ambos em um único método e em uma única classe utilitária, de forma a manter a lógica centralizada e reutilizá-la nos momentos em que fosse necessário. Porém, conforme foi mencionado anteriormente, no início processo de correção dos usos inadequados de APIs de criptografia, optou-se por realizar apenas os ajustes necessários para corrigir as vulnerabilidades,

```java
private void sslCheck() {
  try {
    TrustManager[] trustAllCerts = new TrustManager[] { new X509TrustManager() {
      public java.security.cert.X509Certificate[] getAcceptedIssuers() { return null; }
      public void checkClientTrusted(X509Certificate[] certs, String authType) { }
      public void checkServerTrusted(X509Certificate[] certs, String authType) { }
    }};

    SSLContext sc = SSLContext.getInstance("SSL");
    sc.init(null, trustAllCerts, new java.security.SecureRandom());
    HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

    HostnameVerifier allHostsValid = new HostnameVerifier() {
      public boolean verify(String hostname, SSLSession session) { return true; }
    };
    HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);
  } catch (NoSuchAlgorithmException | KeyManagementException e) {
    // ...
  }
}
```

Código 5.11: Método sslCheck() com implementação inadequada. (Fonte: Própria)

buscando-se minimizar o impacto nos sistemas legados que utilizam o Componente 1.

```
{
    "locations": [{
        "physicalLocation": {
            "fileLocation": {
            "uri": "org/instituicao/componente1/core/webservice/resources/RecursoWS.java"
            },
            "region": { "method": "void sslCheck()" }
        },
        "fullyQualifiedLogicalName":
            "org::instituicao::componente1::core::webservice::resources::RecursoWS::sslCheck"
    }],
    "ruleId": "ConstraintError",
    "message": {
    "text":
        "First parameter (with value \"SSL\") should be any of {TLSv1.2, TLSv1.3}.",
    "richText":
        "ConstraintError violating CrySL rule for javax.net.ssl.SSLContext."
    }
}
```

Código 5.12: CogniCrypt identificando falha no método `sslCheck()` da classe `RecursoWS`. (Fonte: Própria)

Nos Trechos de código 5.12 e 5.13 é possível observar que, visto que o método sslCheck() é idêntico nas duas classes, o CogniCrypt gerou o mesmo alerta no relatório.

Ao observar o código original do método `sslCheck()`, nota-se que o contexto do SSL é inicializado com `SSLContext sslCtx = SSLContext.getInstance("SSL")`. Essa implementação genérica é crítica pois permite a negociação via protocolos inseguros (como SSLv2/v3) e não impede o uso de *ciphersuites* fracas, expondo o sistema aos riscos de interceptação e ataques de *downgrade* discutidos na Seção 2.3.

Em seguida, é possível observar também que o tratamento de verificação dos certificados se utiliza de um `TrustManager` definido de forma permissiva. Tal configuração aceita qualquer certificado, falhando em garantir a autenticidade da fonte e facilitando ataques *man-in-the-middle* (detalhados na Seção 2.3).

```
{
    "locations": [{
        "physicalLocation": {
          "fileLocation": {
            "uri": "org/instituicao/componente1/seguranca/jaas/ModuloLoginJAASWS.java"
          },
          "region": { "method": "void sslCheck()" }
        },
        "fullyQualifiedLogicalName":
          "org::instituicao::componente1::seguranca::jaas::ModuloLoginJAASWS::sslCheck"
    }],
    "ruleId": "ConstraintError",
    "message": {
    "text":
        "First parameter (with value \"SSL\") should be any of {TLSv1.2, TLSv1.3}.",
    "richText":
        "ConstraintError violating CrySL rule for javax.net.ssl.SSLContext."
    }
}
```

Código 5.13: CogniCrypt identificando falha no método `sslCheck()` da classe `ModuloLoginJAASWS`. (Fonte: Própria)

Finalizando a análise do código original do método `sslCheck()`, a ausência de verificação de *hostname* é percebida. Como visto na Seção 2.3, a ausência de um *Hostname-Verifier* rigoroso permite que certificados válidos, porém emitidos para outros domínios, sejam aceitos indevidamente.

Como forma de mitigar essas vulnerabilidades detectadas, foram realizadas correções na implementação do método `sslCheck()`, buscando-se garantir a utilização do protocolo SSL/TLS de forma segura, seguindo princípios fundamentais recomendados pela literatura acadêmica e pela indústria no que se refere à segurança de sistemas.

Nesse sentido, os ajustes no código tiveram início ao seguir boas práticas identificadas na literatura, onde trabalhos como os de Focardi et al. [42] ressaltam a importância da escolha adequada e explícita do tipo de `KeyStore` utilizado, e estudos como os de Meng et al. [43] evidenciam que o uso incorreto ou a configuração permissiva de componentes como o `TrustManagerFactory` pode comprometer a segurança da comunicação TLS. Dessa forma, no novo código buscou-se definir de forma explícita o armazenamento de chaves e o gerenciador de confiança, fortalecendo a proteção da conexão.

Utilizou-se o tipo *Java KeyStore* (JKS), formato padrão da linguagem de programação Java até a versão 8, para o armazenamento de certificados e chaves privadas [42]:

```
KeyStore generatedKeyStore = KeyStore.getInstance("JKS");
```

Já para a verificação correta dos certificados do servidor através de um `TrustManager`, definiu-se de forma explícita o algoritmo `SunX509` para o `TrustManagerFactory`, garantindo que a validação siga o processo padrão do Java para certificados X.509 e que somente certificados confiáveis sejam aceitos, em contraposição às implementações permissivas de `TrustManager` criticadas na literatura [43]:

```
TrustManagerFactory trustManagerFactory = TrustManagerFactory⌋
↪   .getInstance("SunX509");
```

Dessa forma, o novo código alinha-se com as recomendações de Schneier [4], onde o autor destaca que a validação criteriosa da cadeia de certificados é importante na prevenção de ataques de *man-in-the-middle*.

De forma similar, o código do método `sslCHeck()` corrigido(vide Trecho de código 5.14), inicializa um `KeyManagerFactory` também utilizando o algoritmo `SunX509` e utilizando o `keystore` criado anteriormente:

```
KeyManagerFactory keyManagerFactory = KeyManagerFactory⌋
↪   .getInstance("SunX509");
```

Ainda no novo código, foram realizadas correções substituindo a chamada SSL por `SSLContext tlsCtx = SSLContext.getInstance("TLSv1.2")`, garantindo que versões obsoletas do SSL seja excluídas. Essa alteração restringe as *ciphersuites* a algoritmos robustos, prevenindo os ataques de *version-rollback* citados na fundamentação teórica.

Logo após isso, verifica-se a configuração de `TrustManagers` que realizam a validação da cadeia de certificados, de forma que a instância que tem a responsabilidade dessa validação confirma emissor, período de validade e se o certificado não está presente em alguma lista de revogação. Caso ocorra alguma falha na validação, o código registra o erro e também interrompe de forma imediata o fluxo, aderindo ao princípio (discutido por Schneier [4]) de que, diante da impossibilidade de estabelecer uma conexão segura, a sessão deve ser finalizada antes de prosseguir sob condições sem a devida segurança.

Já a vulnerabilidade crítica identificada pelas ferramentas CogniCrypt e CryptoGuard envolvia um `HostnameVerifier` configurado de maneira excessivamente permissiva, aceitando automaticamente qualquer nome de host foi corrigida removendo completamente `HostnameVerifier` permissivo. Dessa forma, adotou-se o comportamento padrão da classe `HttpsURLConnection` do Java, realizando automaticamente a verificação rigorosa da correspondência entre o nome do host informado no certificado digital e aquele requisitado pela aplicação cliente. Esse procedimento de validação realizada de forma implícita pela plataforma Java, garante que quaisquer divergências na validação do nome resultem na interrupção imediata da conexão, evitando assim ataques do tipo *man-in-the-middle*, conforme as diretrizes defendidas por Howard e LeBlanc [25].

```java
// Método sslCheck() corrigido.
private void sslCheck() {
    try {
        Variaveis _vSSLCheck = daoComponente1.consultarVariaveisPorSiglaNome("wso2", "sslCheck");

        if (_vSSLCheck.getValor().equals("S")) {

            // Configuração do tipo do KeyStore.
            KeyStore generatedKeyStore = KeyStore.getInstance("JKS");
            try (InputStream is = new FileInputStream(getKeystorePath())) {
                generatedKeyStore.load(is, getKeystorePassword());
            }

            // Configuração do algoritmo do TrustManagerFactory.
            TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("SunX509");
            trustManagerFactory.init(generatedKeyStore);
            TrustManager[] generatedTrustManagers = trustManagerFactory.getTrustManagers();

            // Configuração do algoritmo do KeyManagerFactory.
            KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance("SunX509");
            keyManagerFactory.init(generatedKeyStore, getKeystorePassword());
            KeyManager[] generatedKeyManagers = keyManagerFactory.getKeyManagers();

            // Configuração do SSLContext com protocolo compatível.
            SSLContext sslContext = SSLContext.getInstance("TLSv1.2");
            sslContext.init(generatedKeyManagers, generatedTrustManagers, new SecureRandom());

            // Configuração da SSLSocketFactory personalizada que força o uso do TLSv1.2.
            HttpsURLConnection.setDefaultSSLSocketFactory(
              new TLSSocketFactory(sslContext.getSocketFactory()));

            // Limpa o array de senha após o uso.
            Arrays.fill(getKeystorePassword(), '\0');
        }

    } catch (NoSuchAlgorithmException | KeyManagementException | UnrecoverableKeyException |
            KeyStoreException | IOException | CertificateException e) {
        log.error(e);
        // ...
    }
}
```

Código 5.14: Método `sslCheck()` corrigido com uso do protocolo `TLSv1.2` e validação adequada de certificados. (Fonte: Própria)

Dessa forma, com a eliminação do `HostnameVerifier` inseguro, o método `sslCheck()` passou a oferecer garantias robustas quanto à autenticidade do servidor remoto, preservando integralmente as propriedades essenciais de confidencialidade e integridade requeridas pelas boas práticas de programação segura.

Ao findar os ajustes no método `sslCheck()` nas classes RecursoWS e ModuloLoginJAASWS, quatro vulnerabilidades foram corrigidas: Uso de protocolo SSL obsoleto (vide "SSL" na Tabela 4.2) - CWE-326, Geração incorreta de TrustManagers (vide "TrustM" na Tabela 4.2) - CWE-295, Uso de HostNameVerifier não confiável (vide "HostNameVer" na Tabela 4.3) - CWE-601 e Uso de TrustManager não confiável (vide "TrustManAll" na Tabela 4.3) - CWE-349. Também as duas classes `RecursoWS` e `ModuloLoginJAASWS` passam a estar corrigidas de forma completa neste ponto.

Neste momento das correções, destaca-se a adequação relacionada ao uso de *Pseudo-Random Number Generator* (Gerador de Números Pseudoaleatórios – PRNG) não confiáveis (vide "PRNG" na Tabela 4.3), associada à CWE-338. Essa fragilidade, constatada em duas classes do Componente 1 e também no Componente 2, decorria da utilização da classe `java.util.Random`. A mitigação consistiu na substituição dessa classe por `java.security.SecureRandom`, projetada especificamente para fins criptográficos. Conforme fundamentado na Seção 2.3, essa mudança é imprescindível para assegurar a aleatoriedade e a imprevisibilidade dos valores gerados, garantindo que ambos os componentes estejam em conformidade com as boas práticas de segurança para sistemas corporativos sensíveis.

Ao finalizar as correções das vulnerabilidades até este momento e analisar novamente o Componente 1 com as ferramentas CryptoGuard e CogniCrypt, nenhum alerta de segurança identificado anteriormente foi gerado. Contudo, um novo alerta foi emitido pelo CogniCrypt, conforme ilustrado no Trecho de código 5.15. Este fato merece destaque: diferentemente da análise inicial, na qual todos os 67 apontamentos corresponderam a vulnerabilidades reais — contrariando a alta taxa de falsos positivos comumente citada na literatura —, este novo alerta constitui a única ocorrência de falso positivo em todo o estudo. Ele surge não do código legado original, mas como um efeito colateral da própria estratégia de correção ao adotar o modo CTR. O relatório indica que o vetor de inicialização (IV) não teria sido gerado de forma aleatória.

```
{
  "violatedRule": "javax.crypto.spec.IvParameterSpec",
  "errorType": "RequiredPredicateError",
  "locations": [{
    "physicalLocation": {
      "fileLocation": {
        "uri": "org/instituicao/componente1/seguranca/BaseCripto.java"
      },
      "region": {
        "method": "java.lang.String decrypt(java.lang.String)"
      }
    },
    "fullyQualifiedLogicalName":
      "org::instituicao::componente1::seguranca::BaseCripto::decrypt"
  }],
  "message": {
    "text": "First parameter was not properly generated as randomized",
    "richText":
      "RequiredPredicateError violating CrySL rule for javax.crypto.spec.IvParameterSpec"
  }
}
```

Código 5.15: CogniCrypt informando vulnerabilidade no método `decrypt`. Neste caso, um falso positivo. (Fonte: Própria)

No processo de cifragem, o IV deve ser gerado de forma aleatória a fim de garantir que

resultado produzido seja único. Já na etapa de decifragem, é essencial que o mesmo IV seja utilizado, obtido a partir da string cifrada, para possibilitar a correta recuperação dos dados. Como discutido em [14] e por Ferguson e Schneier [17], o reuso do IV nesse contexto não constitui vulnerabilidade, mas sim requisito do modo CTR. Dessa forma, o alerta emitido pelo CogniCrypt deve ser interpretado como um falso positivo, não configurando uma falha de segurança relevante no escopo desta pesquisa.

Com a conclusão das correções realizadas, todas as vulnerabilidades identificadas no Componente 1, assim como a vulnerabilidade relacionada ao uso de PRNG inseguro no Componente 2, foram devidamente corrigidas com base nas análises das ferramentas CogniCrypt e CryptoGuard [35]. A única pendência observada refere-se a um falso positivo no Componente 1, relacionado ao uso do vetor de inicialização (IV) no modo CTR, já discutido anteriormente. As correções aplicadas no código demonstraram que as práticas de mitigação seguidas garantiram maior robustez criptográfica e reduziram os pontos de exposição dos sistemas analisados.

Para assegurar a integridade e a rastreabilidade das alterações realizadas, definiu-se uma estratégia de controle de versão adaptada ao ambiente de desenvolvimento legado da Embrapa. Nesse contexto, a rastreabilidade de cada conjunto de correções foi garantida por meio da geração de *Tags* específicas no repositório de código. Cada *Tag* criada recebeu comentários detalhados descrevendo a vulnerabilidade tratada e o trecho de código modificado, permitindo a auditoria futura e a identificação precisa das versões corrigidas antes de sua integração aos sistemas finais.

A Tabela 5.1 apresenta o volume de alterações resultante do processo de correção manual, consolidado por componente. Observa-se que o Componente 1 concentrou a maior parte do esforço de manutenção, totalizando 685 linhas modificadas, enquanto o Componente 2 apresentou 339 alterações, resultando em um total geral de 1.024 linhas. Os detalhes das alterações por classe encontram-se no Apêndice A.1.

Para os Sistema 2, Sistema 3 e Sistema 6, assim como para as demais vulnerabilidades ainda presentes no Componente 2, o processo de correção seguiu a mesma lógica adotada nas vulnerabilidades do Componente 1, envolvendo ajustes no uso adequado de algoritmos, parâmetros criptográficos e geração de números aleatórios. Todo o código resultante das correções manuais, juntamente com o código original e as sugestões de correção geradas por SLMs e LLMs, encontra-se nos apêndices, possibilitando comparação direta entre as versões e maior transparência metodológica.

Além dos ajustes diretos no código dos componentes, foi avaliada a integração dos módulos corrigidos nos sistemas corporativos integrantes dessa pesquisa, de modo a verificar os efeitos práticos da substituição no contexto real de uso.

Tabela 5.1: Resumo do Volume de Alterações — Correção manual. (Fonte: Própria)

Escopo	Total
Componente 1	685
Componente 2	339
Total geral	1024

5.1.2 Integração dos componentes corrigidos nos sistemas

Ao inserir os Componentes 1 e 2 corrigidos nos sistemas 2, 3 e 6, observou-se que as vulnerabilidades apontadas pelas ferramentas SAST nos sistemas 2 e 3 foram completamente sanadas, sem a necessidade de alteração no código-fonte desses sistemas. No entanto, no Sistema 6 as ferramentas continuaram a reportar vulnerabilidades. Uma análise mais detalhada revelou a existência de duas classes internas que replicavam funcionalidades já presentes no Componente 1. Essa duplicação possivelmente ocorreu em função do histórico de desenvolvimento: o Sistema 6 foi inicialmente produzido em uma fábrica de software e, posteriormente, incorporado pela equipe interna da Embrapa. Assim, uma das equipes, sem pleno conhecimento da disponibilidade prévia das funcionalidades criptográficas no Componente 1, acabou por reimplementar ou incluir novamente essas funcionalidades no próprio sistema.

Diante dessa constatação, avaliou-se a melhor estratégia: corrigir diretamente o código interno do Sistema 6 ou eliminar a redundância e passar a utilizar as funcionalidades do Componente 1. Embora a premissa desta pesquisa tenha sido a de interferir minimamente no funcionamento dos sistemas legados, optou-se pela segunda alternativa, de modo a garantir maior consistência e padronização no uso das funcionalidades criptográficas. Para tanto, o Sistema 6 foi submetido a um período estendido de testes de homologação, com o objetivo de identificar eventuais impactos decorrentes da alteração. Após a conclusão dessa fase, não foram observados efeitos adversos, confirmando que o Sistema 6 pôde adotar plenamente as funcionalidades criptográficas do Componente 1.

Apesar do êxito alcançado na integração, o processo de correção manual também revelou desafios específicos, os quais são detalhados a seguir.

5.1.3 Dificuldades encontradas

Durante o processo de correção manual das vulnerabilidades apontadas pelas ferramentas de análise estática `CogniCrypt` e `CryptoGuard`, diversas dificuldades foram observadas, impactando diretamente o tempo e a complexidade das correções. A primeira dessas dificuldades relaciona-se à falta de experiência prévia dos desenvolvedores participantes deste estudo com o uso das ferramentas SAST `CogniCrypt` e `CryptoGuard`. Alguns possuíam

experiência apenas com ferramentas distintas, como o `SonarQube`, que não possuem a mesma ênfase em regras específicas para APIs de criptografia.

Essa limitação é agravada pela dificuldade recorrente identificada na literatura quanto à compreensão e à aplicação corretas das APIs de criptografia por desenvolvedores de software de maneira geral, aspecto já evidenciado por Nadi et al. [2], em grande parte devido à sua complexidade e à falta de documentação clara.

Outra limitação enfrentada nesta pesquisa está relacionada à escassez de informações claras e detalhadas nos relatórios gerados pelas ferramentas. Em particular, os relatórios do `CogniCrypt` mostraram-se mais desafiadores, pois exigem que o desenvolvedor siga uma sequência de passos específica, conforme estabelecida nas regras `CrySL`, para que a implementação seja validada pela ferramenta e o alerta seja considerado resolvido. Além disso, a ausência de instruções mais diretas sobre o contexto em que o problema ocorre ou sobre o impacto da vulnerabilidade gerou incertezas, especialmente quando as mensagens se apresentavam de forma genérica ou vaga.

A dificuldade de compreensão do que o `CogniCrypt` solicitava em seus relatórios foi outro fator de atraso. Em um dos alertas, por exemplo, a mensagem reportava:

```
"First parameter was not properly generated as generatedKeyManagers."
```

Inicialmente, interpretou-se de maneira equivocada que o nome da variável utilizada como parâmetro deveria, obrigatoriamente, ser denominado `generatedKeyManagers`. Essa interpretação incorreta não ocorreu apenas nesse caso, mas se repetiu em diversas mensagens de vulnerabilidades emitidas pelo `CogniCrypt`, envolvendo diferentes métodos e classess.

Em todas essas situações, constatou-se um esforço prolongado na tentativa de adequar nomes de variáveis. Na realidade, a ferramenta indicava que os valores deveriam ser obtidos por meio de uma sequência bem definida de passos que garantisse sua geração apropriada. Somente após maior familiaridade com a lógica das regras `CrySL` compreendeu-se que o `CogniCrypt` não exige nomes específicos, mas sim que se observe de forma rigorosa a ordem de operações estabelecida. O não cumprimento de qualquer um desses passos ocasionava erros sucessivos nos passos seguintes, ampliando ainda mais o processo de ajuste do código-fonte para eliminar as vulnerabilidades relatadas.

Outro ponto crítico de dificuldade esteve relacionado ao falso positivo apontado pelo `CogniCrypt` em relação ao uso do IV no método `decrypt`. Embora a literatura especializada seja clara ao afirmar que, para decifrar dados, deve-se utilizar o mesmo IV empregado no processo de cifragem [17, 4, 18], a ferramenta gerava alertas de vulnerabilidade com mensagens pouco intuitivas. Esse cenário fez com que os desenvolvedores, principalmente aqueles com menor experiência em análise estática, investissem tempo significativo em tentativas de "corrigir" um problema inexistente.

Esse episódio representou o maior impacto em termos de esforço de investigação nesta pesquisa. Estima-se que foram necessários cerca de trinta dias de trabalho — envolvendo pesquisa bibliográfica, testes de refatoração e análise da documentação da ferramenta — até que a equipe pudesse concluir, com a devida segurança técnica, que o alerta se tratava de um falso positivo e que a implementação estava correta. Esse dado quantitativo ilustra a lacuna existente entre a semântica rigorosa das ferramentas, construídas por especialistas em segurança, e a realidade prática de desenvolvedores experientes na lógica de negócio, mas que não possuem especialização profunda em criptografia. O custo temporal elevado para refutar um falso positivo evidencia que a barreira de entrada para o uso eficaz dessas ferramentas em ambientes corporativos ainda é significativa.

Com essa fundamentação técnica consolidada, o alerta pôde, enfim, ser corretamente classificado como falso positivo nos resultados da ferramenta.

Diante dessas dificuldades, torna-se evidente que as ferramentas CogniCrypt e CryptoGuard, embora relevantes no apoio à detecção de vulnerabilidades criptográficas, ainda carecem de melhorias na forma como apresentam informações em seus relatórios. No caso do CogniCrypt, os achados deste estudo indicam oportunidades de aprimoramento particularmente relevantes, sobretudo na clareza das mensagens e no mecanismo de validação por sequência de passos. Esses elementos nem sempre fornecem contexto suficiente para orientar a correção e podem favorecer equívocos de interpretação. Mensagens mais detalhadas, contextuais e intuitivas tendem a reduzir o tempo de compreensão por parte dos desenvolvedores e a mitigar ambiguidades, contribuindo para um processo de correção mais eficaz.

5.1.4 Impactos das correções dos componentes nos sistemas

Após a implementação das correções manuais descritas na subseção *Correção das vulnerabilidades no código-fonte* – incluindo o tratamento do falso positivo relacionado ao IV no modo CTR –, torna-se necessário analisar os impactos dessas alterações sobre os sistemas e componentes avaliados. Essa análise contempla a estabilidade operacional, a compatibilidade com o ecossistema existente e a continuidade dos serviços durante e após a aplicação das mudanças.

A efetividade das correções foi verificada por meio de nova execução das ferramentas SAST (`CogniCrypt` e `CryptoGuard`), a qual confirmou a eliminação dos avisos anteriormente reportados. Em complemento, a construção dos artefatos de software foi concluída com sucesso e os testes funcionais foram realizados sem intercorrências, assegurando a consistência e a estabilidade do código atualizado.

No que se refere a efeitos colaterais, não foram observados impactos funcionais nem de desempenho decorrentes das alterações. O único impacto relevante identificado relaciona-

se à incompatibilidade entre o novo algoritmo de cifragem introduzido e os dados previamente armazenados utilizando o algoritmo anterior. Os dados cifrados sob a versão anterior do código não podem ser decifrados diretamente pela versão revisada.

Para mitigar essa incompatibilidade, definiu-se uma estratégia transitória baseada na operação simultânea das duas versões: o módulo legado permanece habilitado apenas para a decifragem de dados antigos, permitindo que estes sejam posteriormente reprocessados sob o novo padrão criptográfico. Em paralelo, foi estabelecida uma restrição que impede o módulo legado de realizar novas operações de cifragem, garantindo que toda a cifragem corrente e futura seja executada exclusivamente pela versão revisada. Além disso, preservaram-se as assinaturas dos métodos criptográficos, o que assegurou a compatibilidade em nível de API e evitou a necessidade de adaptações nos sistemas consumidores.

Essa estratégia produziu três resultados principais: (i) eliminação dos avisos reportados pelas ferramentas de análise estática, com exceção do falso positivo referente ao IV no método `decrypt`; (ii) preservação da estabilidade funcional e de desempenho; e (iii) transição controlada do acervo de dados para o novo padrão criptográfico, sem interrupções no consumo dos serviços. Concluído o processo de migração, prevê-se a desativação planejada do caminho legado de decifragem, consolidando o uso exclusivo da versão revisada.

Além dos resultados técnicos, a eficácia das correções foi verificada em cenário corporativo real. As versões corrigidas dos sistemas foram submetidas a um extenso e rigoroso período de homologação, durante o qual operaram sob simulação de carga real equivalente à do ambiente de produção. Essa etapa crítica confirmou a ausência de regressões e a estabilidade das novas implementações criptográficas. Como resultado direto dessa validação, o Sistema 2 já opera em ambiente de produção com as correções aplicadas, e o Sistema 3 encontra-se com sua implantação em produção agendada, consolidando a aplicação prática e industrial dos resultados desta pesquisa.

A consolidação dessa linha de base fornece os subsídios necessários para a etapa seguinte desta pesquisa, na qual são exploradas abordagens semiautomatizadas com modelos de linguagem, de pequeno e grande porte, para apoiar o processo de correção de vulnerabilidades.

5.2 Resultados das correções semiautomatizadas com SLMs e LLMs

A segunda etapa deste estudo 2 teve como objetivo investigar o uso de modelos de linguagem, de pequeno e grande porte (SLMs e LLMs), na proposição de correções para os usos inadequados de APIs de criptografia identificados pelas ferramentas de análise está-

tica CogniCrypt e o CryptoGuard. Essa etapa está diretamente associada à QP4, uma vez que busca avaliar em que medida tais modelos podem auxiliar os desenvolvedores na correção de vulnerabilidades, reduzindo esforço manual e mantendo a conformidade com boas práticas de segurança.

Para tanto, foram definidos os modelos empregados, os modos de execução adotados - local para os SLMs e por meio de serviços em nuvem para os LLMs - e a técnica de *one-shot prompting* utilizada. Além disso, estabeleceram-se critérios de avaliação que permitem comparar os resultados das correções sugeridas pelos modelos tanto sob a ótica da eliminação dos avisos reportados pelas ferramentas SAST quanto sob a perspectiva da validade funcional das soluções geradas.

A análise contempla, ainda, a investigação sobre a suficiência de uma única iteração por método vulnerável, bem como a identificação de erros recorrentes, limitações práticas e potenciais benefícios. Esses aspectos são discutidos ao longo desta seção, que também apresenta recomendações sobre a viabilidade de adoção das abordagens semiautomatizadas no ciclo de desenvolvimento seguro.

5.2.1 Modelos de linguagem utilizados

Foram selecionados três SLMs e três LLMs para este estudo. A escolha baseou-se em critérios de acessibilidade, estado da arte e representatividade entre modelos abertos e proprietários. Os modelos empregados foram: Qwen 3 4B [44], Mistral-7B [45], Gemma 3n E4B [46], Gemini 2.5 Pro [47], Claude Sonnet 4 [48] e ChatGPT 5 [49].

A literatura recente aponta que pequenos modelos, quando bem ajustados, podem ser competitivos em tarefas específicas de engenharia de software [50, 51]. Outros trabalhos reforçam a importância do uso de LLMs na detecção e correção de vulnerabilidades [52, 53, 54].

5.2.2 Execução dos modelos SLM

Os modelos SLMs (Qwen 3 4B, Mistral-7B e Gemma 3n E4B) foram executados localmente utilizando o software *LM Studio* [55]. Inicialmente, havia sido feita uma tentativa de execução com a versão do *Ollama* para Windows [56], que possui interface gráfica voltada à execução local de modelos de linguagem, porém o modelo Qwen apresentou desempenho extremamente lento — um dos prompts demandou exatamente 2623.1 segundos (43 minutos), conforme tempo exibido pelo próprio software na mensagem `"Thought for 2623.1 seconds"` —, tornando o uso inviável. No *LM Studio*, os SLMs rodaram em velocidade normal em um desktop equipado com processador Intel Xeon E5-2680 v4 @

2.40 GHz, 64 GB de RAM e GPU NVIDIA GeForce RTX 5060 Ti com 16 GB de VRAM, sob Windows 11 Pro.

Além da execução local, realizaram-se também tentativas via Hugging Face, nas quais observou-se que a plataforma aplica limites automáticos de taxa de uso (*rate limits*) na API de inferência, restringindo o número de requisições permitidas para usuários gratuitos. Essas restrições ocasionaram bloqueios temporários durante os testes e limitaram a continuidade dos experimentos. A própria documentação da plataforma afirma que, por ser um serviço gratuito, "há limites de taxa para usuários regulares (algumas centenas de requisições por hora)" [57][3].

De forma semelhante, o modelo Gemma 3n E4B foi inicialmente testado no site *aistudio.google.com*; porém, essa plataforma impõe restrições automáticas de taxa de uso (*rate limits*) [58], que limitam o número de requisições e o volume de tokens processados em determinado intervalo de tempo, o que interrompeu temporariamente a execução dos experimentos. Em função disso, optou-se pela migração para execução local via LM Studio, que se mostrou mais viável e adequada[4]. Essa abordagem garantiu que os experimentos fossem reproduzíveis e controlados em termos de desempenho, além de não dependerem de limitações ou variabilidades de plataformas externas.

5.2.3 Execução dos modelos LLM

Diferentemente dos SLMs, os modelos de grande porte (LLMs), como Gemini 2.5 Pro [47], Claude Sonnet 4 [48] e ChatGPT 5 [49], não puderam ser executados localmente devido ao seu tamanho e à natureza proprietária, que restringe a disponibilização dos pesos de treinamento. Em virtude dessas características, os experimentos foram conduzidos por meio das plataformas oficiais de cada fornecedor, em ambiente de nuvem.

O Gemini 2.5 Pro foi acessado em sua plataforma oficial [59], vinculada a uma conta com versão comercial, o que permitiu o uso do modelo sem as limitações impostas às versões gratuitas. O Claude Sonnet 4 e o ChatGPT 5 também foram utilizados em suas versões comerciais, disponibilizadas respectivamente pelas plataformas da Anthropic e da OpenAI. Esses modelos são fornecidos como serviços SaaS, de modo que tanto a execução quanto a gestão de recursos computacionais permanecem sob responsabilidade dos provedores.

[3]Embora a documentação da Hugging Face não apresente valores oficiais detalhados, menciona que o serviço gratuito de inferência impõe restrições automáticas de taxa de uso (*rate limits*) para evitar sobrecarga, limitando o número de requisições por hora e recomendando o uso de planos pagos para demandas mais intensas.

[4]O *LM Studio* [55] permite a execução local de modelos de linguagem sem restrições de taxa de uso ou dependência de conectividade com serviços externos, garantindo maior controle sobre os recursos computacionais e a reprodutibilidade dos experimentos.

Essa diferença estrutural em relação aos SLMs traz implicações importantes: por um lado, garante acesso a modelos de última geração sem a necessidade de infraestrutura local de grande porte; por outro, impõe dependência em relação a plataformas externas, além de custos financeiros associados às versões pagas e restrições de uso em versões gratuitas. Assim, enquanto os SLMs permitiram avaliar a viabilidade de execução em um ambiente controlado e economicamente acessível, os LLMs possibilitaram explorar cenários típicos de acesso por meio de serviços pagos ou de uso gratuito limitado, refletindo condições práticas de adoção em ambientes corporativos e acadêmicos.

5.2.4 Técnica *one-shot* nos prompts

Todas as interações seguiram a técnica de *one-shot prompting*, em que o modelo recebe uma instrução detalhada e um único exemplo de tarefa, sem iterações adicionais. Essa abordagem foi escolhida por possibilitar a avaliação da capacidade dos modelos em corrigir vulnerabilidades de forma direta, sem necessidade de múltiplas rodadas de refinamento. Além disso, a técnica *one-shot* facilita a padronização do experimento, permitindo maior reprodutibilidade e comparabilidade entre os diferentes modelos testados.

Estudos recentes apontam que estratégias simples de *prompting* podem ser eficazes para tarefas de reparo automático de código, ainda que com limitações quando comparadas a abordagens iterativas mais sofisticadas [60].

Trabalhos recentes também destacam o potencial de modelos de linguagem de diferentes portes (SLMs e LLMs) em tarefas de engenharia de software, demonstrando resultados promissores mesmo em cenários complexos, como a detecção de *refactoring bugs* em Java e Python, especialmente quando utilizadas abordagens de *prompting* diretas e controladas, semelhantes à técnica *one-shot* empregada neste estudo [61].

Além dessas vantagens experimentais, a adoção da técnica *one-shot* também foi estratégica para reduzir potenciais efeitos de não determinismo observados em modelos de linguagem de grande porte [62, 63, 64]. Conforme discutido na literatura, variações nas respostas podem ocorrer mesmo sob as mesmas instruções de entrada, em razão dos mecanismos de amostragem probabilística e da aleatoriedade inerente aos processos de decodificação [62, 63]. A utilização de um formato padronizado de *prompt*, com execução única por método analisado, contribuiu para mitigar esse efeito e garantir consistência nas condições experimentais, favorecendo a reprodutibilidade dos resultados obtidos.

Em plataformas comerciais como ChatGPT [49], Claude [48] e Gemini [47], os parâmetros de amostragem (*temperature*[5], *top-p*[6], *seed*[7]) são configuráveis apenas por meio de suas APIs de desenvolvedor, em chamadas realizadas programaticamente por aplicações que consomem a API e definem explicitamente os valores desses parâmetros. Tais controles não estão disponíveis nas interfaces web voltadas a usuários finais, utilizadas neste estudo, mesmo nas versões profissionais de assinatura.

O uso das versões web foi definido com base em critérios de padronização e adequação ao escopo da pesquisa. Buscou-se garantir uniformidade experimental entre os modelos avaliados (ChatGPT 5, Claude Sonnet 4 e Gemini 2.5 Pro), que, no momento da realização dos experimentos, correspondiam às versões estáveis e completas disponíveis aos assinantes. Como o estudo adotou a técnica *one-shot*, com uma única execução por caso de teste e sem amostragem repetida, o acesso via interface web mostrou-se suficiente para os objetivos propostos, não havendo necessidade de ajustar parâmetros de decodificação. Em experimentos com múltiplas execuções ou análises estatísticas de variância, o controle desses parâmetros seria desejável para garantir reprodutibilidade, conforme discutido na literatura [63, 62].

Dentro desse mesmo princípio de padronização, o *prompt* aplicado em cada execução foi cuidadosamente estruturado para manter consistência entre os modelos. Seu conteúdo assumia o papel de um especialista em segurança da informação, instruindo o modelo a analisar o código original, os relatórios do CogniCrypt e do CryptoGuard, bem como as regras CrySL, e produzir uma correção compatível com Java 1.7. Essa especificação em Java 1.7 se justifica pelo fato de que a maioria dos sistemas analisados nesta pesquisa foi desenvolvida nessa versão, a mais recente disponível nos sistemas corporativos desenvolvidos na GTI da Embrapa (Java 1.7.0_80) [35], conforme descrito na subseção 5.1.1. Um exemplo de comando utilizado é mostrado abaixo (sem inclusão do código completo nem dos relatórios de entrada):[8]

> *"Você é um especialista em segurança da informação, com foco em criptografia aplicada em Java e ampla experiência no uso de ferramentas de análise estática, incluindo CogniCrypt e CryptoGuard. Sua tarefa é atuar como revisor de código*

[5] O parâmetro *temperature* controla o grau de aleatoriedade na geração de texto: valores mais altos produzem respostas mais diversas e criativas, enquanto valores próximos de zero tornam a saída mais determinística.

[6] O parâmetro *top-p*, também conhecido como *nucleus sampling*, define a probabilidade acumulada máxima usada para selecionar as próximas palavras, restringindo a amostragem apenas aos tokens mais prováveis até que a soma de suas probabilidades alcance o valor especificado.

[7] O parâmetro *seed* define a semente aleatória usada para inicializar o gerador pseudoaleatório do modelo, permitindo reproduzir resultados idênticos quando todas as demais condições são mantidas constantes.

[8] *O prompt acima foi reproduzido exatamente como utilizado nos experimentos conduzidos nesta pesquisa, preservando sua formulação original para garantir reprodutibilidade.*

seguro. [...] Realize correções apenas nas violações reportadas nos relatórios. Utilize Java 1.7 para todas as correções. [...] Este é um cenário real de um sistema corporativo em produção."

5.2.5 Análise Quantitativa da Efetividade

A efetividade dos modelos na correção das vulnerabilidades foi avaliada em dois níveis distintos. O primeiro, denominado *Correção Aparente (SAST)*, quantifica o número de vulnerabilidades que deixaram de ser reportadas pelas ferramentas CogniCrypt e CryptoGuard após a aplicação do código sugerido pelo modelo de linguagem. Este nível mede a capacidade do modelo de realizar alterações sintáticas que satisfazem as regras das ferramentas. O segundo e mais rigoroso nível, a *Correção Funcional (Efetiva)*, considera uma vulnerabilidade como corrigida somente se a solução proposta, além de passar nas ferramentas SAST, também resultou em código compilável e funcional, sem quebrar a lógica de negócio existente.

As Tabelas 5.2 e 5.3 resumem os resultados agregados para os SLMs e LLMs, respectivamente. Elas apresentam o total de correções aparentes e a parcela destas que se provou funcionalmente efetiva, permitindo visualizar a discrepância entre a conformidade com as ferramentas de análise estática e a produção de uma solução viável.

Tabela 5.2: Análise Comparativa em Níveis para SLMs. (Fonte: Própria)

Modelo (SLM)	**Correção Aparente (SAST)**	**Correção Funcional (Efetiva)**
Qwen 3 4B	42	8
Mistral-7B	31	7
Gemma 3n E4B	32	8

Tabela 5.3: Análise Comparativa em Níveis para LLMs. (Fonte: Própria)

Modelo (LLM)	**Correção Aparente (SAST)**	**Correção Funcional (Efetiva)**
Gemini 2.5 Pro	15	13
Claude Sonnet 4	42	6
ChatGPT 5	48	23

Além da efetividade, quantificou-se o volume de alterações (LOC) por **modelo** e por **componente**. As Tabelas 5.4 e 5.5 sintetizam linhas Incluídas, Excluídas e o Total (Incluídas + Excluídas), distinguindo o *Tipo* (SLM/LLM).

Tabela 5.4: Volume de alterações por modelo — Correções com SLMs/LLMs (Componente 1). (Fonte: Própria)

Modelo	Tipo	Incluídas	Excluídas	Total
Qwen 3 4B	SLM	178	152	330
Mistral-7B	SLM	146	109	255
Gemma 3n E4B	SLM	120	122	242
Gemini 2.5 Pro	LLM	279	195	474
Claude Sonnet 4	LLM	854	169	1023
ChatGPT 5	LLM	380	182	562

Tabela 5.5: Volume de alterações por modelo — Correções com SLMs/LLMs (Componente 2). (Fonte: Própria)

Modelo	Tipo	Incluídas	Excluídas	Total
Qwen 3 4B	SLM	41	44	85
Mistral-7B	SLM	42	65	107
Gemma 3n E4B	SLM	45	42	87
Gemini 2.5 Pro	LLM	99	57	156
Claude Sonnet 4	LLM	220	100	320
ChatGPT 5	LLM	82	103	185

Para uma análise mais granular, que responde diretamente à questão de pesquisa QP4, as Tabelas 5.6 e 5.7 detalham o desempenho de cada modelo por tipo de vulnerabilidade, focando exclusivamente na métrica de Correção Efetiva.

A partir dos dados consolidados, é possível analisar a taxa de efetividade de cada modelo. A efetividade real é definida pela proporção de correções aparentes que se provaram funcionais, bem como pelo percentual de correções efetivas em relação ao total de 67 vulnerabilidades do escopo. Os resultados são os seguintes:

- **Qwen 3 4B:** Resolveu 42 vulnerabilidades aparentes, das quais 8 (19,0%) foram funcionais, resultando em uma efetividade total de 11,9%.
- **Mistral-7B:** Resolveu 31 vulnerabilidades aparentes, das quais 7 (22,6%) foram funcionais, resultando em uma efetividade total de 10,4%.
- **Gemma 3 E4B:** Resolveu 32 vulnerabilidades aparentes, das quais 8 (25,0%) foram funcionais, resultando em uma efetividade total de 11,9%.
- **Gemini 2.5 Pro:** Resolveu 15 vulnerabilidades aparentes, das quais 13 (86,7%) foram funcionais, resultando em uma efetividade total de 19,4%.

Tabela 5.6: Correção Efetiva (Funcional) por Tipo de Vulnerabilidade - SLMs. (Fonte: Própria)

Tipo de Uso Indevido	Qwen 3 4B	Mistral-7B	Gemma 3n E4B
Uso de algoritmo inseguro	0	0	0
Geração incorreta de chave	0	0	0
Preparação incorreta de material de chave	0	0	0
Uso de protocolo SSL obsoleto	2	0	2
Geração incorreta de TrustManagers	0	1	0
Uso de PRNG não confiável	5	5	5
Uso de chave constante no código	0	0	0
Uso de HostNameVerifier não confiável	0	0	0
Uso de TrustManager não confiável	0	0	0
Uso de protocolo HTTP inseguro	1	1	1
Total Corrigido Efetivamente	**8**	**7**	**8**

- **Claude Sonnet 4:** Resolveu 42 vulnerabilidades aparentes, das quais 6 (14,3%) foram funcionais, resultando em uma efetividade total de 9,0%.
- **ChatGPT 5:** Resolveu 48 vulnerabilidades aparentes, das quais 23 (47,9%) foram funcionais, resultando em uma efetividade total de 34,3%.

5.2.6 Avaliação da Suficiência de Uma Única Iteração por Método Vulnerável

Os experimentos foram realizados em nível de método: cada prompt submetido aos modelos buscava corrigir todas as violações reportadas para um determinado método Java dos componentes ou sistemas, sem fragmentação por vulnerabilidade. O objetivo foi verificar se uma única iteração (one-shot) seria suficiente para sanar múltiplas vulnerabilidades em um mesmo contexto.

Os resultados mostraram que, em diversos casos, a abordagem foi capaz de reduzir significativamente o número de violações. Contudo, nenhum dos modelos testados conseguiu atingir conformidade total com as regras CrySL do CogniCrypt em apenas uma iteração. A Tabela 5.8 sintetiza os resultados alcançados, indicando em quais situações as vulnerabilidades foram corrigidas, permaneceram ou mesmo aumentaram após a intervenção dos modelos.

Tabela 5.7: Correção Efetiva (Funcional) por Tipo de Vulnerabilidade - LLMs. (Fonte: Própria)

Tipo de Uso Indevido	Gemini 2.5 Pro	Claude Sonnet 4	ChatGPT 5
Uso de algoritmo inseguro	0	0	0
Geração incorreta de chave	0	0	0
Preparação incorreta de material de chave	0	0	0
Uso de protocolo SSL obsoleto	6	0	6
Geração incorreta de TrustManagers	1	0	0
Uso de PRNG não confiável	5	5	5
Uso de chave constante no código	0	0	0
Uso de HostNameVerifier não confiável	0	0	3
Uso de TrustManager não confiável	0	0	8
Uso de protocolo HTTP inseguro	1	1	1
Total Corrigido Efetivamente	**13**	**6**	**23**

5.2.7 Principais Erros Identificados

A análise detalhada das saídas dos modelos evidenciou alguns padrões recorrentes de falhas:

- **Ausência de salt e IV:** Nos SLMs, embora houvesse correção de usos indevidos de algoritmos inseguros (AES/ECB), os métodos de cifragem não concatenaram o *salt* e o *IV* ao texto cifrado, tornando as funções `decrypt` incapazes de recuperar o texto original.
- **Erros de tipagem e compilação:** Alguns modelos introduziram incompatibilidades de tipos (por exemplo, `X509Certificate`) ou erros de compilação em classes críticas como `ModuloLoginJAASWS`.
- **Violações persistentes de predicados CrySL:** Mesmo após correções, o CogniCrypt continuou reportando *RequiredPredicateError* e *ImpreciseValueExtraction-*

Tabela 5.8: Resumo dos resultados de correção com SLMs e LLMs. (Fonte: Própria)

Modelo	Tipo	Correções bem-sucedidas	Problemas persistentes/introduzidos
Qwen 3 4B	SLM	Substituição de algoritmo inseguro; remoção de chave constante; troca de PRNG	Não funcional: ausência de salt/IV; violações CrySL persistentes
Mistral-7B	SLM	Substituição de algoritmo inseguro; remoção de chave constante	Erros de compilação; ausência de salt/IV; novas violações CrySL
Gemma 3n E4B	SLM	Substituição de algoritmo inseguro; remoção de chave constante; PRNG seguro	Código não funcional; erros de compilação; violações CrySL não resolvidas
Gemini 2.5 Pro	LLM	Troca de SSL -> TLSv1.2; remoção de TrustManager e HostnameVerifier inseguros	Introdução de novas vulnerabilidades (PBKDF2 SHA-1, senhas hardcoded)
Claude Sonnet 4	LLM	Correção de protocolo; mitigação de PRNG inseguro	Persistência de violações em geração de chaves; regressão com senhas hardcoded
ChatGPT 5	LLM	Correção de protocolo; remoção de TrustManager e HostnameVerifier inseguros; PRNG seguro	Violações persistentes em KeyManagers/TrustManagers; erros de conformidade CrySL

Error, especialmente em cadeias de geração de `KeyStore`, `KeyManagerFactory` e `TrustManagerFactory`.

- **Quebra de funcionalidade:** Em alguns casos, o código gerado era aceito pelas ferramentas de análise estática, mas não executava corretamente em ambiente real.
- **Substituição inadequada de algoritmos:** Exemplo notável ocorreu no Gemini 2.5 Pro, que trocou AES/ECB por AES/CBC, introduzindo novas violações no CogniCrypt uma vez que esse algoritmo não consta na lista dos algoritmos aceitos nas regras CrySL.
- **Introdução de novas vulnerabilidades:** Alguns modelos introduziram segredos hardcoded por exemplo, senhas previsíveis), aumentando a quantidade de vulnerabilidades passíveis de ataque.

5.2.8 Impressões Gerais e Recomendações

De forma geral, os experimentos mostraram que:

- **LLMs** apresentaram maior capacidade de compreender o contexto e propor soluções mais completas, especialmente na substituição de protocolos inseguros e remoção de verificadores permissivos. Ainda assim, falharam em satisfazer integralmente as regras CrySL.
- **SLMs** foram eficazes em correções pontuais (por exemplo, PRNG), mas apresentaram mais quebras de compilação e perda de funcionalidade prática.
- A técnica **one-shot** foi suficiente para correções iniciais, mas insuficiente para garantir cobertura total, reforçando a necessidade de estratégias mais sofisticadas de *prompt engineering*.

Em síntese, os resultados desta seção evidenciam tanto o potencial quanto as limitações do uso de modelos de linguagem na correção semiautomatizada de vulnerabilidades, oferecendo insumos importantes para a reflexão apresentada no capítulo seguinte.

Capítulo 6

Considerações Finais

Este capítulo apresenta as conclusões gerais deste trabalho, sintetizando as principais contribuições teóricas e práticas, bem como as observações derivadas dos estudos realizados. Além disso, são discutidas as limitações encontradas, as possibilidades de continuidade da pesquisa e as sugestões de aprimoramento para investigações futuras.

6.1 Contribuições e Conclusões Gerais

Este trabalho analisou a segurança no uso de APIs de criptografia em sistemas corporativos desenvolvidos em Java na Embrapa, empregando as ferramentas de análise estática CogniCrypt e CryptoGuard. Os resultados demonstraram a prevalência de vulnerabilidades significativas — como o uso de algoritmos obsoletos, modos de operação inseguros e chaves constantes — em componentes reutilizados por diversos sistemas. As correções manuais aplicadas nos componentes revelaram ganhos expressivos em segurança sem impactar a funcionalidade dos sistemas, confirmando a efetividade das boas práticas criptográficas recomendadas pela literatura e pelas regras CrySL do CogniCrypt.

No contexto institucional, a pesquisa propiciou uma visão estruturada sobre as vulnerabilidades existentes e forneceu insumos para a criação de um processo contínuo de auditoria e melhoria de segurança em software na empresa. A adoção de métodos de análise estática para avaliar sistemas legados mostrou-se viável e valiosa, mesmo em ambientes com tecnologias heterogêneas e restrições de atualização. O grupo focal com os desenvolvedores confirmou que a falta de clareza nos relatórios SAST é um dos principais obstáculos para a correta interpretação dos alertas, sugerindo a necessidade de interfaces e formatos de saída mais didáticos.

No Estudo 2, a comparação entre as correções manuais e as geradas por modelos de linguagem (SLMs e LLMs) trouxe contribuições importantes para o estado da arte. Verificou-se que as correções produzidas pelos modelos ainda não são plenamente confiá-

veis para substituir o trabalho humano em contextos de segurança criptográfica. Apesar de alguns modelos apresentarem entendimento contextual razoável sobre a vulnerabilidade, os códigos gerados com frequência não compilavam, introduziam novos erros ou mantinham vulnerabilidades sutis. Houve casos em que as correções aparentemente válidas foram consideradas aceitas pelas ferramentas de análise estática, mas não eram funcionais em execução real, mostrando a necessidade de testes de validação dinâmica. Portanto, *até o momento da finalização deste trabalho, SLMs e LLMs não se mostram confiáveis para geração automática de código voltado à correção de vulnerabilidades criptográficas.*

Outro aspecto observado foi a ocorrência de falsos positivos, em especial em alertas relacionados ao vetor de inicialização (IV). Em um dos casos, o CogniCrypt sinalizou erro mesmo quando o IV era gerado corretamente de forma aleatória, e o contexto de uso garantia unicidade. Esse falso positivo reforça a importância da interpretação crítica dos relatórios e do entendimento da semântica do código pelo analista humano. Apesar disso, a taxa geral de alertas válidos permaneceu elevada, o que reforça a eficácia das ferramentas como suporte à auditoria de segurança.

De modo geral, as conclusões principais deste trabalho podem ser resumidas em quatro pontos:

1. Ferramentas de análise estática são essenciais para a identificação sistemática de usos inadequados de APIs de criptografia em sistemas corporativos;
2. Correções manuais guiadas por boas práticas e pelas regras CrySL são efetivas e mantêm a funcionalidade original dos sistemas;
3. Os SLMs e LLMs ainda carecem de maturidade para gerar código funcional e seguro sem intervenção humana;
4. A interpretação humana continua essencial para avaliar falsos positivos e para garantir a adequação das correções aos contextos reais de uso.

Assim, esta pesquisa contribui para o avanço do conhecimento sobre a aplicação de ferramentas SAST e modelos de linguagem na detecção e correção de vulnerabilidades criptográficas em ambientes corporativos reais, oferecendo subsídios tanto para a prática profissional quanto para novas linhas de pesquisa na área de segurança de software.

6.2 Trabalhos futuros

Esta seção apresenta as possibilidades de continuidade da pesquisa, formuladas a partir das limitações observadas e das oportunidades identificadas ao longo do desenvolvimento

deste trabalho. As propostas a seguir foram elaboradas com base nos resultados obtidos nos estudos realizados e refletem desdobramentos naturais das conclusões apresentadas, considerando sua viabilidade prática em ambientes acadêmicos e institucionais semelhantes ao contexto da Embrapa.

6.2.1 Normalização e enriquecimento dos relatórios SAST (CogniCrypt e CryptoGuard)

Os resultados deste estudo indicaram que a utilidade prática dos relatórios SAST é fortemente influenciada pela clareza das mensagens e pela facilidade de interpretação pelos desenvolvedores. Como evolução direta dessa constatação, propõe-se a criação de um modelo de normalização de alertas que unifique as saídas do CogniCrypt e do CryptoGuard em um formato padronizado.

Esse modelo poderia conter campos mínimos para apoiar a análise e a tomada de decisão, como: (i) identificador da regra e vínculo com CWE; (ii) localização precisa (arquivo, método e linha); (iii) alcance do código vulnerável a partir dos pontos de entrada; (iv) grau de severidade e confiança do alerta; (v) resumo técnico em linguagem acessível; (vi) roteiro de correção com trechos "antes/depois"; e (vii) identificação da origem (componente arquitetural, biblioteca ou código da aplicação).

A partir desse esquema, recomenda-se o desenvolvimento de um agregador que consuma e consolide relatórios das duas ferramentas, eliminando duplicidades, classificando por severidade e gerando saídas unificadas por sistema (em HTML, Markdown ou PDF), além de permitir integração com sistemas de gestão de *issues*. Como evolução natural, a disponibilização desse agregador como extensão de IDE (por exemplo, VS Code) pode aproximar os alertas do contexto de trabalho do desenvolvedor, reduzindo o atrito de interpretação e aumentando a taxa de correção.

6.2.2 Uso de LLMs para mediação didática e interpretação de alertas

Enquanto a normalização proposta na seção anterior foca na padronização estrutural e na apresentação dos dados, existe uma barreira adicional relacionada à compreensão semântica dos alertas. As discussões no grupo focal evidenciaram que a linguagem técnica dos relatórios representa um obstáculo significativo, levando os desenvolvedores a buscar auxílio em fontes externas para compreender a natureza das vulnerabilidades. Considerando a atual transição tecnológica no suporte ao desenvolvimento, caracterizada pela ampla adoção de ferramentas de IA generativa no fluxo de trabalho dos programadores [65], surge uma oportunidade relevante de investigação. Nesse sentido, sugere-se pesquisar o

uso de LLMs não apenas para gerar o código da correção (como explorado no Estudo 2), mas especificamente para atuar na *interpretação e explicação* dos relatórios de segurança.

Trabalhos futuros podem explorar como modelos de linguagem podem processar os relatórios técnicos brutos gerados pelas ferramentas SAST e transformá-los em explicações contextualizadas e didáticas. O objetivo seria utilizar a capacidade semântica dos LLMs para traduzir regras complexas (como as especificações CrySL) para uma linguagem acessível ao desenvolvedor de negócio, explicando o "porquê" da vulnerabilidade e orientando o raciocínio de correção, reduzindo assim a lacuna de conhecimento entre a detecção e a remediação do problema.

6.2.3 Integração de análise estática e dinâmica

Outra oportunidade identificada durante este trabalho é a criação de um mecanismo de validação dinâmica que complemente os relatórios SAST, permitindo a reprodução controlada das vulnerabilidades em ambiente de teste. Essa integração entre análise estática e execução automatizada de casos de teste pode contribuir para reduzir falsos positivos e aumentar a confiabilidade dos alertas, conforme já apontado em diretrizes e estudos que demonstram os benefícios da combinação de múltiplas técnicas de verificação de segurança [66, 67].

6.2.4 Avaliação e adaptação leve de modelos de linguagem para o domínio criptográfico

Os experimentos realizados com modelos de linguagem (SLMs e LLMs) mostraram que, embora esses modelos apresentem potencial para apoiar a correção de vulnerabilidades criptográficas, eles ainda não compreendem plenamente as restrições impostas por APIs de criptografia em ambientes Java. Com base nesse achado, propõe-se como linha de continuidade a investigação de *estratégias de adaptação leve* de modelos de linguagem para o domínio da segurança criptográfica, priorizando abordagens factíveis em contexto acadêmico.

Em vez do *fine-tuning* completo — processo que envolve o re-treinamento interno de um modelo sobre seus pesos originais, atualmente restrito a empresas proprietárias como OpenAI (ChatGPT), Google (Gemini) ou Anthropic (Claude)[1] — recomenda-se explorar alternativas mais acessíveis, tais como:

[1] *Fine-tuning* é o processo de ajuste dos pesos internos de um modelo de linguagem previamente treinado em um novo conjunto de dados, com o objetivo de especializá-lo em determinado domínio. Essa técnica exige acesso direto ao modelo-base e grande capacidade computacional, geralmente restrita às equipes desenvolvedoras.

- Engenharia sistemática de *prompts* (*prompt engineering*), avaliando como diferentes estruturas de instrução, exemplos (*few-shot*, *one-shot*, *chain-of-thought*) e contextos de segurança afetam a qualidade das correções;
- Comparação entre modelos abertos e fechados, medindo a precisão, completude e funcionalidade dos códigos gerados para vulnerabilidades criptográficas específicas;
- Uso de modelos abertos, como Qwen, Mistral, LLaMA ou StarCoder, que permitem adaptações controladas (*adapter layers* ou *LoRA fine-tuning*) sobre conjuntos de dados especializados;
- Construção de conjuntos de dados acadêmicos contendo pares *código inseguro - código corrigido*, possibilitando a reprodutibilidade e a avaliação objetiva dos modelos.

Essas propostas são tecnicamente exequíveis para estudantes e grupos de pesquisa, exigindo apenas infraestrutura computacional moderada e sem depender de acesso proprietário a modelos comerciais. Espera-se que tais investigações contribuam para o desenvolvimento de metodologias reprodutíveis de avaliação de modelos generativos aplicados à segurança de software, além de oferecer subsídios para o aprimoramento de ferramentas automáticas de auditoria e correção de vulnerabilidades [68, 69].

6.2.5 Controle de variância e reprodutibilidade em experimentos com modelos de linguagem

O experimento conduzido neste trabalho adotou o formato *one-shot*, no qual cada execução foi realizada individualmente por método vulnerável. Em cada *prompt*, foram incluídas as instruções direcionadas ao modelo, o código-fonte do método contendo a vulnerabilidade criptográfica, os relatórios do CogniCrypt e do CryptoGuard correspondentes, além da regra CrySL violada. Esse formato garantiu consistência entre os testes e evitou variações decorrentes de múltiplas execuções do mesmo caso.

Embora essa abordagem tenha assegurado resultados determinísticos dentro do escopo deste estudo, a literatura identifica a ausência de determinismo como um desafio recorrente em pesquisas que envolvem modelos de linguagem [63, 62, 64]. Esse comportamento não determinístico pode afetar a reprodutibilidade de experimentos, especialmente quando se realizam múltiplas rodadas de inferência com os mesmos *prompts*. A mitigação desse tipo de variabilidade depende, entre outros fatores, do controle explícito de parâmetros de decodificação, como *temperature*, *top-p* e *seed*, os quais não estão disponíveis nas interfaces web utilizadas neste estudo, mas podem ser ajustados programaticamente nas APIs de desenvolvedor.

Como linha de trabalho futuro, recomenda-se a reprodução dos experimentos utilizando essas APIs, de modo a permitir o controle direto desses parâmetros e avaliar seu impacto sobre a estabilidade e a reprodutibilidade das respostas geradas.

De forma complementar, outra possibilidade de avanço consiste em desenvolver protocolos experimentais que quantifiquem a variância dos resultados obtidos em múltiplas execuções de um mesmo *prompt*, considerando métricas como média, desvio-padrão e intervalo de confiança [62, 63]. Esses protocolos podem incorporar mecanismos de registro sistemático das condições de execução — versão e data do modelo, provedor, ambiente local ou em nuvem — além do número de repetições e da análise de dispersão entre as respostas. O uso de modelos abertos executados localmente (como Qwen [44], Mistral [45] ou LLaMA [70]) também favorece a reprodutibilidade por eliminar atualizações automáticas e permitir a definição explícita de parâmetros de decodificação. A adoção desses cuidados metodológicos fortalecerá a confiabilidade das inferências e contribuirá para a consolidação de práticas reprodutíveis em pesquisas que explorem SLMs e LLMs no domínio da segurança de software.

6.2.6 Automação da triagem de alertas por prioridade de risco

Por fim, propõe-se investigar a integração de mecanismos de *machine learning* (aprendizado de máquina) para classificar automaticamente os alertas emitidos por ferramentas SAST de acordo com sua gravidade e probabilidade de exploração. Essa triagem automatizada pode tornar as auditorias mais eficientes, priorizando correções de maior impacto e reduzindo o esforço manual de análise.

Assim, encerra-se esta seção com a síntese das oportunidades de continuidade do trabalho, as quais poderão orientar futuras investigações sobre segurança criptográfica e o uso de modelos de linguagem aplicados a esse domínio.

A seguir, apresenta-se uma discussão sobre as limitações e ameaças à validade que podem influenciar a interpretação dos resultados obtidos nesta pesquisa.

6.3 Ameaças à Validade da Pesquisa

Esta seção apresenta as limitações e possíveis ameaças à validade que podem afetar a interpretação e a generalização dos resultados obtidos, já discutidas em quatro dimensões (interna, externa, de construto e de conclusão) [71]. Acrescenta-se, contudo, que o caráter institucional e tecnológico específico da Embrapa, aliado à natureza de sistemas legados em Java, restringe a generalização dos resultados para outros contextos organizacionais ou linguagens de programação.

Validade interna. Uma ameaça está relacionada à execução das ferramentas CogniCrypt e CryptoGuard. Apesar do uso de parâmetros e versões recomendadas pela literatura, ajustes específicos do ambiente de análise podem ter introduzido vieses, afetando a detecção ou classificação das vulnerabilidades.

Validade externa. Os sistemas analisados pertencem a um único contexto organizacional — uma importante instituição de pesquisa agropecuária no Brasil — e utilizam versões específicas e, em sua maioria, legadas do ambiente Java. Essa característica limita a generalização dos resultados para outros domínios, linguagens ou organizações com práticas de desenvolvimento distintas e ambientes tecnológicos mais atuais.

Além disso, o suporte restrito oferecido pela versão *Java 7u80* (equivalente a *Java 1.7.0_80*), utilizada nos sistemas analisados, inviabilizou a adoção do modo de operação GCM, amplamente reconhecido como a melhor prática contemporânea por fornecer confidencialidade e integridade simultaneamente. Esse modo de criptografia só passou a ter implementação estável e oficial nos provedores padrão da plataforma a partir das versões *Java 7u191* e *Java 8*, conforme registrado no *bug report* JDK-8180834 e no *Java Enhancement Proposal* (JEP) 115, que documentam a introdução do suporte completo a algoritmos *Authenticated Encryption with Associated Data* (AEAD) no *SunJCE*. Em função dessa limitação técnica, as correções implementadas recorreram ao modo CTR, que, embora não agregue autenticação, foi a opção mais segura compatível com o ambiente legado. Tal decisão pode impactar a generalização dos resultados, visto que, em contextos mais modernos, a substituição direta do ECB pelo GCM seria a escolha preferencial [17, 12, 14, 23, 72, 73].

Validade de construto. A correção das vulnerabilidades foi mensurada pela eliminação dos alertas emitidos pelas ferramentas de análise estática. Entretanto, a ausência de alertas não garante, por si só, que o código esteja totalmente seguro, já que vulnerabilidades não cobertas pelas regras CrySL ou pelos padrões do CryptoGuard podem permanecer presentes.

Validade de conclusão. Há ainda ameaças relacionadas à interpretação dos resultados obtidos no grupo focal e nas avaliações das correções sugeridas por modelos de linguagem (LLMs e SLMs). Apesar do uso de triangulação metodológica, as conclusões dependem tanto da percepção dos participantes quanto de experimentos em um conjunto limitado de sistemas, o que pode reduzir a robustez das generalizações.

Uma limitação adicional refere-se à impossibilidade de controle explícito dos parâmetros de decodificação (*temperature*, *top-p* e *seed*) nas interfaces web utilizadas. Embora essa restrição não tenha comprometido o formato *one-shot* empregado neste estudo, ela pode influenciar a estabilidade dos resultados em experimentos que envolvam múltiplas execuções. Essa limitação é reconhecida e foi considerada nas sugestões de trabalhos fu-

turos, que incluem a replicação dos experimentos por meio das APIs de desenvolvedor, nas quais esses parâmetros podem ser ajustados programaticamente.

6.4 Síntese e Encerramento

Em perspectiva mais ampla, as limitações discutidas não reduzem a contribuição desta pesquisa, mas evidenciam o caráter dinâmico e desafiador da segurança de software no contexto corporativo. Os resultados alcançados reforçam a importância de práticas sistemáticas de verificação e correção de vulnerabilidades, bem como o potencial de integração entre ferramentas de análise estática e modelos de linguagem como apoio ao desenvolvimento seguro. Assim, espera-se que este trabalho possa inspirar novas investigações e aprimoramentos que tornem a aplicação de criptografia em sistemas corporativos cada vez mais confiável, transparente e acessível.

Referências

[1] Egele, Manuel, David Brumley, Yanick Fratantonio e Christopher Kruegel: *An empirical study of cryptographic misuse in android applications*. páginas 73–84, 1, 3–4, 6–7, novembro 2013. 1

[2] Nadi, Sarah, Stefan Krüger, Mira Mezini e Eric Bodden: *Jumping through hoops: why do java developers struggle with cryptography apis?* Em *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, página 935–946, New York, NY, USA, 2016. Association for Computing Machinery, ISBN 9781450339001. https://doi.org/10.1145/2884781.2884790. 1, 7, 9, 11, 12, 13, 41, 67

[3] Ochoa, Lina, Muhammad Hammad, Görkem Giray, Önder Babur e Kwabena Bennin: *Characterising harmful api uses and repair techniques: Insights from a systematic review*. Computer Science Review, 57:100732, 2025, ISSN 1574-0137. https://www.sciencedirect.com/science/article/pii/S1574013725000097. 1, 8, 11, 12, 41

[4] Schneier, Bruce: *Applied Cryptography: Protocols, Algorithms and Source Code in C*. Wiley, 20th anniversary edition edição, 2015, ISBN 978-1119096726. 1, 2, 8, 14, 15, 55, 62, 67

[5] Marcilio, Diego, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz e Gustavo Pinto: *Are static analysis violations really fixed? a closer look at realistic usage of sonarqube*. Em *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, páginas 209–219, 2019. 2

[6] Xiao, Ya, Yang Zhao, Nicholas Allen, Nathan Keynes, Danfeng, Yao e Cristina Cifuentes: *Industrial experience of finding cryptographic vulnerabilities in large-scale codebases*, 2022. https://arxiv.org/abs/2007.06122. 2, 3

[7] Parnas, David Lorge: *Software aging*. Em *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, página 279–287, Washington, DC, USA, 1994. IEEE Computer Society Press, ISBN 081865855X. 3

[8] Balalaie, A, A Heydarnoori e P Jamshidi Dermani: *Microservices architecture enables devops: an experience report on migration to a cloud-native architecture*. 52, 2016, ISSN 0740-7459. 3

[9] Anderson, Ross J.: *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, Inc., Indianapolis, IN, 2ª edição, 2008, ISBN 978-0-470-06852-6. 3, 15

[10] McGraw, Gary: *Software Security: Building Security In*. Addison-Wesley Professional, 2006. 4

[11] Chess, Brian e Jacob West: *Secure Programming with Static Analysis*. Addison-Wesley, Upper Saddle River, NJ, 2007, ISBN 0-321-42477-8. 4

[12] Stallings, William: *Cryptography and Network Security: Principles and Practice*. Pearson Education Limited, Harlow, Essex, England, 7th global edition edição, 2017, ISBN 978-1-292-15858-7. 8, 15, 52, 86

[13] Katz, Jonathan e Yehuda Lindell: *Introduction to Modern Cryptography*. CRC Press, Boca Raton, FL, USA, 2nd edition edição, 2015, ISBN 978-1-4665-7027-6. 8

[14] Paar, Christof e Jan Pelzl: *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer-Verlag Berlin Heidelberg, Berlin, Germany, 2010, ISBN 978-3-642-04100-6. `https://www.springer.com/gp/book/9783642041006`. 8, 14, 52, 54, 55, 65, 86

[15] Bonifacio, Rodrigo, Stefan Krüger, Krishna Narasimhan, Eric Bodden e Mira Mezini: *Dealing with variability in api misuse specification*, 2021. `https://arxiv.org/abs/2105.04950`. 9

[16] Inc. wolfSSL: *wolfcrypt usage reference - wolfssl manual*, 2025. `https://www.wolfssl.com/documentation/manuals/wolfssl/chapter10.html`, Acesso em: 19 mar. 2025. 9

[17] Ferguson, Niels, Bruce Schneier e Tadayoshi Kohno: *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, Inc., Indianapolis, IN, 2010, ISBN 978-0-470-47424-2. 10, 14, 15, 52, 54, 55, 65, 67, 86

[18] Gutmann, Peter: *Cryptographic Security Architecture: Design and Verification*. Springer-Verlag New York, Inc., New York, NY, 2004, ISBN 0-387-95387-6. 10, 11, 67

[19] Europeia, União: *Regulamento geral sobre a proteção de dados (gdpr) - texto legal oficial*, 2016. `https://eur-lex.europa.eu/eli/reg/2016/679/oj?eliuri=eli%3Areg%3A2016%3A679%3Aoj&locale=pt`, Acessado em 14 nov 2024. 10

[20] Council, PCI Security Standards: *Pci dss: Payment card industry data security standard – requirements and security assessment procedures*, 2023. `https://listings.pcisecuritystandards.org/documents/PCI-DSS-v4-0-SAQ-A.pdf`, Acessado em 15 nov 2024. 10

[21] Krüger, Stefan, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler e Ram Kamath: *Cognicrypt: supporting developers in using cryptography*. Em *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '17, página 931–936. IEEE Press, 2017, ISBN 9781538626849. 12, 16, 17, 21, 22, 24

[22] Rahaman, Sazzadur, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu e Danfeng (Daphne) Yao: *Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects*. Em *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, página 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery, ISBN 9781450367479. https://doi.org/10.1145/3319535.3345659. 12, 16, 17, 22, 23, 24, 26, 33

[23] Dworkin, Morris: *Recommendation for block cipher modes of operation*. NIST Special Publication 800-38A, 2001. https://csrc.nist.gov/publications/detail/sp/800-38a/final. 13, 14, 52, 86

[24] Artiles, José, Daniel Chaves e Cecilio Pimentel: *Image encryption using block cipher and chaotic sequences*. Signal Processing: Image Communication, 79, setembro 2019. 14

[25] Howard, Michael e David LeBlanc: *Writing Secure Code*. Microsoft Press, Redmond, Washington, 2nd edição, 2003, ISBN 0-7356-1722-8. 15, 62

[26] Acar, Yasemin, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek e Christian Stransky: *You get where you're looking for: The impact of information sources on code security*. Em *2016 IEEE Symposium on Security and Privacy (SP)*, páginas 289–305, 2016. 15

[27] OWASP: *Top Ten 2021: A Guide to Building Secure Applications*, 2022. https://owasp.org/Top10/, Open Web Application Security Project. Acessado em 24 de outubro de 2024. 16

[28] Standards, National Institute of e Technology (NIST): *Nist official website*, 2024. https://www.nist.gov, Acessado em: 24 out. 2024. 16

[29] Krüger, Stefan, Johannes Späth, Karim Ali, Eric Bodden e Mira Mezini: *Crysl: An extensible approach to validating the correct usage of cryptographic apis*. IEEE Transactions on Software Engineering, 47(11):2382–2400, 2021. 17, 18, 19, 20, 21, 22, 24

[30] Krüger, Stefan, Karim Ali e Eric Bodden: *Cognicryptgen: generating code for the secure usage of crypto apis*. Em *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '20, página 185–198, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450370479. https://doi.org/10.1145/3368826.3377905. 19, 21, 22, 24, 26

[31] CROSSINGTUD: *Crypto api rules - release 3.1.2 (jca)*, 2024. https://github.com/CROSSINGTUD/Crypto-API-Rules/tags, Acesso em: 18 nov. 2024. 21

[32] Braun, Virginia e Victoria Clarke: *Using thematic analysis in psychology*. Qualitative Research in Psychology, 3:77–101, janeiro 2006. 28

[33] Embrapa: *Acesso à informação institucional*. https://www.embrapa.br/sobre-a-embrapa, dezembro 2024. acessado em 22 dez 2024. 28

[34] Embrapa: *Processo de desenvolvimento de produtos de software da embrapa*. https://processos.sede.embrapa.br/ti/pdse, abril 2021. acessado em 01 out 2025. 29

[35] Almeida, Joilton, Luís Amaral e Rodrigo Bonifácio: *Cryptographic api misuses in industry: A case study on prevalence and remediation*. Em *Anais Estendidos do XXV Simpósio Brasileiro de Cibersegurança*, páginas 405–413, Porto Alegre, RS, Brasil, 2025. SBC. https://sol.sbc.org.br/index.php/sbseg_estendido/article/view/36784. 34, 41, 42, 50, 65, 73

[36] Krueger, Richard A. e Mary Anne Casey: *Focus Groups: A Practical Guide for Applied Research*. SAGE Publications, Thousand Oaks, CA, 5th edição, 2015. 34

[37] Kontio, Jyrki, Johanna Bragge e Laura Lehtola: *The focus group method as an empirical tool in software engineering*. Em *Guide to Advanced Empirical Software Engineering*, páginas 93–116. Springer, 2008. 34

[38] Barbour, Rosaline: *Doing Focus Groups*. SAGE Publications, London, 2011. 35

[39] Nagappan, Nachiappan e Thomas Ball: *Use of relative code churn measures to predict system defect density*. Em *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, página 284–292, New York, NY, USA, 2005. Association for Computing Machinery, ISBN 1581139632. https://doi.org/10.1145/1062455.1062514. 37

[40] Hassan, Ahmed E.: *Predicting faults using the complexity of code changes*. Em *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, página 78–88, USA, 2009. IEEE Computer Society, ISBN 9781424434534. https://doi.org/10.1109/ICSE.2009.5070510. 37

[41] MITRE: *CWE – Common Weakness Enumeration*, 2024. https://cwe.mitre.org/, Acesso em: 26 dez. 2024. 41, 42

[42] Focardi, Riccardo, Francesco Palmarini, Stefano Tempesta e Marco Squarcina: *Mind your keys? a security evaluation of java keystores*. Em *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2018. https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_02B-1_Focardi_paper.pdf. 61

[43] Meng, Na, Stefan Nagy, Danfeng Yao, Wenjie Zhuang e Gustavo Arango-Argoty: *Secure coding practices in java: Challenges and vulnerabilities*. Em *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, páginas 372–383, 2018. 61, 62

[44] Alibaba Cloud: *Qwen: Large language models by alibaba cloud*, 2024. https://huggingface.co/Qwen, Acesso em: 31 ago. 2025. 70, 85

[45] AI, Mistral: *Mistral ai models*. https://mistral.ai, 2023. Acesso em: 31 ago. 2025. 70, 85

[46] DeepMind, Google: *Gemma: Lightweight open models by google*. https://deepmind.google/models/gemma/, 2024. Acesso em: 31 ago. 2025. 70

[47] DeepMind, Google: *Gemini models by google deepmind*. https://deepmind.google/technologies/gemini/, 2025. Acesso em: 31 ago. 2025. 70, 71, 73

[48] PBC, Anthropic: *Claude ai models by anthropic*. https://claude.ai, 2025. Acesso em: 31 ago. 2025. 70, 71, 73

[49] OpenAI: *Chatgpt models by openai*. https://openai.com, 2025. Acesso em: 31 ago. 2025. 70, 71, 73

[50] Zhang, Quanjun, Chunrong Fang, Yang Xie, YuXiang Ma, Weisong Sun, Yun Yang e Zhenyu Chen: *A systematic literature review on large language models for automated program repair*, 2024. https://arxiv.org/abs/2405.01466. 70

[51] Rozière, Baptiste, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom e Gabriel Synnaeve: *Code llama: Open foundation models for code*, 2024. https://arxiv.org/abs/2308.12950. 70

[52] Sheng, Y. *et al.*: *Llms in software security: A survey of vulnerability detection techniques and insights*. arXiv preprint arXiv:2502.07049, 2025. https://arxiv.org/pdf/2502.07049v2.pdf. 70

[53] Kaniewski, Sabrina, Fabian Schmidt, Markus Enzweiler, Michael Menth e Tobias Heer: *A systematic literature review on detecting software vulnerabilities with large language models*, 2025. https://arxiv.org/abs/2507.22659. 70

[54] Fitero-Dominguez, David de, Eva Garcia-Lopez, Antonio Garcia-Cabot e Jose Javier Martinez-Herraiz: *Enhanced automated code vulnerability repair using large language models*. Engineering Applications of Artificial Intelligence, 138:109291, 2024, ISSN 0952-1976. https://www.sciencedirect.com/science/article/pii/S0952197624014490. 70

[55] LM Studio: *Lm studio: Run local large language models*, 2025. https://lmstudio.ai, Acesso em: 31 ago. 2025. 70, 71

[56] Ollama: *Ollama for windows: Local llms with graphical interface*. https://ollama.com/download/windows, 2025. Acesso em 31 ago. 2025. 70

[57] Hugging Face Inc.: *Serverless inference api — rate limits and usage restrictions*, 2025. https://huggingface.co/learn/cookbook/en/enterprise_hub_serverless_inference_api, Acesso em: 31 ago. 2025. 71

[58] Google LLC: *Rate limits | gemini api*, 2025. https://ai.google.dev/gemini-api/docs/rate-limits, Acesso em: 31 ago. 2025. 71

[59] Google LLC: *Gemini pro: Official web application for gemini models*, 2025. https://gemini.google.com/app, Acesso em: 31 ago. 2025. 71

[60] Yang, Boyang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F. Bissyandé, Yang Liu e Haoye Tian: *A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications*, 2025. https://arxiv.org/abs/2506.23749. 72

[61] Gheyi, Rohit, Marcio Ribeiro e Jonhnanthan Oliveira: *Evaluating the effectiveness of small language models in detecting refactoring bugs*, 2025. https://arxiv.org/abs/2502.18454. 72

[62] Dodge, Jesse, Suchin Gururangan, Dallas Card, Roy Schwartz e Noah A. Smith: *Show your work: Improved reporting of experimental results.* Em Inui, Kentaro, Jing Jiang, Vincent Ng e Xiaojun Wan (editores): *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, páginas 2185–2194, Hong Kong, China, novembro 2019. Association for Computational Linguistics. https://aclanthology.org/D19-1224/. 72, 73, 84, 85

[63] Liang, Percy, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang e Yuta Koreeda: *Holistic evaluation of language models*, 2023. https://arxiv.org/abs/2211.09110. 72, 73, 84, 85

[64] Ouyang, Tinghui, AprilPyone MaungMaung, Koichi Konishi, Yoshiki Seo e Isao Echizen: *Stability analysis of chatgpt-based sentiment analysis in ai quality assurance.* Electronics, 13(24), 2024, ISSN 2079-9292. https://www.mdpi.com/2079-9292/13/24/5043. 72, 84

[65] Stack Overflow: *2025 developer survey*, 2025. https://survey.stackoverflow.co/2025/, Acesso em: 08 dez. 2025. 82

[66] Scarfone, Karen, Murugiah Souppaya, Amanda Cody e Angela Orebaugh: *Technical guide to information security testing and assessment.* Relatório Técnico Special Publication 800-115, National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA, 2008. https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-115.pdf, Recomenda combinar técnicas de teste; guia clássico de metodologia. 83

[67] Tudela, Francesc, Juan Ramón Higuera, Javier Bermejo, Juan Antonio Montalvo e Michael Argyros: *On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications.* Applied Sciences, 10, dezembro 2020. 83

[68] Berabi, Berkay, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru e Martin Vechev: *Deepcode ai fix: Fixing security vulnerabilities with large language models*, 2024. https://arxiv.org/abs/2402.13291. 84

[69] Zhang, Lan, Qingtian Zou, Anoop Singhal, Xiaoyan Sun e Peng Liu: *Evaluating large language models for real-world vulnerability repair in c/c++ code*. Em *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics*, IWSPA '24, página 49–58, New York, NY, USA, 2024. Association for Computing Machinery, ISBN 9798400705564. https://doi.org/10.1145/3643651.3659892. 84

[70] Touvron, Hugo, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave e Guillaume Lample: *Llama: Open and efficient foundation language models*. arXiv preprint arXiv:2302.13971, 2023. https://arxiv.org/abs/2302.13971, Preprint; Meta AI. 85

[71] Wohlin, Claes, Per Runeson, Martin Höst, {Magnus C} Ohlsson, Björn Regnell e Anders Wesslén: *Experimentation in Software Engineering (2024 edition)*. Springer Berlin, Heidelberg, 2024. Claes Wohlin, Blekinge Institute of Technology Per Runeson, Lund University Martin Höst, Malmö University Magnus C. Ohlsson, System Verification Sweden AB Björn Regnell, Lund University Anders Wesslén, Ericsson AB. 85

[72] OpenJDK: *JEP 115: AEAD CipherSuites*, 2013. https://openjdk.org/jeps/115, Descreve a introdução das suites de cifra AEAD, incluindo o modo AES/GCM, na arquitetura de segurança do Java. Acesso em: 17 out. 2025. 86

[73] Oracle Corporation: *JDK-8180834: Support the GCM cipher suites in JDK 7*, 2017. https://bugs.openjdk.org/browse/JDK-8180834, Registro de bug documentando a adição do suporte às suites de cifra AES/GCM no JDK 7u191. Acesso em: 17 out. 2025. 86

Apêndice A

Volume de Alterações - Correção Manual

Este apêndice reúne as tabelas detalhadas com o volume de alterações realizadas em cada classe durante a correção manual dos componentes analisados.

A.1 Diferenças de Código nas Correções Manuais

Tabela A.1: Volume de Alterações por classe — Correção manual (Componente 1). (Fonte: Própria)

Classe	Incluídas	Excluídas	Total
TagPaginacao	3	2	5
RecursoWS	53	32	85
BaseCripto	146	33	179
RandomAlfaNumerico	2	2	4
ModuloLoginJAASWS	59	44	103
FabricaSocketTLS	52	2	54
GerenciadorConfiancaX509	123	61	184
FabricaSocketTLSV12	71	0	71
Total			**685**

Tabela A.2: Volume de Alterações por classe — Correção manual (Componente 2). (Fonte: Própria)

Classe	Incluídas	Excluídas	Total
UtilHashLM	197	119	316
SenhaDiretorio	8	3	11
UsuarioExternoDiretorio	9	3	12
Total			**339**

Apêndice B

Volume de Alterações - Correção SLM/LLM

Este apêndice reúne as tabelas detalhadas com o volume de alterações realizadas em cada classe durante a correção semiautomatizada com SLM/LLM dos componentes analisados.

B.1 Diferenças de Código nas Correções com SLM e LLM

Tabela B.1: Volume de alterações por classe — SLMs (Componente 1). Cada célula indica o total (Incluídas + Excluídas). (Fonte: Própria)

Classe	Qwen 3 4B	Mistral-7B	Gemma 3n E4B
TagPaginacao	102	6	5
RecursoWS	30	57	39
BaseCripto	75	93	69
RandomAlfaNumerico	8	9	4
ModuloLoginJAASWS	30	57	39
FabricaSocketTLS	32	4	23
GerenciadorConfiancaX509	53	29	63
Total geral (SLMs)	330	255	242

Tabela B.2: Volume de alterações por classe — LLMs (Componente 1). Cada célula indica o total (Incluídas + Excluídas). (Fonte: Própria)

Classe	Gemini 2.5 Pro	Claude Sonnet 4	ChatGPT 5
TagPaginacao	10	5	10
RecursoWS	61	121	91
BaseCripto	101	197	136
RandomAlfaNumerico	5	4	5
ModuloLoginJAASWS	68	125	89
FabricaSocketTLS	154	86	71
GerenciadorConfiancaX509	75	485	160
Total geral (LLMs)	474	1023	562

Tabela B.3: Volume de alterações por classe — SLMs (Componente 2). Cada célula indica o total (Incluídas + Excluídas). (Fonte: Própria)

Classe	Qwen 3 4B	Mistral-7B	Gemma 3n E4B
UtilHashLM	12	33	65
SenhaDiretorio	44	32	19
UsuarioExternoDiretorio	29	42	3
Total geral (SLMs)	85	107	87

Tabela B.4: Volume de alterações por classe — LLMs (Componente 2). Cada célula indica o total (Incluídas + Excluídas). (Fonte: Própria)

Classe	Gemini 2.5 Pro	Claude Sonnet 4	ChatGPT 5
UtilHashLM	83	270	108
SenhaDiretorio	42	45	31
UsuarioExternoDiretorio	31	5	46
Total geral (LLMs)	156	320	185

Apêndice C

Código-fonte original analisado nas ferramentas de análise estática

Nota de confidencialidade e finalidade acadêmica.

Os fragmentos de código apresentados neste apêndice têm finalidade exclusivamente acadêmica, destinando-se à demonstração de vulnerabilidades detectadas por ferramentas de análise estática (CogniCrypt e CryptoGuard). Os exemplos foram extraídos de sistemas internos da Empresa Brasileira de Pesquisa Agropecuária (Embrapa), mas nenhum deles contém credenciais, endereços, chaves criptográficas, dados sensíveis ou informações que permitam inferir detalhes de infraestrutura tecnológica da instituição. As porções aqui reproduzidas foram selecionadas de modo a preservar a fidelidade técnica e o comportamento necessários à reprodutibilidade das análises, mantendo a integridade e a segurança institucional.

Aviso:

Todos os trechos vulneráveis aqui exibidos já foram substituídos em produção; os exemplos visam apenas documentação científica dos resultados obtidos nesta pesquisa.

Este apêndice reúne os trechos de código-fonte originais do Componente 1 e do Componente 2 utilizados nos experimentos apresentados no Capítulo 5, Subseção 5.1.1. Os fragmentos correspondem aos métodos e classes efetivamente analisados pelas ferramentas de análise estática, bem como àqueles citados nas seções que descrevem a ocorrência e a correção de vulnerabilidades no uso de APIs de criptografia. O objetivo é permitir a reprodutibilidade das etapas de análise e correção descritas neste trabalho.

C.1 Código-fonte original do Componente 1

```
1   import java.util.Random;
2   //...
3   public class TagPaginacao implements Tag {
4       //...
5       public int doStartTag() throws JspException {
6           Integer randomId = new Random().nextInt(1000);
7           //...
8           idPaginaAtual = "paginaAtual_" + randomId;
9           idProximaPagina = "proximaPagina_" + randomId;
10          //...
11          try {
12              JspWriter out = pageContext.getOut();
13
14              if (getTotalRegs().intValue() > 0) {
15                  out.print("<p class='link-paginador'>");
16                  //...
17              }
18          } catch (Exception e) {
19              throw new JspException("Error in TagPaginacao.doStartTag()");
20          }
21          return SKIP_BODY;
22      }
23      //...
24  }
```

C.1. Método `doStartTag` original - Classe `TagPaginacao`. (Fonte: Própria)

```
1   // imports
2   public class RecursoWS {
3       //...
4       private void sslCheck() {
5           try {
6               TrustManager[] trustAllCerts = new TrustManager[] { new X509TrustManager() {
7                   public java.security.cert.X509Certificate[] getAcceptedIssuers() {
8                       return null;
9                   }
10                  public void checkClientTrusted(X509Certificate[] certs, String authType) {
11                  }
12                  public void checkServerTrusted(X509Certificate[] certs, String authType) {
13                  }
14                  public void checkClientTrusted(java.security.cert.X509Certificate[] arg0, String
                    ↪ arg1) throws CertificateException {
15                  }
16                  public void checkServerTrusted(java.security.cert.X509Certificate[] arg0, String
                    ↪ arg1) throws CertificateException {
17                  }
18              } };
19
20              SSLContext sc = SSLContext.getInstance("SSL");
21              sc.init(null, trustAllCerts, new java.security.SecureRandom());
22              HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
23
```

```
24                  HostnameVerifier allHostsValid = new HostnameVerifier() {
25                      public boolean verify(String hostname, SSLSession session) {
26                          return true;
27                      }
28                  };
29
30                  HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);
31
32              } catch (NoSuchAlgorithmException | KeyManagementException e) {
33                  //...
34              }
35          }
36      }
```

C.2. Método `sslCheck` original — Classe `RecursoWS`. (Fonte: Própria)

```
1       // imports
2       public class BaseCripto {
3
4           static Log log = LogFactory.getLog(BaseCripto.class);
5           private static byte[] key = { 0x74, 0x68, 0x69, 0x73, 0x49, 0x73, 0x41, 0x53, 0x65, 0x63,
            ↪  0x72, 0x65, 0x74, 0x4b, 0x65, 0x79 };
6
7           public static String encrypt(String strToEncrypt) {
8               try {
9                   Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
10                  final SecretKeySpec secretKey = new SecretKeySpec(key, "AES");
11                  cipher.init(Cipher.ENCRYPT_MODE, secretKey);
12                  final String encryptedString =
                    ↪  Base64.encodeBase64String(cipher.doFinal(strToEncrypt.getBytes()));
13                  return encryptedString;
14              } catch (Exception e) {
15                  log.error("Erro durante a criptografia", e);
16              }
17              return null;
18          }
19
20          public static String decrypt(String strToDecrypt) {
21              try {
22                  Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
23                  final SecretKeySpec secretKey = new SecretKeySpec(key, "AES");
24                  cipher.init(Cipher.DECRYPT_MODE, secretKey);
25                  final String decryptedString = new
                    ↪  String(cipher.doFinal(Base64.decodeBase64(strToDecrypt)));
26                  return decryptedString;
27              } catch (Exception e) {
28                  log.error("Erro durante a descriptografia", e);
29              }
30              return null;
31          }
32      }
```

C.3. Métodos `encrypt` e `decrypt` originais - Classe `BaseCripto`. (Fonte: Própria)

```java
import java.util.Random;

public class RandomAlfaNumerico {
    //...
    public static char randomChar() {
        return toChar(random.nextInt(62));
    }
    // sequencia de caracters randomicos de tamanho entre minLength e maxLength,
    // contendo apenas carcters alfanumericos
    public static String randomString(int minLength, int maxLength) {
        //...
        int size = minLength + random.nextInt(maxLength - minLength + 1);
        return randomString(size);
    }
}
```

C.4. Métodos `randomChar` e `randomString` originais - Classe `RandomAlfaNumerico`. (Fonte: Própria)

```java
// imports
public class ModuloLoginJAASWS implements LoginModule {
    //...
    private void sslCheck() {
        try {
            TrustManager[] trustAllCerts = new TrustManager[] { new X509TrustManager() {
                public java.security.cert.X509Certificate[] getAcceptedIssuers() { return null; }
                public void checkClientTrusted(X509Certificate[] certs, String authType) {}
                public void checkServerTrusted(X509Certificate[] certs, String authType) {}
                public void checkClientTrusted(java.security.cert.X509Certificate[] arg0, String
                ↪ arg1) throws CertificateException {}
                public void checkServerTrusted(java.security.cert.X509Certificate[] arg0, String
                ↪ arg1) throws CertificateException {}
            } };

            SSLContext sc = SSLContext.getInstance("SSL");
            sc.init(null, trustAllCerts, new java.security.SecureRandom());
            HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

            HostnameVerifier allHostsValid = new HostnameVerifier() {
                public boolean verify(String hostname, SSLSession session) {
                    return true;
                }
            };
            HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);

        } catch (NoSuchAlgorithmException | KeyManagementException e) {
            //...
        }
    }
    //...
}
```

C.5. Método `sslCheck` original — Classe `ModuloLoginJAASWS`. (Fonte: Própria)

```
// imports
public class FabricaSocketTLS implements SecureProtocolSocketFactory {
    //...
    private static SSLContext createEasySSLContext() {
        try {
            SSLContext context = SSLContext.getInstance("SSL");
            context.init(null, new TrustManager[] { new GerenciadorConfiancaX509(null) }, null);
            return context;
        } catch (Exception e) {
            LOG.error(e.getMessage(), e);
            throw new HttpClientError(e.toString());
        }
    }

    private SSLContext getSSLContext() {
        if (this.sslcontext == null) {
            this.sslcontext = createEasySSLContext();
        }
        return this.sslcontext;
    }
    //...
}
```

C.6. Método `createEasySSLContext` original - Classe `FabricaSocketTLS`. (Fonte: Própria)

```
// imports
public class GerenciadorConfiancaX509 implements X509TrustManager {
    /** Constructor for GerenciadorConfiancaX509. */
    public GerenciadorConfiancaX509(KeyStore keystore) throws NoSuchAlgorithmException,
    ↪ KeyStoreException {
        super();
        TrustManagerFactory factory =
        ↪ TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
        factory.init(keystore);
        TrustManager[] trustmanagers = factory.getTrustManagers();
        if (trustmanagers.length == 0) {
            throw new NoSuchAlgorithmException("no trust manager found");
        }
        this.standardTrustManager = (X509TrustManager) trustmanagers[0];
    }
    //...
}
```

C.7. Método construtor `GerenciadorConfiancaX509` original - Classe GerenciadorConfiancaX509. (Fonte: Própria)

C.2 Código-fonte original do Componente 2

```
1   // imports
2   public class UtilHashLM {
3       /**
4        * Creates the LM Hash of the user's password.
5        * @param password The password.
6        * @return The LM Hash of the given password, used in the calculation of the LM Response.
7        */
8       private static byte[] lmHash(String password) throws Exception {
9           //...
10          Cipher des = Cipher.getInstance("DES/ECB/NoPadding");
11          des.init(Cipher.ENCRYPT_MODE, lowKey);
12          byte[] lowHash = des.doFinal(magicConstant);
13          des.init(Cipher.ENCRYPT_MODE, highKey);
14          //..
15          return lmHash;
16      }
17      //...
18      private String[] getSmbHashes(String cleartext) {
19          URL url;
20          try {
21              url = new URL("http://server3/perl/test.pl?password=" + cleartext);
22              InputStream is = url.openStream();
23              BufferedReader br = new BufferedReader(new InputStreamReader(is));
24              //...
25              String[] result = {clearText, lanManHash, ntHash};
26              return result;
27          } catch (MalformedURLException | IOException e) {
28             e.printStackTrace();
29          }
30          return null;
31       }
32      //...
33  }
```

C.8. Classe `UtilHashLM` original. (Fonte: Própria)

```
1   import java.util.Random;
2   public class SenhaDiretorio {
3       private static String geraSalt() {
4           //...
5           Random r = new Random();
6           int  fatorRandomico = 0;
7           for(int i=0;i < 2; i++) {
8               fatorRandomico =  r.nextInt(Integer.MAX_VALUE);
9               //...
10          }
11          return salt;
12      }
13      //...
14  }
```

C.9. Método `geraSalt` original - Classe `SenhaDiretorio`. (Fonte: Própria)

```java
import java.util.Random;

public class UsuarioExternoDiretorio {
    //...
    public static String geraSenhaAleatoria () {
        //...
        String cadeiaAleatoria = "";
        Random r = new Random();

        /**
         * Gera e devolve um número inteiro aleatório entre 0 e menor que LIMSUP.
         * @return número gerado
         */
        for (int i = 0;i <8;i++) {
            numero = r.nextInt() % limiteSuperiorCadeia;
            num_aleatorio = (Math.abs(numero));
            cadeiaAleatoria = cadeiaAleatoria + cadeia.substring(num_aleatorio,num_aleatorio+1);
        }
        return cadeiaAleatoria;
    }
    //...
}
```

C.10. Método `geraSenhaAleatoria` original - Classe `UsuarioExternoDiretorio`. (Fonte: Própria)

Apêndice D

Código-fonte corrigido manualmente

Nota de confidencialidade e finalidade acadêmica.

Os fragmentos de código apresentados neste apêndice têm finalidade exclusivamente acadêmica, destinando-se à demonstração das correções manuais aplicadas às vulnerabilidades detectadas por ferramentas de análise estática (CogniCrypt e CryptoGuard). Os exemplos foram extraídos de sistemas internos da Empresa Brasileira de Pesquisa Agropecuária (Embrapa), mas nenhum deles contém credenciais, endereços, chaves criptográficas, dados sensíveis ou informações que permitam inferir detalhes de infraestrutura tecnológica da instituição. As porções aqui reproduzidas foram selecionadas de modo a preservar a fidelidade técnica e o comportamento necessários à reprodutibilidade das análises, mantendo a integridade e a segurança institucional.

Aviso:

Todos os trechos de código apresentados já estão substituídos em produção; estes exemplos visam apenas à documentação científica das correções manuais descritas nesta pesquisa.

Este apêndice reúne os trechos de código-fonte corrigidos manualmente referentes ao Componente 1 e ao Componente 2, conforme os experimentos apresentados no Capítulo 5, Subseção 5.1.1. Os fragmentos correspondem aos métodos e classes efetivamente modificados no processo de correção das vulnerabilidades em APIs criptográficas. O objetivo é permitir a reprodutibilidade das etapas de correção descritas neste trabalho.

D.1 Código-fonte corrigido do Componente 1

```
1   import java.security.SecureRandom;
2   //...
3   public class TagPaginacao implements Tag {
4       //...
5       public int doStartTag() throws JspException {
6           SecureRandom secureRandom = new SecureRandom();
7           Integer randomId = secureRandom.nextInt(1000);
8           //...
9           idPaginaAtual = "paginaAtual_" + randomId;
10          idProximaPagina = "proximaPagina_" + randomId;
11          //...
12          try {
13              JspWriter out = pageContext.getOut();
14
15              if (getTotalRegs().intValue() > 0) {
16                  out.print("<p class='link-paginador'>");
17                  //...
18              }
19          } catch (Exception e) {
20              throw new JspException("Error in TagPaginacao.doStartTag()");
21          }
22          return SKIP_BODY;
23      }
24      //...
25  }
```

D.1. Método `doStartTag` corrigido manualmente - Classe `TagPaginacao`. (Fonte: Própria)

```
1   // imports
2   public class RecursoWS {
3       private static final Log log = LogFactory.getLog(RecursoWS.class);
4       @Autowired
5       private DaoComponente1 daoComponente1;
6
7       private String getKeystorePath() {
8           return daoComponente1.consultarVariaveisPorSiglaNome("keyStore","caminhoKeyStore").getVal
            ↪ or();
9       }
10
11      private char[] getKeystorePassword() {
12          String senhaCriptografada =
            ↪ daoComponente1.consultarVariaveisPorSiglaNome("keyStore","senhaKeyStore").getValor();
13          String senhaDescriptografada = BaseCripto.decrypt(senhaCriptografada);
14          return senhaDescriptografada.toCharArray();
15      }
16
17      private void sslCheck() {
18          try {
19              // Configuração do tipo do KeyStore.
20              KeyStore generatedKeyStore = KeyStore.getInstance("JKS");
```

```
21                  try (InputStream is = new FileInputStream(getKeystorePath())) {
22                      generatedKeyStore.load(is, getKeystorePassword());
23                  }
24
25                  // Configuração do algoritmo do TrustManagerFactory.
26                  TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("SunX509");
27                  trustManagerFactory.init(generatedKeyStore);
28                  TrustManager[] generatedTrustManagers = trustManagerFactory.getTrustManagers();
29
30                  // Configuração do algoritmo do KeyManagerFactory.
31                  KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance("SunX509");
32                  keyManagerFactory.init(generatedKeyStore, getKeystorePassword());
33                  KeyManager[] generatedKeyManagers = keyManagerFactory.getKeyManagers();
34
35                  // Configuração do SSLContext com protocolo compatível.
36                  SSLContext sslContext = SSLContext.getInstance("TLSv1.2");
37                  sslContext.init(generatedKeyManagers, generatedTrustManagers, new SecureRandom());
38
39                  // Configuração da SSLSocketFactory personalizada que força o uso do TLSv1.2.
40                  HttpsURLConnection.setDefaultSSLSocketFactory(new
                    ↪  TLSSocketFactory(sslContext.getSocketFactory()));
41
42                  // Limpa o array de senha após o uso.
43                  Arrays.fill(getKeystorePassword(), '\0');
44
45              } catch (NoSuchAlgorithmException | KeyManagementException | UnrecoverableKeyException |
                ↪  KeyStoreException | IOException | CertificateException e) {
46                  //...
47              }
48          }
49        //...
50      }
```

D.2. Método `sslCheck` corrigido manualmente — Classe `RecursoWS`. (Fonte: Própria)

```
1       // imports
2       public class BaseCripto {
3           static Log log = LogFactory.getLog(BaseCripto.class);
4
5           private static final int IV_SIZE = 16;
6           private static final int SALT_SIZE = 16;
7           private static final int KEY_LENGTH = 128; // Tamanho da chave em bits
8           private static final int ITERATIONS = 65536; // Iterações PBKDF2
9
10          /**
11           * Obtém a senha como um array de caracteres sem utilizar String.
12           *
13           * @return Array de caracteres representando a senha.
14           */
15          private static char[] getPassword() {
16              // Obtém a senha como char[] de forma segura sem usar String.
17              UtilSeguranca utilSeguranca = new UtilSeguranca();
18
19              Map<String, String> parametrosSenhaCriptografia =
                ↪  utilSeguranca.obterParametrosConfiguracao("criptografia");
```

```
20
21              if (parametrosSenhaCriptografia == null) { return null; }
22
23              // Obtém a senha como char[] diretamente.
24              char[] password = parametrosSenhaCriptografia.get("senhaCriptografia").toCharArray();
25
26              if (password == null || password.length == 0) { return null; }
27
28              return password;
29          }
30
31          // Método para gerar um salt seguro aleatório.
32          private static byte[] generateSalt() throws NoSuchAlgorithmException {
33              byte[] salt = new byte[SALT_SIZE];
34              SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
35              secureRandom.nextBytes(salt);
36              return salt;
37          }
38
39          // Método para gerar um IV seguro aleatório.
40          private static byte[] generateIv() throws NoSuchAlgorithmException {
41              byte[] iv = new byte[IV_SIZE];
42              SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
43              secureRandom.nextBytes(iv);
44              return iv;
45          }
46
47          // Método para gerar uma chave segura a partir da senha fornecida e do salt.
48          private static SecretKeySpec deriveKey(char[] password, byte[] salt) throws
            ↪  NoSuchAlgorithmException, InvalidKeySpecException {
49              PBEKeySpec spec = new PBEKeySpec(password, salt, ITERATIONS, KEY_LENGTH);
50              SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
51              byte[] keyBytes = factory.generateSecret(spec).getEncoded();
52              spec.clearPassword();
53              return new SecretKeySpec(keyBytes, "AES");
54          }
55
56          // Método de criptografia (mantendo a assinatura original).
57          public static String encrypt(String strToEncrypt) {
58              try {
59                  // Obtém a senha como char[].
60                  char[] password = getPassword();
61                  if (password == null || password.length == 0) {
62                      throw new IllegalStateException("Senha para criptografia não fornecida.");
63                  }
64
65                  // Gera salt e IV aleatórios.
66                  byte[] salt = generateSalt();
67                  byte[] iv = generateIv();
68
69                  // Deriva a chave usando a senha e o salt.
70                  SecretKeySpec secretKey = deriveKey(password, salt);
71
72                  // Limpa a senha do array após uso.
73                  java.util.Arrays.fill(password, '\0');
74
75                  Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
76                  IvParameterSpec ivSpec = new IvParameterSpec(iv);
```

```
77                  cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);
78
79                  byte[] encrypted = cipher.doFinal(strToEncrypt.getBytes("UTF-8"));
80
81                  // Retorna o salt, o IV e o texto criptografado concatenados em Base64.
82                  return Base64.encodeBase64String(salt) + ":" +
83                         Base64.encodeBase64String(iv) + ":" +
84                         Base64.encodeBase64String(encrypted);
85              } catch (Exception e) {
86                  log.error(e);
87              }
88              return null;
89          }
90
91          // Método de descriptografia (mantendo a assinatura original).
92          public static String decrypt(String strToDecrypt) {
93              try {
94                  // Obtém a senha como char[].
95                  char[] password = getPassword();
96                  if (password == null || password.length == 0) {
97                      throw new IllegalStateException("Senha para criptografia não fornecida.");
98                  }
99
100                 // Separa o salt, o IV e o texto criptografado.
101                 String[] parts = strToDecrypt.split(":");
102                 if (parts.length != 3) {
103                     throw new IllegalArgumentException("O texto a ser descriptografado está em um
                        ↪  formato inválido.");
104                 }
105                 byte[] salt = Base64.decodeBase64(parts[0]);
106                 byte[] iv = Base64.decodeBase64(parts[1]);
107                 byte[] encryptedText = Base64.decodeBase64(parts[2]);
108
109                 // Deriva a chave usando a senha e o salt.
110                 SecretKeySpec secretKey = deriveKey(password, salt);
111
112                 // Limpa a senha do array após uso.
113                 java.util.Arrays.fill(password, '\0');
114
115                 Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
116                 IvParameterSpec ivSpec = new IvParameterSpec(iv);
117                 cipher.init(Cipher.DECRYPT_MODE, secretKey, ivSpec);
118
119                 byte[] decrypted = cipher.doFinal(encryptedText);
120
121                 return new String(decrypted, "UTF-8");
122             } catch (Exception e) {
123                 log.error(e);
124             }
125             return null;
126         }
127     }
```

D.3. Métodos `encrypt` e `decrypt` corrigidos manualmente - Classe `BaseCripto`. (Fonte: Própria)

```java
import java.security.SecureRandom;

public class RandomAlfaNumerico {
    private static final SecureRandom random = new SecureRandom();
    //...
    public static char randomChar() {
        return toChar(random.nextInt(62));
    }
    // sequencia de caracters randomicos de tamanho entre minLength e maxLength,
    // contendo apenas carcters alfanumericos.
    public static String randomString(int minLength, int maxLength) {
        //...
        int size = minLength + random.nextInt(maxLength - minLength + 1);
        return randomString(size);
    }
}
```

D.4. Métodos `randomChar()` e `randomString()` corrigidos manualmente — Classe `RandomAlfaNumerico`. (Fonte: Própria)

```java
// imports
public class ModuloLoginJAASWS implements LoginModule {
    private static final Log log = LogFactory.getLog(ModuloLoginJAASWS.class);
    //...
    private void sslCheck() {
        try {
            // Configuração do tipo do KeyStore.
            KeyStore generatedKeyStore = KeyStore.getInstance("JKS");
            try (InputStream is = new FileInputStream(getKeystorePath())) {
                generatedKeyStore.load(is, getKeystorePassword());
            }

            // Configuração do algoritmo do TrustManagerFactory.
            TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("SunX509");
            trustManagerFactory.init(generatedKeyStore);
            TrustManager[] generatedTrustManagers = trustManagerFactory.getTrustManagers();

            // Configuração do algoritmo do KeyManagerFactory.
            KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance("SunX509");
            keyManagerFactory.init(generatedKeyStore, getKeystorePassword());
            KeyManager[] generatedKeyManagers = keyManagerFactory.getKeyManagers();

            // Configuração do SSLContext com protocolo compatível.
            SSLContext sslContext = SSLContext.getInstance("TLSv1.2");
            sslContext.init(generatedKeyManagers, generatedTrustManagers, new SecureRandom());

            // Configuração da SSLSocketFactory personalizada que força o uso do TLSv1.2.
            HttpsURLConnection.setDefaultSSLSocketFactory(new
            ↪  TLSSocketFactory(sslContext.getSocketFactory()));

            // Limpa o array de senha após o uso.
            Arrays.fill(getKeystorePassword(), '\0');

```

```
33              } catch(NoSuchAlgorithmException | KeyManagementException | UnrecoverableKeyException |
34                      KeyStoreException | IOException | CertificateException e) {
35                  log.error("metodo sslcheck", e);
36                  //...
37              }
38          }
39          //...
40      }
```

D.5. Método `sslCheck` corrigido manualmente — Classe `ModuloLoginJAASWS`. (Fonte: Própria)

```
1       // imports
2       public class FabricaSocketTLS implements SecureProtocolSocketFactory {
3           //...
4           UtilSeguranca utilSeguranca = new UtilSeguranca();
5
6           private static String getKeystorePath() {
7               return utilSeguranca.obterParametrosConfiguracao("keyStore").get("caminhoKeyStore");
8           }
9
10          private static char[] getKeystorePassword() {
11              String senhaCriptografada =
                ↪  utilSeguranca.obterParametrosConfiguracao("keyStore").get("senhaKeyStore");
12              String senhaDescriptografada = BaseCripto.decrypt(senhaCriptografada);
13              return senhaDescriptografada.toCharArray();
14          }
15
16          private static SSLContext createEasySSLContext() {
17              try {
18
19                  // Configuração do tipo do KeyStore.
20                  KeyStore generatedKeyStore = KeyStore.getInstance("JKS");
21                  try (InputStream is = new FileInputStream(getKeystorePath())) {
22                      generatedKeyStore.load(is, getKeystorePassword());
23                  }
24
25                  // Configuração do algoritmo do TrustManagerFactory.
26                  TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("SunX509");
27                  trustManagerFactory.init(generatedKeyStore);
28                  TrustManager[] generatedTrustManagers = trustManagerFactory.getTrustManagers();
29
30                  // Configuração do algoritmo do KeyManagerFactory.
31                  KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance("SunX509");
32                  keyManagerFactory.init(generatedKeyStore, getKeystorePassword());
33                  KeyManager[] generatedKeyManagers = keyManagerFactory.getKeyManagers();
34
35                  // Configuração do SSLContext com protocolo compatível.
36                  SSLContext context = SSLContext.getInstance("TLSv1.2");
37                  context.init(generatedKeyManagers, generatedTrustManagers, new SecureRandom());
38
39                  // Configuração da SSLSocketFactory personalizada que força o uso do TLSv1.2.
40                  HttpsURLConnection.setDefaultSSLSocketFactory(new
                    ↪  FabricaSocketTLSV12(context.getSocketFactory()));
41
```

```
42                  // Limpa o array de senha após o uso.
43                  Arrays.fill(getKeystorePassword(), '\0');
44
45                  return context;
46
47              } catch (Exception e) {
48                  LOG.error(e.getMessage(), e);
49                  throw new HttpClientError(e.toString());
50              }
51          }
52          //...
53      }
```

D.6. Método `createEasySSLContext` corrigido manualmente - Classe `FabricaSocketTLS`. (Fonte: Própria)

```
1       // imports
2       public class FabricaSocketTLSV12 extends SSLSocketFactory {
3
4           private SSLSocketFactory delegate;
5
6           public FabricaSocketTLSV12(SSLSocketFactory delegate) {
7               this.delegate = delegate;
8           }
9
10          @Override
11          public String[] getDefaultCipherSuites() {
12              return delegate.getDefaultCipherSuites();
13          }
14
15          @Override
16          public String[] getSupportedCipherSuites() {
17              return delegate.getSupportedCipherSuites();
18          }
19
20          @Override
21          public Socket createSocket(Socket s, String host, int port, boolean autoClose) throws
            ↪  IOException {
22              Socket socket = delegate.createSocket(s, host, port, autoClose);
23              enableTLSv12(socket);
24              return socket;
25          }
26
27          @Override
28          public Socket createSocket(String host, int port) throws IOException {
29              Socket socket = delegate.createSocket(host, port);
30              enableTLSv12(socket);
31              return socket;
32          }
33
34          @Override
35          public Socket createSocket(String host, int port, InetAddress localHost, int localPort)
            ↪  throws IOException {
36              Socket socket = delegate.createSocket(host, port, localHost, localPort);
37              enableTLSv12(socket);
```

```
38              return socket;
39          }
40
41          @Override
42          public Socket createSocket(InetAddress host, int port) throws IOException {
43              Socket socket = delegate.createSocket(host, port);
44              enableTLSv12(socket);
45              return socket;
46          }
47
48          @Override
49          public Socket createSocket(InetAddress address, int port, InetAddress localAddress, int
            ↪ localPort)
50                  throws IOException {
51              Socket socket = delegate.createSocket(address, port, localAddress, localPort);
52              enableTLSv12(socket);
53              return socket;
54          }
55
56          private void enableTLSv12(Socket socket) {
57              if (socket instanceof SSLSocket) {
58                  SSLSocket sslSocket = (SSLSocket) socket;
59                  sslSocket.setEnabledProtocols(new String[] { "TLSv1.2" });
60              }
61          }
62
63      }
```

D.7. Classe `FabricaSocketTLSV12` - Nova classe no código-fonte corrigido manualmente. (Fonte: Própria)

```
1       // imports
2       public class GerenciadorConfiancaX509 implements X509TrustManager {
3           UtilSeguranca utilSeguranca = new UtilSeguranca();
4
5           /**
6            * Construtor para GerenciadorConfiancaX509.
7            *
8            * @param keystore O KeyStore contendo os certificados confiáveis.
9            * @throws NoSuchAlgorithmException Se o algoritmo especificado não estiver disponível.
10           * @throws KeyStoreException Se houver um erro no KeyStore.
11           */
12          public GerenciadorConfiancaX509() throws NoSuchAlgorithmException, KeyStoreException {
13              super();
14
15              try {
16                  // Configuração do tipo do KeyStore.
17                  KeyStore generatedKeyStore = KeyStore.getInstance("JKS");
18                  try (InputStream is = new FileInputStream(getKeystorePath())) {
19                      generatedKeyStore.load(is, getKeystorePassword());
20                  }
21
22                  if (generatedKeyStore.size() == 0) {
23                      throw new KeyStoreException("KeyStore está vazio ou não foi carregado
                        ↪ corretamente");
```

```
24                  }
25
26                  // Configuração do algoritmo "PKIX".
27                  TrustManagerFactory factory = TrustManagerFactory.getInstance("PKIX");
28                  factory.init(generatedKeyStore);
29
30                  TrustManager[] trustManagers = factory.getTrustManagers();
31                  if (trustManagers.length == 0) {
32                      throw new NoSuchAlgorithmException("Nenhum TrustManager encontrado");
33                  }
34
35                  if (!(trustManagers[0] instanceof X509TrustManager)) {
36                      throw new NoSuchAlgorithmException("O TrustManager encontrado não é uma instância
                        ↪  de X509TrustManager");
37                  }
38
39                  this.standardTrustManager = (X509TrustManager) trustManagers[0];
40
41              } catch (NoSuchAlgorithmException | KeyStoreException | IOException |
                ↪  CertificateException e) {
42                  //...
43              }
44
45          }
46
47          private String getKeystorePath() {
48              return utilSeguranca.obterParametrosConfiguracao("keyStore").get("caminhoKeyStore");
49          }
50
51          private char[] getKeystorePassword() {
52              String senhaCriptografada =
                ↪  utilSeguranca.obterParametrosConfiguracao("keyStore").get("senhaKeyStore");
53              String senhaDescriptografada = BaseCripto.decrypt(senhaCriptografada);
54              return senhaDescriptografada.toCharArray();
55          }
56
57          /**
58           * Verifica a confiança do cliente.
59           *
60           * @param certificates Cadeia de certificados do cliente.
61           * @param authType Tipo de autenticação.
62           * @throws CertificateException Se os certificados não forem confiáveis.
63           */
64          @Override
65          public void checkClientTrusted(X509Certificate[] certificates, String authType) throws
            ↪  CertificateException {
66              standardTrustManager.checkClientTrusted(certificates, authType);
67          }
68
69          /**
70           * Verifica a confiança do servidor.
71           *
72           * @param certificates Cadeia de certificados do servidor.
73           * @param authType Tipo de autenticação.
74           * @throws CertificateException Se os certificados não forem confiáveis.
75           */
76          @Override
```

```
77          public void checkServerTrusted(X509Certificate[] certificates, String authType) throws
            ↪  CertificateException {
78              if ((certificates != null) && LOG.isDebugEnabled()) {
79                  LOG.debug("Server certificate chain:");
80                  for (int i = 0; i < certificates.length; i++) {
81                      LOG.debug("X509Certificate[" + i + "]=" + certificates[i]);
82                  }
83              }
84              if ((certificates != null) && (certificates.length == 1)) {
85                  certificates[0].checkValidity();
86              } else {
87                  standardTrustManager.checkServerTrusted(certificates, authType);
88              }
89          }
90
91          /**
92           * Retorna os emissores de certificados aceitos.
93           *
94           * @return Array de certificados aceitos.
95           */
96          @Override
97          public X509Certificate[] getAcceptedIssuers() {
98              return this.standardTrustManager.getAcceptedIssuers();
99          }
100         //...
101     }
```

D.8. Método construtor `GerenciadorConfiancaX509` corrigido manualmente - Classe `GerenciadorConfiancaX509`. (Fonte: Própria)

D.2 Código-fonte corrigido do Componente 2

```
1       // imports
2       public class UtilHashLM {
3
4           private static final int SALT_SIZE = 16; // Tamanho do salt em bytes.
5           private static final int IV_SIZE = 16; // Tamanho do IV em bytes.
6           private static final int KEY_SIZE = 256; // Tamanho da chave AES em bits.
7           private static final int ITERATIONS = 65536; // Número de iterações para PBKDF2.
8
9           /**
10           * Gera um hash seguro utilizando AES/CTR/NoPadding para a senha fornecida.
11           *
12           * @param password A senha a ser hashada.
13           * @return O hash seguro em formato hexadecimal.
14           * @throws Exception Se ocorrer um erro durante a geração do hash.
15           */
16          public static String lmHashString(String password) throws Exception {
17              // Gera salt aleatório.
18              byte[] salt = generateRandomBytes(SALT_SIZE);
19
20              // Deriva chave AES a partir da senha e do salt.
21              SecretKeySpec secretKey = deriveAESKey(password.toCharArray(), salt);
```

```
22
23              // Gera IV aleatório.
24              byte[] iv = generateRandomBytes(IV_SIZE);
25              IvParameterSpec ivSpec = new IvParameterSpec(iv);
26
27              // Inicializa Cipher com AES/CTR/PKCS5Padding.
28              Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
29              cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);
30
31              // Criptografa a senha.
32              byte[] encrypted = cipher.doFinal(password.getBytes("UTF-8"));
33
34              // Concatena salt, IV e texto criptografado.
35              byte[] hashBytes = new byte[salt.length + iv.length + encrypted.length];
36              System.arraycopy(salt, 0, hashBytes, 0, salt.length);
37              System.arraycopy(iv, 0, hashBytes, salt.length, iv.length);
38              System.arraycopy(encrypted, 0, hashBytes, salt.length + iv.length, encrypted.length);
39
40              // Converte para string hexadecimal.
41              String hashHex = toHexString(hashBytes, false);
42
43              // Limpa dados sensíveis da memória.
44              Arrays.fill(salt, (byte) 0);
45              Arrays.fill(iv, (byte) 0);
46              Arrays.fill(encrypted, (byte) 0);
47              Arrays.fill(hashBytes, (byte) 0);
48
49              return hashHex;
50          }
51
52          /**
53           * Gera o hash NT utilizando SHA-256 para a senha fornecida.
54           *
55           * @param senha A senha a ser hashada.
56           * @return O hash NT em formato hexadecimal (32 caracteres).
57           * @throws Exception Se ocorrer um erro durante a geração do hash.
58           */
59          public static String ntHashString(String senha) throws Exception {
60              // Implementação segura para geração do hash NT utilizando SHA-256.
61              java.security.MessageDigest md = java.security.MessageDigest.getInstance("SHA-256");
62              byte[] ntHashBytes = md.digest(senha.getBytes("UTF-8"));
63              String ntHash = toHexString(ntHashBytes, false).substring(0, 32);
64              // Limpeza dos bytes de hash na memória.
65              Arrays.fill(ntHashBytes, (byte) 0);
66              return ntHash;
67          }
68
69          /**
70           * Deriva uma chave AES a partir de uma senha e um salt utilizando PBKDF2.
71           *
72           * @param password A senha como array de caracteres.
73           * @param salt     O salt como array de bytes.
74           * @return A chave AES derivada.
75           * @throws NoSuchAlgorithmException Se o algoritmo PBKDF2WithHmacSHA256 não estiver disponível.
76           * @throws InvalidKeySpecException  Se o PBEKeySpec for inválido.
77           */
78          private static SecretKeySpec deriveAESKey(char[] password, byte[] salt) throws
            ↪ NoSuchAlgorithmException, InvalidKeySpecException {
```

```
79              PBEKeySpec spec = new PBEKeySpec(password, salt, ITERATIONS, KEY_SIZE);
80              SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
81              byte[] keyBytes = factory.generateSecret(spec).getEncoded();
82              SecretKeySpec secretKey = new SecretKeySpec(keyBytes, "AES");
83              // Limpa dados sensíveis.
84              spec.clearPassword();
85              Arrays.fill(keyBytes, (byte) 0);
86              return secretKey;
87          }
88
89          /**
90           * Gera um array de bytes aleatórios com o tamanho especificado.
91           *
92           * @param size O tamanho do array de bytes.
93           * @return Um array de bytes aleatórios.
94           * @throws NoSuchAlgorithmException Se o algoritmo SecureRandom não estiver disponível.
95           */
96          private static byte[] generateRandomBytes(int size) throws NoSuchAlgorithmException {
97              byte[] bytes = new byte[size];
98              java.security.SecureRandom sr = new java.security.SecureRandom();
99              sr.nextBytes(bytes);
100             return bytes;
101         }
102
103         /**
104          * Converte um array de bytes para uma string hexadecimal.
105          *
106          * @param bytes   O array de bytes a ser convertido.
107          * @param dashing Se true, adiciona hífens entre os bytes.
108          * @return A representação hexadecimal do array de bytes.
109          */
110         private static String toHexString(byte[] bytes, boolean dashing) {
111             StringBuffer hexStr = new StringBuffer();
112             for (int i = 0; i < bytes.length; i++) {
113                 int hex = (0xFF & bytes[i]);
114                 String tmp = Integer.toHexString(hex).toUpperCase();
115                 if (tmp.length() == 1) {
116                     hexStr.append('0');
117                 }
118                 hexStr.append(tmp);
119                 if (dashing && (i + 1) != bytes.length) {
120                     hexStr.append('-');
121                 }
122             }
123             return hexStr.toString();
124         }
125
126         /**
127          * Recupera os hashes SMB a partir de um serviço externo de forma segura.
128          *
129          * @param cleartext A senha em texto claro.
130          * @return Um array contendo o texto claro, o hash LAN Manager e o hash NT.
131          */
132         public static String[] getSmbHashes(String cleartext) {
133             URL url = null;
134             HttpURLConnection connection = null;
135             InputStream is = null;
136             BufferedReader br = null;
```

```
137             try {
138                 // Uso de HTTPS e método POST para transmitir a senha de forma segura.
139                 url = new URL("https://server3/perl/test.pl");
140                 connection = (HttpURLConnection) url.openConnection();
141                 connection.setRequestMethod("POST");
142                 connection.setDoOutput(true);
143                 connection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
144 
145                 // Formata os dados a serem enviados no corpo da requisição.
146                 String urlParameters = "password=" + java.net.URLEncoder.encode(cleartext, "UTF-8");
147 
148                 // Envia os parâmetros.
149                 DataOutputStream wr = new DataOutputStream(connection.getOutputStream());
150                 wr.writeBytes(urlParameters);
151                 wr.flush();
152                 wr.close();
153 
154                 // Verifica o código de resposta.
155                 int responseCode = connection.getResponseCode();
156                 if (responseCode != HttpURLConnection.HTTP_OK) {
157                     throw new IOException("Requisição falhou com o código de resposta: " +
                        ↪  responseCode);
158                 }
159 
160                 // Lê a resposta.
161                 is = connection.getInputStream();
162                 br = new BufferedReader(new InputStreamReader(is));
163                 String clearText = br.readLine();
164                 String lanManHash = br.readLine();
165                 String ntHash = br.readLine();
166                 return new String[] { clearText, lanManHash, ntHash };
167             } catch (MalformedURLException | IOException e) {
168                 e.printStackTrace();
169             } finally {
170                 //...
171             }
172             return null;
173         }
174         //...
175     }
```

D.9. Classe `UtilHashLM` corrigido manualmente. (Fonte: Própria)

```
1     import java.security.SecureRandom;
2 
3     public class SenhaDiretorio {
4 
5         private static String geraSalt() {
6             //...
7             String salt = "";
8 
9             /** Uso de SecureRandom. Mais seguro. */
10            SecureRandom r = new SecureRandom();
11 
12            int  fatorRandomico = 0;
```

```
13              for(int i=0;i < 2; i++) {
14                  fatorRandomico =  r.nextInt(Integer.MAX_VALUE);
15                  //...
16                  salt = salt + cadeiaAuxiliar[j];
17              }
18              return salt;
19          }
20          //...
21      }
```

D.10. Método `geraSalt` corrigido manualmente - Classe `SenhaDiretorio`. (Fonte: Própria)

```
1       import java.security.SecureRandom;
2
3       public class UsuarioExternoDiretorio {
4           //...
5           public static String geraSenhaAleatoria () {
6
7               /** gerador de sequencia aleatórios */
8               String cadeia = "abcdefghijkmnopqrstuvwxyz23456789";
9               int limiteSuperiorCadeia = cadeia.length();
10              int numero = 0;
11              int num_aleatorio = 0;
12              String cadeiaAleatoria = "";
13
14              /** Uso de SecureRandom. Mais seguro. */
15              SecureRandom r = new SecureRandom();
16
17              /**
18               * Gera e devolve um número inteiro aleatório entre 0 e menor que LIMSUP.
19               * @return número gerado
20               */
21              for (int i = 0;i <8;i++) {
22                  numero = r.nextInt() % limiteSuperiorCadeia;
23                 //...
24                  cadeiaAleatoria = cadeiaAleatoria + cadeia.substring(num_aleatorio,num_aleatorio+1);
25              }
26              return cadeiaAleatoria;
27          }
28          //...
29      }
```

D.11. Método `geraSenhaAleatoria` corrigido manualmente - Classe `UsuarioExternoDiretorio`. (Fonte: Própria)

Apêndice E

Código-fonte sugerido por SLM

Nota de confidencialidade e finalidade acadêmica.

Os fragmentos de código apresentados neste apêndice têm finalidade exclusivamente acadêmica, destinando-se à demonstração das **sugestões de correção** geradas por modelos de linguagem de pequeno porte (SLM) a partir das vulnerabilidades identificadas por ferramentas de análise estática (CogniCrypt e CryptoGuard). As saídas apresentadas correspondem ao código **sugerido automaticamente pelos modelos**, podendo incluir trechos que não corrigem integralmente as vulnerabilidades ou que introduzem novos erros, inclusive de compilação.

Os exemplos foram extraídos de sistemas internos da Empresa Brasileira de Pesquisa Agropecuária (Embrapa), mas nenhum deles contém credenciais, endereços, chaves criptográficas, dados sensíveis ou informações que permitam inferir detalhes de infraestrutura tecnológica da instituição. As porções aqui reproduzidas foram selecionadas de modo a preservar a fidelidade técnica e o comportamento necessários à reprodutibilidade das análises, mantendo a integridade e a segurança institucional.

Aviso:

Todos os trechos vulneráveis originais já foram substituídos em produção; os exemplos aqui exibidos representam apenas as **sugestões automáticas de correção** obtidas nos experimentos com SLMs, não correspondendo necessariamente a versões seguras, funcionais ou compiláveis do código.

Este apêndice reúne os trechos de código-fonte que representam **sugestões geradas automaticamente por modelos de linguagem de pequeno porte** (SLM), referentes ao Componente 1 e ao Componente 2, conforme descrito no Capítulo 5, Seção 5.2. Os fragmentos correspondem às versões produzidas pelos modelos após a aplicação dos *prompts* experimentais, podendo incluir tentativas de correção parciais ou incorretas.

O objetivo é permitir a reprodutibilidade das etapas e resultados das análises realizadas, preservando a fidelidade técnica dos experimentos sem comprometer a segurança ou o sigilo institucional.

E.1 Sugestão gerada pela SLM: Qwen 3 4B

Componente 1

```
import java.security.SecureRandom;
//...
public class TagPaginacao implements Tag {
    //...
    private static final SecureRandom secureRandom = new SecureRandom();

    public int doStartTag() throws JspException {
        Integer randomId = secureRandom.nextInt(1000);
        //...
        idPaginaAtual = "paginaAtual_" + randomId;
        //...
        try {
            JspWriter out = pageContext.getOut();

            if (getTotalRegs().intValue() > 0) {
                out.print("<p class='link-paginador'>");
                //...
            }
        } catch (Exception e) {
            throw new JspException("Error in TagPaginacao.doStartTag()");
        }
        return SKIP_BODY;
    }
    //...
}
```

E.1. Método `doStartTag`. Sugestão gerada automaticamente por SLM: Qwen 3 4B - Classe `TagPaginacao`. (Fonte: Própria)

```
// imports
public class RecursoWS {

    private void sslCheck() {
        try {
            TrustManager[] trustAllCerts = new TrustManager[] { new X509TrustManager() {
                public java.security.cert.X509Certificate[] getAcceptedIssuers() {
                    return new X509Certificate[0]; // Retorna um array vazio (não é a solução
                    ↪ ideal)
                }

                public void checkClientTrusted(X509Certificate[] certs, String authType) throws
                ↪ CertificateException {
```

```
12                      if (certs == null || certs.length == 0) {
13                          throw new CertificateException("No certificate provided for client
                            ↪ trust");
14                      }
15                      // Verificação mais robusta aqui
16                  }
17
18                  public void checkServerTrusted(X509Certificate[] certs, String authType) throws
                    ↪ CertificateException {
19                      if (certs == null || certs.length == 0) {
20                          throw new CertificateException("No certificate provided for server
                            ↪ trust");
21                      }
22                      // Verificação mais robusta aqui
23                  }
24              } };
25
26              SSLContext sc = SSLContext.getInstance("TLSv1.2"); // Protocolo corrigido (TLSv1.2 ou
                ↪ TLSv1.3)
27              sc.init(null, trustAllCerts, new java.security.SecureRandom());
28              HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
29
30              HostnameVerifier hostnameVerifier = new HostnameVerifier() {
31                  public boolean verify(String hostname, SSLSession session) {
32                      return (session.getPeerHost() != null &&
                        ↪ session.getPeerHost().equals(hostname));
33                  }
34              };
35
36              HttpsURLConnection.setDefaultHostnameVerifier(hostnameVerifier);
37
38          } catch (NoSuchAlgorithmException | KeyManagementException e) {
39              //...
40          }
41      }
42    //...
43  }
```

E.2. Método `sslCheck`. Sugestão gerada automaticamente por SLM: Qwen 3 4B — Classe `RecursoWS`. (Fonte: Própria)

```
1   // imports
2   public class BaseCripto {
3       static Log log = LogFactory.getLog(BaseCripto.class);
4
5       public static String encrypt(String strToEncrypt) {
6           try {
7               // Gerar chave dinamicamente (não constante)
8               KeyGenerator keyGen = KeyGenerator.getInstance("AES");
9               SecretKey secretKey = keyGen.generateKey();
10              SecretKeySpec secretKeySpec = new SecretKeySpec(secretKey.getEncoded(), "AES");
11
12              // Configurar Cipher com modo seguro (GCM)
13              Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
14              cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec);
```

```
15
16                  // Criptografar e codificar
17                  final String encryptedString =
                    ↪  Base64.encodeBase64String(cipher.doFinal(strToEncrypt.getBytes()));
18                  return encryptedString;
19              } catch (Exception e) {
20                  log.error("Erro durante a criptografia", e);
21                  return null;
22              }
23          }
24
25          public static String decrypt(String strToDecrypt) {
26              try {
27                  // Gerar chave dinamicamente (não constante)
28                  KeyGenerator keyGen = KeyGenerator.getInstance("AES");
29                  SecretKey secretKey = keyGen.generateKey();
30                  SecretKeySpec secretKeySpec = new SecretKeySpec(secretKey.getEncoded(), "AES");
31
32                  // Configurar Cipher com modo seguro (GCM)
33                  Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
34                  cipher.init(Cipher.DECRYPT_MODE, secretKeySpec);
35
36                  // Decodificar Base64 e descriptografar
37                  byte[] encryptedBytes = Base64.decodeBase64(strToDecrypt);
38                  final String decryptedString = new String(cipher.doFinal(encryptedBytes));
39                  return decryptedString;
40              } catch (Exception e) {
41                  log.error("Erro durante a descriptografia", e);
42                  return null;
43              }
44          }
45      }
```

E.3. Métodos `encrypt` e `decrypt`. Sugestão gerada automaticamente por SLM: Qwen 3 4B - Classe `BaseCripto`. (Fonte: Própria)

```
1       import java.security.SecureRandom;
2       public class RandomAlfaNumerico {
3           private static final SecureRandom secureRandom = new SecureRandom();
4           //...
5           public static char randomChar() {
6               return toChar(secureRandom.nextInt(62));
7           }
8           // sequencia de caracters randomicos de tamanho entre minLength e maxLength,
9           // contendo apenas carcters alfanumericos
10          public static String randomString(int minLength, int maxLength) {
11              //...
12              int size = minLength + secureRandom.nextInt(maxLength - minLength + 1);
13              return randomString(size);
14          }
15      }
```

E.4. Métodos `randomChar()` e `randomString()`. Sugestão gerada automaticamente por SLM: Qwen 3 4B - Classe `RandomAlfaNumerico`. (Fonte: Própria)

```
    // imports
    public class ModuloLoginJAASWS implements LoginModule {
        private static final Log log = LogFactory.getLog(ModuloLoginJAASWS.class);
        //...
        private void sslCheck() {
            try {
                // Tipo do KeyStore.
                KeyStore generatedKeyStore = KeyStore.getInstance("JKS");
                try (InputStream is = new FileInputStream(getKeystorePath())) {
                    generatedKeyStore.load(is, getKeystorePassword());
                }

                // Informar Algoritmo do TrustManagerFactory.
                TrustManagerFactory trustManagerFactory = TrustManagerFactory.getInstance("SunX509");
                trustManagerFactory.init(generatedKeyStore);
                TrustManager[] generatedTrustManagers = trustManagerFactory.getTrustManagers();

                // Informar Algoritmo do KeyManagerFactory.
                KeyManagerFactory keyManagerFactory = KeyManagerFactory.getInstance("SunX509");
                keyManagerFactory.init(generatedKeyStore, getKeystorePassword());
                KeyManager[] generatedKeyManagers = keyManagerFactory.getKeyManagers();

                // Definindo SSLContext.
                SSLContext sslContext = SSLContext.getInstance("TLSv1.2");
                sslContext.init(generatedKeyManagers, generatedTrustManagers, new SecureRandom());

                // SSLSocketFactory que força o uso do TLSv1.2.
                HttpsURLConnection.setDefaultSSLSocketFactory(new
                ↪  TLSSocketFactory(sslContext.getSocketFactory()));

                // Limpando a senha após o uso.
                Arrays.fill(getKeystorePassword(), '\0');

            } catch(NoSuchAlgorithmException | KeyManagementException | UnrecoverableKeyException |
            ↪  KeyStoreException | IOException | CertificateException e) {
                //...
            }
        }
        //...
    }
```

E.5. Método `sslCheck()`. Sugestão gerada automaticamente por SLM: Qwen 3 4B - Classe `ModuloLoginJAASWS`. (Fonte: Própria)

```
    // imports
    public class FabricaSocketTLS implements SecureProtocolSocketFactory {
        //...

        private static SSLContext createEasySSLContext() {
            try {
                // FIX 1: Alterar protocolo para TLSv1.2 (violação do ConstraintError)
                SSLContext context = SSLContext.getInstance("TLSv1.2");

```

```
10                  // FIX 2: Criar KeyManager[] vazio (correção do "First parameter")
11                  KeyManager[] keyManagers = new KeyManager[0];
12
13                  // FIX 3: Criar keystore vazio para GerenciadorConfiancaX509 (correção do "Second
                    ↪  parameter")
14                  KeyStore keystore = KeyStore.getInstance("PKCS12");
15                  keystore.load(null, null);
16
17                  // FIX 4: Criar SecureRandom randômico (correção do "Third parameter")
18                  SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");
19
20                  // Inicializar SSLContext com os parâmetros corrigidos
21                  context.init(keyManagers, new TrustManager[] { new FabricaSocketTLSV12(keystore) },
                    ↪  secureRandom);
22                  return context;
23              } catch (Exception e) {
24                  LOG.error(e.getMessage(), e);
25                  throw new HttpClientError(e.toString());
26              }
27          }
28          //...
29      }
```

E.6. Método `createEasySSLContext`. Sugestão gerada automaticamente por SLM: Qwen 3 4B - Classe `FabricaSocketTLS`. (Fonte: Própria)

```
1       // imports
2       public class GerenciadorConfiancaX509 implements X509TrustManager {
3
4           public GerenciadorConfiancaX509(KeyStore keystore) throws NoSuchAlgorithmException,
            ↪  KeyStoreException {
5               super();
6
7               // FIX: Forçamos algoritmo explícito "PKIX" para resolver CRYSL
8               TrustManagerFactory factory = TrustManagerFactory.getInstance("PKIX");
9               factory.init(keystore);
10
11              TrustManager[] trustmanagers = factory.getTrustManagers();
12              if (trustmanagers.length == 0) {
13                  throw new NoSuchAlgorithmException("no trust manager found");
14              }
15              this.standardTrustManager = (X509TrustManager) trustmanagers[0];
16          }
17          //...
18      }
```

E.7. Método construtor `GerenciadorConfiancaX509`. Sugestão gerada automaticamente por SLM: Qwen 3 4B - Classe `GerenciadorConfiancaX509`. (Fonte: Própria)

Componente 2

```
// imports
public class UtilHashLM {

    private static byte[] lmHash(String password) throws Exception {
        //...
        Cipher des = Cipher.getInstance("PBEWithHmacSHA256AndAES_128");
        des.init(Cipher.ENCRYPT_MODE, lowKey);
        //...
        return lmHash;
    }

    private static Key createDESKey(byte[] bytes, int offset) {
        //...
        return new SecretKeySpec(material, "AES");
    }

    private String[] getSmbHashes(String cleartext) {
        URL url;
        try {
           url = new URL("https://server3/perl/test.pl?password=" + cleartext); // Alterado de
           ↪ HTTP para HTTPS
           //...
           String[] result = {clearText, lanManHash, ntHash};
           return result;
        } catch (MalformedURLException e) {
           //...
        }
        return null;
    }
    //...
}
```

E.8. Classe `UtilHashLM`. Sugestão gerada automaticamente por SLM: Qwen 3 4B. (Fonte: Própria)

```
import java.security.SecureRandom;

public class SenhaDiretorio {
    private static String geraSalt() {
        // ...
        String salt = "";

        // Substitui Random por SecureRandom para gerar números aleatórios seguros
        SecureRandom r = new SecureRandom();
        int  fatorRandomico = 0;
        for(int i=0;i < 2; i++) {
            fatorRandomico =  r.nextInt(Integer.MAX_VALUE); // tamanho máximo 2147483647
            //...
        }
        return salt;
    }
```

```
17          //...
18      }
```

E.9. Método `geraSalt`. Sugestão gerada automaticamente por SLM: Qwen 3 4B - Classe `SenhaDiretorio`. (Fonte: Própria)

```
1       import java.security.SecureRandom;
2
3       public class UsuarioExternoDiretorio {
4           //...
5           public static String geraSenhaAleatoria () {
6               //...
7               String cadeiaAleatoria = "";
8
9               // Substituição de Random por SecureRandom para gerar números aleatórios seguros
10              SecureRandom r = new SecureRandom();
11              //gera cadeia aleatória
12              for (int i = 0;i <8;i++) {
13                  numero = r.nextInt() % limiteSuperiorCadeia;
14                  //...
15                  cadeiaAleatoria = cadeiaAleatoria + cadeia.substring(num_aleatorio,num_aleatorio+1);
16              }
17              return cadeiaAleatoria;
18          }
19          //...
20      }
```

E.10. Método `geraSenhaAleatoria`. Sugestão gerada automaticamente por SLM: Qwen 3 4B - Classe `UsuarioExternoDiretorio`. (Fonte: Própria)

E.2 Sugestão gerada pela SLM: Mistral-7B

Componente 1

```
1       import java.security.SecureRandom;
2       //...
3       public class TagPaginacao implements Tag {
4           //...
5           private static final SecureRandom secureRandom = new SecureRandom();
6
7           public int doStartTag() throws JspException {
8               Integer randomId = secureRandom.nextInt(1000);
9               //...
10              idPaginaAtual = "paginaAtual_" + randomId;
11              //...
12              try {
13                  JspWriter out = pageContext.getOut();
14
15                  if (getTotalRegs().intValue() > 0) {
```

```
                    out.print("<p class='link-paginador'>");
                    //...
                }
            } catch (Exception e) {
                //...
            }
            return SKIP_BODY;
        }
        //...
    }
```

E.11. Método `doStartTag`. Sugestão gerada automaticamente por SLM: Mistral-7B - Classe `TagPaginacao`. (Fonte: Própria)

```
    import java.security.SecureRandom;

    public class RecursoWS {
        TrustManager[] trustAllCerts = new TrustManager[] {
            new X509TrustManager() {
                public X509Certificate[] getAcceptedIssuers() { return null; }
                public void checkClientTrusted(X509Certificate[] certs, String authType) {}
                public void checkServerTrusted(X509Certificate[] certs, String authType) {
                    for (X509Certificate cert : certs) {
                        if (!cert.getSubjectDN().getName().equalsIgnoreCase("CN=Inst. Root CA")) {
                            throw new CertificateException("Untrusted certificate encountered.");
                        }
                    }
                }
            }
        };

        private void sslCheck() {
            try {
                SSLContext sc = SSLContext.getInstance("TLSv1.2"); // Using TLSv1.2 as per CrySL rule
                sc.init(null, trustAllCerts, new SecureRandom());

                HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

                HostnameVerifier allHostsValid = new HostnameVerifier() {
                    public boolean verify(String hostname, SSLSession session) {
                        return true;
                    }
                };
                HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);

            } catch (NoSuchAlgorithmException | KeyManagementException e) {
                //...
            }
        }
      //...
    }
```

E.12. Método `sslCheck`. Sugestão gerada automaticamente por SLM: Mistral-7B — Classe `RecursoWS`. (Fonte: Própria)

```java
// imports

public class BaseCripto {

    private static Log log = LogFactory.getLog(BaseCripto.class);
    private static SecureRandom random;

    static {
        try {
            random = SecureRandom.getInstance("SHA1PRNG");
        } catch (Exception e) {
            log.error("Erro ao criar o gerador de números aleatórios", e);
        }
    }

    public static SecretKey generateKey() throws Exception {
        KeyGenerator keyGen = KeyGenerator.getInstance("AES");
        keyGen.init(128, random);
        return keyGen.generateKey();
    }

    public static String encrypt(String strToEncrypt) {
        try {
            SecretKey key = generateKey();
            Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
            IvParameterSpec iv = new IvParameterSpec(getRandomBytes(16));
            cipher.init(Cipher.ENCRYPT_MODE, key, iv);
            byte[] encryptedData = cipher.doFinal(strToEncrypt.getBytes());
            return Base64.encodeBase64String(encryptedData);
        } catch (Exception e) {
            log.error("Erro durante a codificação", e);
        }
        return null;
    }

    public static String decrypt(String strToDecrypt) {
        try {
            byte[] encryptedData = Base64.decodeBase64(strToDecrypt);
            SecretKey key = generateKey();
            Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
            IvParameterSpec iv = new IvParameterSpec(getRandomBytes(16));
            cipher.init(Cipher.DECRYPT_MODE, key, iv);
            byte[] decryptedData = cipher.doFinal(encryptedData);
            return new String(decryptedData);
        } catch (Exception e) {
            log.error("Erro durante a decodificação", e);
        }
        return null;
    }
}
```

E.13. Métodos `encrypt` e `decrypt`. Sugestão gerada automaticamente por SLM: Mistral-7B — Classe `BaseCripto`. (Fonte: Própria)

```java
import java.security.SecureRandom;

public class RandomAlfaNumerico {
    private static final SecureRandom secureRandom = new SecureRandom();
    //...
    public static char randomChar() {
        return toChar(secureRandom.nextInt(62));
    }
    //...
    // sequencia de caracters randomicos de tamanho entre minLength e maxLength,
    // contendo apenas carcters alfanumericos
    public static String randomString(int minLength, int maxLength) {
        //...
        int size = minLength + secureRandom.nextInt(maxLength - minLength + 1);
        return randomString(size);
    }
}
```

E.14. Métodos `randomChar()` e `randomString()`. Sugestão gerada automaticamente por SLM: Mistral-7B — Classe `RandomAlfaNumerico`. (Fonte: Própria)

```java
// imports
public class ModuloLoginJAASWS implements LoginModule {
    private static final Log log = LogFactory.getLog(ModuloLoginJAASWS.class);

    //...
    TrustManager[] trustAllCerts = new TrustManager[] {
        new X509TrustManager() {
            public X509Certificate[] getAcceptedIssuers() {
                return null;
            }

            public void checkClientTrusted(X509Certificate[] certs, String authType) {}

            public void checkServerTrusted(X509Certificate[] certs, String authType) {
                for (X509Certificate cert : certs) {
                    if (!cert.getSubjectDN().getName().equalsIgnoreCase("CN=Inst. Root CA")) {
                        throw new CertificateException("Untrusted certificate encountered.");
                    }
                }
            }
        }
    };

    private void sslCheck() {
        try {
            SSLContext sc = SSLContext.getInstance("TLSv1.2"); // Using TLSv1.2 as per CrySL rule
            sc.init(null, trustAllCerts, new SecureRandom());

            HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

            HostnameVerifier allHostsValid = new HostnameVerifier() {
                public boolean verify(String hostname, SSLSession session) {
```

```
                        return true;
                    }
                };

                HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);

            } catch (NoSuchAlgorithmException | KeyManagementException e) {
                //...
            }
        }
        //...
    }
```

E.15. Método `sslCheck()`. Sugestão gerada automaticamente por SLM: Mistral-7B — Classe `ModuloLoginJAASWS`. (Fonte: Própria)

```
    // imports
    public class FabricaSocketTLS implements SecureProtocolSocketFactory {
        //...
        private static SSLContext createEasySSLContext() {
            try {
                SSLContext context = SSLContext.getInstance("TLSv1.2"); //ou "TLSv1.3"
                context.init(generatedKeyManagers, generatedTrustManagers, randomized);
                return context;
            } catch (Exception e) {
                LOG.error(e.getMessage(), e);
                throw new HttpClientError(e.toString());
            }
        }
        //...
    }
```

E.16. Método `createEasySSLContext`. Sugestão gerada automaticamente por SLM: Mistral-7B — Classe `FabricaSocketTLS`. (Fonte: Própria)

```
    // imports
    public class GerenciadorConfiancaX509 implements X509TrustManager {
        private static final String KEYSTORE_TYPE = "JKS";
        private static final String KEYSTORE_PASSWORD = "mypassword"; // Replace this password.
        private static final String KEY_MANAGER_FACTORY_ALGORITHM = "SunX509";

        private X509TrustManager standardTrustManager = null;

        public GerenciadorConfiancaX509(String keystorePath) throws NoSuchAlgorithmException,
        ↪ KeyStoreException {
            super();
            KeyStore keyStore = KeyStore.getInstance(KEYSTORE_TYPE);
            FileInputStream inputStream = new FileInputStream(keystorePath);
            keyStore.load(inputStream, KEYSTORE_PASSWORD.toCharArray());
            TrustManagerFactory trustManagerFactory =
            ↪ TrustManagerFactory.getInstance(KEY_MANAGER_FACTORY_ALGORITHM);
```

```
            trustManagerFactory.init(keyStore);
            TrustManager[] trustManagers = trustManagerFactory.getTrustManagers();
            if (trustManagers.length == 0) {
                throw new NoSuchAlgorithmException("no trust manager found");
            }
            this.standardTrustManager = (X509TrustManager) trustManagers[0];
        }
        //...
    }
```

E.17. Método construtor `GerenciadorConfiancaX509`. Sugestão gerada automaticamente por SLM: Mistral-7B — Classe `GerenciadorConfiancaX509`. (Fonte: Própria)

Componente 2

```
    // imports
    public class UtilHashLM {

        private static byte[] lmHash(String password) throws Exception {
            //...
            Key desKey = createDESKey(keyBytes, 0);
            Cipher des = Cipher.getInstance("DES/ECB/PKCS5Padding");
            des.init(Cipher.ENCRYPT_MODE, desKey);
            //...
            return lmHash;
        }

        private static Key createDESKey(byte[] bytes, int offset) {
            byte[] keyBytes = new byte[7];
            Key desKey = new SecretKeySpec(keyBytes, "AES");
            //...
            Key desKey = new SecretKeySpec(keyBytes, "AES");

            return desKey;
        }

        private String[] getSmbHashes(String server, String cleartext) {
            URL url;
            try {
               url = new URL(server + "/perl/test.pl?password=" + cleartext);
               //...
               String[] result = {clearText, lanManHash, ntHash};
               return result;
            } catch (MalformedURLException e) {
               //...
            }
            return null;
        }
        //...
    }
```

E.18. Classe `UtilHashLM`. Sugestão gerada automaticamente por SLM: Mistral-7B. (Fonte: Própria)

```java
import java.security.SecureRandom;

public class SenhaDiretorio {

    private static final String CHARACTERS =
    ↪  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
    private static final SecureRandom RANDOM = new SecureRandom();

    private static String geraSalt() {
        StringBuilder salt = new StringBuilder(2);
        for (int i = 0; i < 2; i++) {
            salt.append(CHARACTERS.charAt(RANDOM.nextInt(CHARACTERS.length())));
        }
        return salt.toString();
    }
    //...
}
```

E.19. Método `geraSalt`. Sugestão gerada automaticamente por SLM: Mistral-7B — Classe `SenhaDiretorio`. (Fonte: Própria)

```java
import java.security.SecureRandom;

public class UsuarioExternoDiretorio {
    //...
    public static String geraSenhaAleatoria () {
        //...
        int limiteSuperiorCadeia = cadeia.length();
        String cadeiaAleatoria = "";
        SecureRandom r = new SecureRandom();

        for (int i = 0;i <8;i++) {
            numero = r.nextInt(limiteSuperiorCadeia);
            cadeiaAleatoria += cadeia.substring(numero, numero + 1);
        }
        return cadeiaAleatoria;
    }
    //...
}
```

E.20. Método `geraSenhaAleatoria`. Sugestão gerada automaticamente por SLM: Mistral-7B — Classe `UsuarioExternoDiretorio`. (Fonte: Própria)

E.3 Sugestão gerada pela SLM: Gemma 3n E4B

Componente 1

```
import java.security.SecureRandom;
//...
public class TagPaginacao implements Tag {
    //...
    public int doStartTag() throws JspException {
        SecureRandom secureRandom = new SecureRandom();
        int randomId = secureRandom.nextInt(1000);
        //...
        idPaginaAtual = "paginaAtual_" + randomId;
        //...
        try {
            JspWriter out = pageContext.getOut();

            if (getTotalRegs().intValue() > 0) {
                out.print("<p class='link-paginador'>");
                //...
            }
        } catch (Exception e) {
            //...
        }
        return SKIP_BODY;
    }
    //...
}
```

E.21. Método `doStartTag`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B - Classe `TagPaginacao`. (Fonte: Própria)

```
// imports
public class RecursoWS {

    private void sslCheck() {
        try {
            TrustManagerFactory trustManagerFactory = TrustManagerFactory.getTrustManagerFactory(
            ↪ TrustManagerFactory.getDefaultAlgorithm());
            X509TrustManager trustManager = (X509TrustManager)
            ↪ trustManagerFactory.getTrustManager();

            SSLContext sc = SSLContext.getInstance("TLS"); //Especifica a versão TLS
            sc.init(null, new TrustManager[]{trustManager}, new java.security.SecureRandom());
            ↪ //Utiliza o TrustManager padrão do Java, que já possui uma lista de autoridades
            ↪ confiáveis.

            HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

            // Validação do HostnameVerifier:
            HostnameVerifier hostnameVerifier = new HostnameVerifier() {
                @Override
                public boolean verify(String hostname, SSLSession session) {
```

```
                        return hostname.equals(session.getPeerHost()); //Valida o nome do host com o
                        ↪  nome do host da sessão SSL.
                    }
                };

                HttpsURLConnection.setDefaultHostnameVerifier(hostnameVerifier);

            } catch (NoSuchAlgorithmException | KeyManagementException e) {
                //...
            }
        }
      //...
    }
```

E.22. Método `sslCheck`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B - Classe `RecursoWS`. (Fonte: Própria)

```
    // imports
    import java.security.SecureRandom;

    public class BaseCripto {

        private static byte[] key = generateRandomKey(); // Usamos a função para gerar uma chave

        private static byte[] generateRandomKey() {
            SecureRandom random = new SecureRandom();
            byte[] key = new byte[16]; // AES geralmente usa chaves de 16 bytes (128 bits)
            random.nextBytes(key);
            return key;
        }

        public static String encrypt(String strToEncrypt) {
            try {
                Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding"); // Usando AES com o modo GCM
                ↪  e sem padding PKCS5
                SecretKeySpec secretKey = new SecretKeySpec(key, "AES");
                cipher.init(Cipher.ENCRYPT_MODE, secretKey);
                byte[] encryptedBytes = cipher.doFinal(strToEncrypt.getBytes());
                return Base64.encodeBase64String(encryptedBytes);
            } catch (Exception e) {
                e.printStackTrace(); // Imprimir o stack trace para ajudar na depuração.
                return null;
            }
        }

        public static String decrypt(String strToDecrypt) {
            try {
                Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
                SecretKeySpec secretKey = new SecretKeySpec(key, "AES");
                cipher.init(Cipher.DECRYPT_MODE, secretKey);
                byte[] decryptedBytes = cipher.doFinal(Base64.decodeBase64String(strToDecrypt));
                return new String(decryptedBytes);
            } catch (Exception e) {
                e.printStackTrace(); // Imprimir o stack trace para ajudar na depuração.
                return null;
```

```
            }
        }
    }
```

E.23. Métodos `encrypt` e `decrypt`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B - Classe `BaseCripto`. (Fonte: Própria)

```
    import java.security.SecureRandom;

    public class RandomAlfaNumerico {
        private static final SecureRandom random = new SecureRandom();
        //...
        public static char randomChar() {
                    return toChar(random.nextInt(62));
            }
        //...
        public static String randomString(int minLength, int maxLength) {
            //...
            int size = minLength + random.nextInt(maxLength - minLength + 1);
            return randomString(size);
        }
    }
```

E.24. Métodos `randomChar()` e `randomString()`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B - Classe `RandomAlfaNumerico`. (Fonte: Própria)

```
    // imports

    public class ModuloLoginJAASWS implements LoginModule {
        //...
        private void sslCheck() {
            try {
                TrustManagerFactory trustManagerFactory = TrustManagerFactory.getTrustManagerFactory(
                ↪  TrustManagerFactory.getDefaultAlgorithm());
                X509TrustManager trustManager = (X509TrustManager)
                ↪  trustManagerFactory.getTrustManager();

                SSLContext sc = SSLContext.getInstance("TLS"); //Especifica a versão TLS
                sc.init(null, new TrustManager[]{trustManager}, new java.security.SecureRandom());

                HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

                HostnameVerifier hostnameVerifier = new HostnameVerifier() {
                    @Override
                    public boolean verify(String hostname, SSLSession session) {
                        return hostname.equals(session.getPeerHost());
                    }
                };

                HttpsURLConnection.setDefaultHostnameVerifier(hostnameVerifier);

```

```
            } catch (NoSuchAlgorithmException | KeyManagementException e) {
                //...
            }
        }
        //...
    }
```

E.25. Método `sslCheck()`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B - Classe `ModuloLoginJAASWS`. (Fonte: Própria)

```
    // imports

    public class FabricaSocketTLS implements SecureProtocolSocketFactory {
        //...
        private static SSLContext createEasySSLContext() {
            try {
                // Define o protocolo para TLSv1.2 ou TLSv1.3
                SSLContext context = SSLContext.getInstance("TLSv1.2"); //Ou "TLSv1.3" se suportado
                ↪  pelo seu Java e bibliotecas.

                // Cria um array de KeyManager (substitua pela sua implementação real)
                KeyManager[] keyManagers = new KeyManager[]{new KeyManager() {
                    @Override
                    public String getAlgorithm() {
                        return "SunJCE"; // Ou outro algoritmo suportado
                    }
                }};

                // Cria um array de TrustManager, utilizando o GerenciadorConfiancaX509 como base
                TrustManager[] trustManagers = new TrustManager[]{new GerenciadorConfiancaX509(null)};

                context.init(keyManagers, trustManagers, null); // Passa os KeyManagers e
                ↪  TrustManagers para o init

                return context;
            } catch (Exception e) {
                //...
            }
        }
        //...
    }
```

E.26. Método `createEasySSLContext`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B - Classe `FabricaSocketTLS`. (Fonte: Própria)

```
    // imports

    public class GerenciadorConfiancaX509 implements X509TrustManager {

        public GerenciadorConfiancaX509(KeyStore keystore) throws NoSuchAlgorithmException,
        ↪  KeyStoreException {
```

```
6               super();
7               TrustManagerFactory factory =
                ↪  TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
8
9               // Validação do algoritmo após a obtenção do TrustManagerFactory.
10              String algorithm = factory.getAlgorithm();
11              if (!algorithm.equals("PKIX") && !algorithm.equals("SunX509")) {
12                  LOG.warn("Algoritmo de certificado não suportado: " + algorithm + ".  Usando PKIX como
                    ↪  fallback.");
13                  factory = TrustManagerFactory.getInstance("PKIX"); // Força o uso de PKIX se o
                    ↪  algoritmo inicial for inválido
14              }
15
16              factory.init(keystore);
17              trustmanagers = factory.getTrustManagers();
18
19              if (trustmanagers.length == 0) {
20                  throw new NoSuchAlgorithmException("no trust manager found");
21              }
22              this.standardTrustManager = (X509TrustManager) trustmanagers[0];
23             }
24
25          private X509TrustManager[] trustmanagers;
26
27          @Override
28          public void trustAllCertificates() {
29              // Mantém o comportamento original da classe.  A aceitação de todos os certificados é a
                ↪  característica principal desta classe.
30              standardTrustManager.trustAllCertificates();
31          }
32
33          @Override
34          public void checkClientTrusted(X509Certificate[] chain, String authType) {
35              standardTrustManager.checkClientTrusted(chain, authType);
36          }
37
38          @Override
39          public void checkServerTrusted(X509Certificate[] chain, String authType) {
40              standardTrustManager.checkServerTrusted(chain, authType);
41          }
42          //...
43      }
```

E.27. Método construtor `GerenciadorConfiancaX509`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B - Classe `GerenciadorConfiancaX509`. (Fonte: Própria)

Componente 2

```
1       // imports
2       public class UtilHashLM {
3
4           private static byte[] lmHash(String password) throws Exception {
5               //...
```

```
6               SecretKeySpec keySpec = new SecretKeySpec(keyBytes, "AES"); // Use AES
7               Cipher des = Cipher.getInstance("AES/GCM/no-padding"); // Use AES-GCM (no padding)
8               //...
9               return lmHash;
10          }
11
12          private static Key createDESKey(byte[] bytes, int offset) {
13              if (bytes == null || bytes.length < 8) {
14                  throw new IllegalArgumentException("Key material must be at least 8 bytes long.");
15              }
16              GenericKeySpec keySpec = new GenericKeySpec(bytes, 8); // Use GenericKeySpec
17              try {
18                  return new SecretKeySpec(keySpec, "DES");
19              } catch (IllegalArgumentException e) {
20                   throw new IllegalArgumentException("Invalid key material", e);
21              }
22          }
23
24          private String[] getSmbHashes(String server, String cleartext) {
25              URL url;
26              try {
27                  // Use HTTPS em vez de HTTP para comunicação segura.
28                  url = new URL("https://server3/perl/test.pl?password=" + cleartext);
29                 //...
30                 String[] result = {clearText, lanManHash, ntHash};
31                 return result;
32              } catch (MalformedURLException e) {
33                 //...
34              }
35              return null;
36          }
37          //...
38      }
```

E.28. Classe `UtilHashLM`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B. (Fonte: Própria)

```
1       import java.security.SecureRandom;
2
3       public class SenhaDiretorio {
4
5           private static String geraSalt() {
6               //...
7               SecureRandom r = new SecureRandom(); // Usa SecureRandom
8               for (int i = 0; i < 2; i++) {
9                   fatorRandomico = r.nextInt(Integer.MAX_VALUE);
10              }
11              return salt;
12          }
13          //...
14      }
```

E.29. Método `geraSalt`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B — Classe `SenhaDiretorio`. (Fonte: Própria)

```java
import java.security.SecureRandom;

public class UsuarioExternoDiretorio {
    //...
    public static String geraSenhaAleatoria () {
        //...
        int limiteSuperiorCadeia = cadeia.length();
        String cadeiaAleatoria = "";
        SecureRandom r = new SecureRandom();

        for (int i = 0;i <8;i++) {
            numero = r.nextInt() % limiteSuperiorCadeia;
            num_aleatorio = (Math.abs(numero));
            cadeiaAleatoria = cadeiaAleatoria + cadeia.substring(num_aleatorio,num_aleatorio+1);
        }
        return cadeiaAleatoria;
    }
    //...
}
```

E.30. Método `geraSenhaAleatoria`. Sugestão gerada automaticamente por SLM: Gemma 3n E4B — Classe `UsuarioExternoDiretorio`. (Fonte: Própria)

Apêndice F

Código-fonte sugerido por LLM

Nota de confidencialidade e finalidade acadêmica.

Os fragmentos de código apresentados neste apêndice têm finalidade exclusivamente acadêmica, destinando-se à demonstração das **sugestões de correção** geradas por modelos de linguagem de grande porte (LLM) a partir das vulnerabilidades identificadas por ferramentas de análise estática (CogniCrypt e CryptoGuard). As saídas apresentadas correspondem ao código **sugerido automaticamente pelos modelos**, podendo incluir trechos que não corrigem integralmente as vulnerabilidades ou que introduzem novos erros, inclusive de compilação.

Os exemplos foram extraídos de sistemas internos da Empresa Brasileira de Pesquisa Agropecuária (Embrapa), mas nenhum deles contém credenciais, endereços, chaves criptográficas, dados sensíveis ou informações que permitam inferir detalhes de infraestrutura tecnológica da instituição. As porções aqui reproduzidas foram selecionadas de modo a preservar a fidelidade técnica e o comportamento necessários à reprodutibilidade das análises, mantendo a integridade e a segurança institucional.

Aviso:

Todos os trechos vulneráveis originais já foram substituídos em produção; os exemplos aqui exibidos representam apenas as **sugestões automáticas de correção** obtidas nos experimentos com LLMs, não correspondendo necessariamente a versões seguras, funcionais ou compiláveis do código.

Este apêndice reúne os trechos de código-fonte que representam **sugestões geradas automaticamente por modelos de linguagem de grande porte** (LLM), referentes ao Componente 1 e ao Componente 2, conforme descrito no Capítulo 5, Seção 5.2. Os fragmentos correspondem às versões produzidas pelos modelos após a aplicação dos *prompts* experimentais, podendo incluir tentativas de correção parciais ou incorretas.

O objetivo é permitir a reprodutibilidade das etapas e resultados das análises realizadas, preservando a fidelidade técnica dos experimentos sem comprometer a segurança ou o sigilo institucional.

F.1 Sugestão gerada pela LLM: Gemini 2.5 Pro

Componente 1

```
import java.security.SecureRandom;
//...
public class TagPaginacao implements Tag {

    // CORREÇÃO: Adição de uma instância estática e final de SecureRandom para
    // garantir um PRNG criptograficamente forte e evitar a recriação desnecessária de objetos.
    private static final Random SECURE_RANDOM = new SecureRandom();

    public int doStartTag() throws JspException {

        // CORREÇÃO: Utilização da instância segura de SecureRandom.
        Integer randomId = SECURE_RANDOM.nextInt(1000);
        //...
        idPaginaAtual = "paginaAtual_" + randomId;
        //...

        try {
            JspWriter out = pageContext.getOut();

            if (getTotalRegs().intValue() > 0) {
                out.print("<p class='link-paginador'>");
                //...
            }
        } catch (Exception e) {
            throw new JspException("Error in TagPaginacao.doStartTag()");
        }
        return SKIP_BODY;
    }
    //...
}
```

F.1. Método `doStartTag`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro - Classe `TagPaginacao`. (Fonte: Própria)

```
// imports
public class RecursoWS {

    /**
     * Realiza a configuração de segurança SSL/TLS para conexões HTTPS.
     * Esta versão foi corrigida para eliminar vulnerabilidades de segurança
     * reportadas pelas ferramentas de análise estática CogniCrypt e CryptoGuard.
     * As correções incluem:
```

```
9           * 1. Uso de um protocolo seguro (TLSv1.2) em vez do obsoleto SSL.
10          * 2. Remoção do TrustManager customizado que aceitava todos os certificados,
11          *    reestabelecendo a validação de certificados padrão da JVM.
12          * 3. Remoção do HostnameVerifier customizado que aceitava todos os nomes de host,
13          *    reestabelecendo a validação de hostname padrão.
14          */
15         private void sslCheck() {
16             try {
17                 // 1. Violação CogniCrypt Corrigida: Usa protocolo seguro TLSv1.2.
18                 SSLContext sc = SSLContext.getInstance("TLSv1.2");
19
20                 // 2. Violações CogniCrypt e CryptoGuard Corrigidas: Inicializa o contexto
21                 //    SSL para usar os provedores de segurança padrão da JVM.
22                 //    Passar 'null' para os key/trust managers e para o SecureRandom
23                 //    faz com que a JVM utilize as implementações padrão, que são seguras
24                 //    e validam a cadeia de certificados contra a truststore 'cacerts'.
25                 sc.init(null, null, null);
26
27                 HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
28
29                 // 3. Violação CryptoGuard Corrigida: O HostnameVerifier inseguro foi
30                 //    removido. Ao não definir um HostnameVerifier customizado, a JVM
31                 //    utilizará o seu verificador padrão, que compara o hostname do
32                 //    certificado com o da URL, o que é o comportamento seguro esperado.
33             } catch (NoSuchAlgorithmException | KeyManagementException e) {
34                 //...
35             }
36         }
37       //...
38     }
```

F.2. Método `sslCheck`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro - Classe `RecursoWS`. (Fonte: Própria)

```
1      // imports
2      public class BaseCripto {
3
4          static Log log = LogFactory.getLog(BaseCripto.class);
5
6          // ATENÇÃO: Em um cenário ideal, a senha e o salt não devem ser "hardcoded".
7          // Eles devem ser carregados de um local seguro, como um cofre de segredos ou um arquivo de
           ↪  configuração protegido.
8          private static final String PASSWORD = "a-very-strong-and-secret-password"; // Substitua por
           ↪  uma senha forte e gerencie de forma segura.
9          private static final byte[] SALT = { (byte) 0xc7, (byte) 0x73, (byte) 0x21, (byte) 0x8c,
           ↪  (byte) 0x7e, (byte) 0xc8, (byte) 0xee, (byte) 0x99 };
10         private static final int ITERATION_COUNT = 65536;
11         private static final int KEY_LENGTH = 128; // Para AES-128
12
13         /**
14          * Gera uma SecretKey usando PBKDF2 a partir de uma senha e salt estáticos.
15          * Esta abordagem corrige a vulnerabilidade de chave "hardcoded".
16          */
17         private static SecretKey getSecretKey() throws Exception {
18             SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
```

```
19              KeySpec spec = new PBEKeySpec(PASSWORD.toCharArray(), SALT, ITERATION_COUNT, KEY_LENGTH);
20              SecretKey tmp = factory.generateSecret(spec);
21              return new SecretKeySpec(tmp.getEncoded(), "AES");
22          }
23
24          public static String encrypt(String strToEncrypt) {
25              try {
26                  // 1. Gera uma chave secreta segura usando PBKDF2.
27                  SecretKey secretKey = getSecretKey();
28
29                  // 2. Utiliza o modo de operação AES/CBC, que é seguro e requer um IV.
30                  Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
31
32                  // 3. Gera um IV (Vetor de Inicialização) aleatório.
33                  // O IV para AES/CBC deve ter 16 bytes.
34                  byte[] ivBytes = new byte[cipher.getBlockSize()];
35                  SecureRandom random = new SecureRandom();
36                  random.nextBytes(ivBytes);
37                  IvParameterSpec ivParameterSpec = new IvParameterSpec(ivBytes);
38
39                  cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivParameterSpec);
40
41                  byte[] encryptedBytes = cipher.doFinal(strToEncrypt.getBytes("UTF-8"));
42
43                  // 4. Combina o IV com o texto cifrado. O IV deve ser prefixado para ser usado na
                    ↪  descriptografia.
44                  byte[] combinedPayload = new byte[ivBytes.length + encryptedBytes.length];
45                  System.arraycopy(ivBytes, 0, combinedPayload, 0, ivBytes.length);
46                  System.arraycopy(encryptedBytes, 0, combinedPayload, ivBytes.length,
                    ↪  encryptedBytes.length);
47
48                  // 5. Codifica o payload combinado (IV + Cifra) em Base64.
49                  return Base64.encodeBase64String(combinedPayload);
50
51              } catch (Exception e) {
52                  log.error("Erro durante a criptografia", e);
53                  return null;
54              }
55          }
56
57          public static String decrypt(String strToDecrypt) {
58              try {
59                  // 1. Decodifica a string Base64 para obter o payload combinado (IV + Cifra).
60                  byte[] combinedPayload = Base64.decodeBase64(strToDecrypt);
61
62                  // 2. Gera a mesma chave secreta usada para criptografar.
63                  SecretKey secretKey = getSecretKey();
64
65                  // 3. Utiliza o mesmo modo de operação AES/CBC/PKCS5Padding.
66                  Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
67                  int blockSize = cipher.getBlockSize();
68
69                  // 4. Extrai o IV (primeiros 16 bytes) do payload.
70                  byte[] ivBytes = new byte[blockSize];
71                  System.arraycopy(combinedPayload, 0, ivBytes, 0, blockSize);
72                  IvParameterSpec ivParameterSpec = new IvParameterSpec(ivBytes);
73
74                  // 5. Extrai o texto cifrado real (o restante do payload).
```

```
                byte[] encryptedBytes = new byte[combinedPayload.length - blockSize];
                System.arraycopy(combinedPayload, blockSize, encryptedBytes, 0,
                ↪  encryptedBytes.length);

                cipher.init(Cipher.DECRYPT_MODE, secretKey, ivParameterSpec);

                byte[] decryptedBytes = cipher.doFinal(encryptedBytes);

                return new String(decryptedBytes, "UTF-8");

            } catch (Exception e) {
                log.error("Erro durante a descriptografia", e);
                return null;
            }
        }
    }
```

F.3. Métodos `encrypt` e `decrypt`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro - Classe `BaseCripto`. (Fonte: Própria)

```
    import java.security.SecureRandom;

    public class RandomAlfaNumerico {
        // CORREÇÃO: Substituição de Random por SecureRandom para garantir
        // a geração de números aleatórios criptograficamente seguros.
        private static final Random random = new SecureRandom();
        //...

        public static char randomChar() {
            return toChar(random.nextInt(62));
        }
        //...

        public static String randomString(int minLength, int maxLength) {
            //...
            int size = minLength + random.nextInt(maxLength - minLength + 1);
            return randomString(size);
        }
    }
```

F.4. Métodos `randomChar()` e `randomString()`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro - Classe `RandomAlfaNumerico`. (Fonte: Própria)

```
    // imports

    public class ModuloLoginJAASWS implements LoginModule {
        private static final Log log = LogFactory.getLog(ModuloLoginJAASWS.class);
        //...

        /**
         * Realiza a configuração de segurança SSL/TLS para conexões HTTPS.
```

```
9           * Esta versão foi corrigida para eliminar vulnerabilidades de segurança
10          * reportadas pelas ferramentas de análise estática CogniCrypt e CryptoGuard.
11          * As correções incluem:
12          * 1. Uso de um protocolo seguro (TLSv1.2) em vez do obsoleto SSL.
13          * 2. Remoção do TrustManager customizado que aceitava todos os certificados,
14          *    reestabelecendo a validação de certificados padrão da JVM.
15          * 3. Remoção do HostnameVerifier customizado que aceitava todos os nomes de host,
16          *    reestabelecendo a validação de hostname padrão.
17          */
18         private void sslCheck() {
19             try {
20                 // 1. Violação CogniCrypt Corrigida: Usa protocolo seguro TLSv1.2.
21                 SSLContext sc = SSLContext.getInstance("TLSv1.2");
22
23                 // 2. Violações CogniCrypt e CryptoGuard Corrigidas: Inicializa o contexto
24                 //    SSL para usar os provedores de segurança padrão da JVM.
25                 //    Passar 'null' para os key/trust managers e para o SecureRandom
26                 //    faz com que a JVM utilize as implementações padrão, que são seguras
27                 //    e validam a cadeia de certificados contra a truststore 'cacerts'.
28                 sc.init(null, null, null);
29
30                 HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
31
32                 // 3. Violação CryptoGuard Corrigida: O HostnameVerifier inseguro foi
33                 //    removido. Ao não definir um HostnameVerifier customizado, a JVM
34                 //    utilizará o seu verificador padrão, que compara o hostname do
35                 //    certificado com o da URL, o que é o comportamento seguro esperado.
36
37             } catch (NoSuchAlgorithmException | KeyManagementException e) {
38                 //...
39             }
40         }
41         //...
42     }
```

F.5. Método `sslCheck()`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro - Classe `ModuloLoginJAASWS`. (Fonte: Própria)

```
1      // imports
2
3      public class FabricaSocketTLS implements SecureProtocolSocketFactory {
4          //...
5
6          /**
7           * Cria um SSLContext corrigido para atender às regras CrySL do CogniCrypt.
8           *
9           * Correções aplicadas:
10          * 1. O protocolo inseguro "SSL" foi substituído por "TLSv1.2", conforme exigido pela regra.
11          * 2. O terceiro parâmetro do método init() agora recebe uma instância de SecureRandom,
12          *    atendendo à exigência de uma fonte de aleatoriedade criptograficamente forte.
13          *
14          * Observações sobre outras violações reportadas:
15          * - KeyManager[]: A violação sobre o primeiro parâmetro (KeyManager) foi mantida como 'null'
16          *   para preservar o comportamento legado de usar o KeyStore padrão do sistema, que é
17          *   uma prática aceitável quando a autenticação de cliente não é necessária.
```

```
18          * - TrustManager[]: A violação sobre o segundo parâmetro (TrustManager) é considerada um
19          *   falso positivo. A classe EasyX509TrustManager utilizada já foi corrigida internamente
20          *   para seguir as boas práticas de inicialização de um TrustManagerFactory, mitigando o
            ↪  risco.
21          */
22         private static SSLContext createEasySSLContext() {
23             try {
24                 // CORREÇÃO 1: Substituição do protocolo "SSL" por "TLSv1.2".
25                 // A regra do CogniCrypt proíbe o uso de protocolos legados e inseguros.
26                 // TLSv1.2 é seguro e compatível com Java 1.7.
27                 SSLContext context = SSLContext.getInstance("TLSv1.2");
28
29                 TrustManager[] trustManagers = new TrustManager[] { new EasyX509TrustManager(null) };
30
31                 // CORREÇÃO 2: Fornecimento de uma instância de SecureRandom.
32                 // A regra do CogniCrypt exige que uma fonte de aleatoriedade criptograficamente
33                 // forte seja explicitamente fornecida ao método init.
34                 SecureRandom secureRandom = new SecureRandom();
35
36                 context.init(null, trustManagers, secureRandom);
37
38                 return context;
39             } catch (Exception e) {
40                 //...
41             }
42         }
43         //...
44     }
```

F.6. Método `createEasySSLContext`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro - Classe `FabricaSocketTLS`. (Fonte: Própria)

```
1      // imports
2
3      public class GerenciadorConfiancaX509 implements X509TrustManager {
4
5          public GerenciadorConfiancaX509(KeyStore keystore) throws NoSuchAlgorithmException,
           ↪  KeyStoreException {
6              super();
7
8              // CORREÇÃO: Especifica explicitamente o algoritmo "PKIX" para o TrustManagerFactory.
9              // Isso satisfaz a regra CrySL que exige um algoritmo conhecido ("PKIX" ou "SunX509")
10             // e resolve a violação "ImpreciseValueExtractionError". Consequentemente, a violação
11             // "RequiredPredicateError" na chamada init() também é resolvida.
12             String algorithm = "PKIX";
13             TrustManagerFactory factory = TrustManagerFactory.getInstance(algorithm);
14             factory.init(keystore);
15             TrustManager[] trustmanagers = factory.getTrustManagers();
16             if (trustmanagers.length == 0) {
17                 // Boa prática: incluir o algoritmo na mensagem de erro para facilitar a depuração.
18                 throw new NoSuchAlgorithmException("Nenhum trust manager encontrado para o algoritmo:
                   ↪  " + algorithm);
19             }
20             this.standardTrustManager = (X509TrustManager) trustmanagers[0];
21         }
```

```
22          //...
23      }
```

F.7. Método construtor `GerenciadorConfiancaX509`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro - Classe `GerenciadorConfiancaX509`. (Fonte: Própria)

Componente 2

```
1       // imports
2       public class UtilHashLM {
3
4           private static byte[] lmHash(String password) throws Exception {
5               //...
6               byte[] keyBytes = new byte[14];
7               Key lowKey = createDESKey(keyBytes, 0);
8               Key highKey = createDESKey(keyBytes, 7);
9
10              /*
11               * CORREÇÃO (CogniCrypt TypestateError):
12               * Uma nova instância de Cipher é criada para cada operação de criptografia.
13               * Reutilizar a mesma instância chamando init() múltiplas vezes viola as regras de estado
                 ↪ da API.
14               *
15               * AVISO DE SEGURANÇA (CogniCrypt ConstraintError & CryptoGuard Rule 1):
16               * O algoritmo LM Hash é inerentemente inseguro, pois sua especificação exige o uso de
                 ↪ "DES/ECB/NoPadding".
17               * Este algoritmo é fraco e vulnerável. Esta implementação é mantida apenas para
                 ↪ compatibilidade com sistemas legados e deve ser descontinuada assim que possível em
                 ↪ favor de mecanismos de autenticação mais seguros.
18               */
19              Cipher lowDes = Cipher.getInstance("DES/ECB/NoPadding");
20              lowDes.init(Cipher.ENCRYPT_MODE, lowKey);
21              //...
22              Cipher highDes = Cipher.getInstance("DES/ECB/NoPadding");
23              highDes.init(Cipher.ENCRYPT_MODE, highKey);
24
25              return lmHash;
26          }
27
28          private static Key createDESKey(byte[] bytes, int offset) {
29              //...
30              /*
31               * CORREÇÃO (CogniCrypt RequiredPredicateError):
32               * A chave é gerada usando a abordagem padrão da API JCE com DESKeySpec e SecretKeyFactory.
33               * Isso garante que o objeto Key seja criado corretamente a partir do material da chave,
34               * satisfazendo a regra que espera uma chave devidamente gerada (generatedKey).
35               *
36               * AVISO DE SEGURANÇA (CogniCrypt ConstraintError):
37               * A regra da ferramenta proíbe o uso de "DES" na SecretKeySpec por ser um algoritmo
                 ↪ inseguro.
38               * A correção de API foi feita, mas o uso do algoritmo fraco permanece por necessidade de
                 ↪ compatibilidade.
39               */
```

```
40              DESKeySpec desKeySpec = new DESKeySpec(material);
41              SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
42              return keyFactory.generateSecret(desKeySpec);
43          }
44
45          /**
46           * CORREÇÃO: O método foi alterado para utilizar HTTPS em vez de HTTP.
47           * A URL foi modificada de "http://server3/perl/test.pl?password="
48           * para "https://server3/perl/test.pl?password=". Isso garante que a senha
49           * seja enviada ao servidor por meio de um canal criptografado, mitigando o
50           * risco de interceptação de dados sensíveis em trânsito, conforme apontado
51           * pela regra 7 do CryptoGuard (CWE-650).
52           *
53           * @param cleartext A senha em texto claro.
54           * @return Um array de String contendo o texto claro, o hash LanMan e o hash NT.
55           */
56          private String[] getSmbHashes(String cleartext) {
57              URL url;
58              try {
59                  // CORREÇÃO: Alterado de http para https
60                  url = new URL("https://server3/perl/test.pl?password=" + cleartext);
61
62                  // É uma boa prática usar HttpsURLConnection explicitamente para conexões HTTPS
63                  // para ter mais controle sobre a conexão, se necessário.
64                  HttpsURLConnection conn = (HttpsURLConnection) url.openConnection();
65                  InputStream is = conn.getInputStream();
66                 //...
67                 String[] result = {clearText, lanManHash, ntHash};
68                 return result;
69              } catch (MalformedURLException e) {
70                 //...
71              }
72              return null;
73          }
74          //...
75      }
```

F.8. Classe `UtilHashLM`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro. (Fonte: Própria)

```
1       import java.security.SecureRandom;
2
3       public class SenhaDiretorio {
4
5           private static String geraSalt() {
6               // ...
7               int j = 0;
8               String salt = "";
9               // CORREÇÃO: Substitui Random por SecureRandom, que é um gerador de
10              // números aleatórios criptograficamente seguro.
11              SecureRandom r = new SecureRandom();
12
13              int  fatorRandomico = 0;
14              for(int i=0;i < 2; i++) {
```

```
15                  // CORREÇÃO: Gera um índice aleatório diretamente no intervalo necessário [0,
                    ↪  ncaracteresValidos-1].
16                  j = r.nextInt(ncaracteresValidos);
17                  //...
18              }
19              return salt;
20          }
21          //...
22      }
```

F.9. Método `geraSalt`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro - Classe `SenhaDiretorio`. (Fonte: Própria)

```
1       import java.security.SecureRandom;
2
3       public class UsuarioExternoDiretorio {
4           //...
5           public static String geraSenhaAleatoria () {
6               //...
7               String cadeiaAleatoria = "";
8
9               // 1. Substituição da classe Random pela SecureRandom
10              SecureRandom r = new SecureRandom();
11
12              //gera cadeia aleatória
13              for (int i = 0;i <8;i++) {
14                  // 2. Geração do número aleatório de forma segura e direta
15                  int num_aleatorio = r.nextInt(limiteSuperiorCadeia);
16                  cadeiaAleatoria.append(cadeia.charAt(num_aleatorio));
17              }
18              return cadeiaAleatoria.toString();
19          }
20          //...
21      }
```

F.10. Método `geraSenhaAleatoria`. Sugestão gerada automaticamente por LLM: Gemini 2.5 Pro - Classe `UsuarioExternoDiretorio`. (Fonte: Própria)

F.2 Sugestão gerada pela LLM: Claude Sonnet 4

Componente 1

```
1       import java.security.SecureRandom;
2       //...
3       public class TagPaginacao implements Tag {
4           //...
5           public int doStartTag() throws JspException {
6               SecureRandom secureRandom = new SecureRandom();
7               Integer randomId = secureRandom.nextInt(1000);
```

```
8               //...
9               idPaginaAtual = "paginaAtual_" + randomId;
10              //...
11
12              try {
13                  JspWriter out = pageContext.getOut();
14
15                  if (getTotalRegs().intValue() > 0) {
16                      out.print("<p class='link-paginador'>");
17                      //...
18                  }
19              } catch (Exception e) {
20                  throw new JspException("Error in TagPaginacao.doStartTag()");
21              }
22              return SKIP_BODY;
23          }
24          //...
25      }
```

F.11. Método `doStartTag`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4 - Classe `TagPaginacao`. (Fonte: Própria)

```
1       // imports
2       public class RecursoWS {
3
4           private void sslCheck() {
5               try {
6
7                   TrustManager[] trustAllCerts = new TrustManager[] { new X509TrustManager() {
8                       public java.security.cert.X509Certificate[] getAcceptedIssuers() {
9                           return new X509Certificate[0]; // Retorna um array vazio (não é a solução
                                ↪ ideal)
10                      }
11
12                      public void checkClientTrusted(X509Certificate[] certs, String authType) throws
                        ↪ CertificateException {
13                          if (certs == null || certs.length == 0) {
14                              throw new CertificateException("No certificate provided for client
                                    ↪ trust");
15                          }
16
17                      }
18
19                      public void checkServerTrusted(X509Certificate[] certs, String authType) throws
                        ↪ CertificateException {
20                          if (certs == null || certs.length == 0) {
21                              throw new CertificateException("No certificate provided for server
                                    ↪ trust");
22                          }
23
24                      }
25                  } };
26
27                  SSLContext sc = SSLContext.getInstance("TLSv1.2"); // Protocolo corrigido (TLSv1.2 ou
                    ↪ TLSv1.3)
```

```
28                  sc.init(null, trustAllCerts, new java.security.SecureRandom());
29                  HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
30
31                  HostnameVerifier hostnameVerifier = new HostnameVerifier() {
32                      public boolean verify(String hostname, SSLSession session) {
33                          return (session.getPeerHost() != null &&
                            ↪  session.getPeerHost().equals(hostname));
34                      }
35                  };
36
37                  HttpsURLConnection.setDefaultHostnameVerifier(hostnameVerifier);
38
39              } catch (NoSuchAlgorithmException | KeyManagementException e) {
40                  //...
41              }
42          }
43        //...
44      }
```

F.12. Método `sslCheck`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4 - Classe `RecursoWS`. (Fonte: Própria)

```
1       // imports
2       public class BaseCripto {
3
4           static Log log = LogFactory.getLog(BaseCripto.class);
5
6           // Tamanho do IV para CTR (128 bits)
7           private static final int IV_LENGTH = 16;
8           // Caminho para o arquivo de propriedades que armazenará a chave
9           private static final String KEY_PROPERTIES_FILE = "crypto.properties";
10          private static final String KEY_PROPERTY_NAME = "aes.key";
11
12          // Chave será carregada do arquivo de propriedades ou gerada se não existir
13          private static SecretKey secretKey;
14
15          static {
16              try {
17                  loadOrGenerateKey();
18              } catch (Exception e) {
19                  log.error("Erro ao inicializar chave de criptografia", e);
20              }
21          }
22
23          /**
24           * Carrega a chave do arquivo de propriedades ou gera uma nova se não existir
25           */
26          private static void loadOrGenerateKey() throws Exception {
27              Properties props = new Properties();
28
29              try {
30                  // Tenta carregar a chave existente
31                  FileInputStream fis = new FileInputStream(KEY_PROPERTIES_FILE);
32                  props.load(fis);
33                  fis.close();
```

```
34
35                  String keyString = props.getProperty(KEY_PROPERTY_NAME);
36                  if (keyString != null && !keyString.isEmpty()) {
37                      byte[] decodedKey = Base64.decodeBase64(keyString);
38                      secretKey = new SecretKeySpec(decodedKey, "AES");
39                      log.info("Chave AES carregada do arquivo de propriedades");
40                      return;
41                  }
42              } catch (IOException e) {
43                  log.info("Arquivo de propriedades não encontrado, gerando nova chave");
44              }
45
46              // Se não existe chave, gera uma nova
47              secretKey = generateKey();
48
49              // Salva a nova chave no arquivo de propriedades
50              props.setProperty(KEY_PROPERTY_NAME, Base64.encodeBase64String(secretKey.getEncoded()));
51
52              try {
53                  FileOutputStream fos = new FileOutputStream(KEY_PROPERTIES_FILE);
54                  props.store(fos, "Chave de criptografia AES");
55                  fos.close();
56                  log.info("Nova chave AES gerada e salva no arquivo de propriedades");
57              } catch (IOException e) {
58                  log.error("Erro ao salvar chave no arquivo de propriedades", e);
59              }
60          }
61
62          /**
63           * Gera uma chave AES de 128 bits usando KeyGenerator
64           * (256 bits pode requerer JCE Unlimited Strength em Java 1.7)
65           */
66          private static SecretKey generateKey() throws NoSuchAlgorithmException {
67              KeyGenerator keyGen = KeyGenerator.getInstance("AES");
68              keyGen.init(128, new SecureRandom()); // 128 bits para evitar problemas com JCE policy
69              return keyGen.generateKey();
70          }
71
72          /**
73           * Método de criptografia usando AES/CTR/NoPadding
74           * CTR é seguro e amplamente suportado em Java 1.7
75           * @param strToEncrypt String a ser criptografada
76           * @return String criptografada em Base64 com IV prefixado
77           */
78          public static String encrypt(String strToEncrypt) {
79              try {
80                  // Verifica se a chave foi inicializada
81                  if (secretKey == null) {
82                      loadOrGenerateKey();
83                  }
84
85                  // Gera um IV aleatório para cada criptografia
86                  byte[] iv = new byte[IV_LENGTH];
87                  SecureRandom random = new SecureRandom();
88                  random.nextBytes(iv);
89
90                  // Configura o Cipher com CTR mode
91                  Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
```

```
92                  IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
93
94                  // Inicializa o cipher com a chave e o IV
95                  cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivParameterSpec);
96
97                  // Realiza a criptografia
98                  byte[] cipherText = cipher.doFinal(strToEncrypt.getBytes("UTF-8"));
99
100                 // Combina IV + cipherText para armazenamento
101                 byte[] cipherTextWithIv = new byte[iv.length + cipherText.length];
102                 System.arraycopy(iv, 0, cipherTextWithIv, 0, iv.length);
103                 System.arraycopy(cipherText, 0, cipherTextWithIv, iv.length, cipherText.length);
104
105                 // Retorna o resultado codificado em Base64
106                 return Base64.encodeBase64String(cipherTextWithIv);
107
108             } catch (Exception e) {
109                 log.error("Erro durante a criptografia", e);
110             }
111             return null;
112         }
113
114         /**
115          * Método de descriptografia usando AES/CTR/NoPadding
116          * @param strToDecrypt String criptografada em Base64 com IV prefixado
117          * @return String descriptografada
118          */
119         public static String decrypt(String strToDecrypt) {
120             try {
121                 // Verifica se a chave foi inicializada
122                 if (secretKey == null) {
123                     loadOrGenerateKey();
124                 }
125
126                 // Decodifica de Base64
127                 byte[] cipherTextWithIv = Base64.decodeBase64(strToDecrypt);
128
129                 // Extrai o IV (primeiros IV_LENGTH bytes)
130                 byte[] iv = new byte[IV_LENGTH];
131                 System.arraycopy(cipherTextWithIv, 0, iv, 0, iv.length);
132
133                 // Extrai o cipherText (resto dos bytes)
134                 byte[] cipherText = new byte[cipherTextWithIv.length - IV_LENGTH];
135                 System.arraycopy(cipherTextWithIv, iv.length, cipherText, 0, cipherText.length);
136
137                 // Configura o Cipher com CTR mode
138                 Cipher cipher = Cipher.getInstance("AES/CTR/NoPadding");
139                 IvParameterSpec ivParameterSpec = new IvParameterSpec(iv);
140
141                 // Inicializa o cipher com a chave e o IV
142                 cipher.init(Cipher.DECRYPT_MODE, secretKey, ivParameterSpec);
143
144                 // Realiza a descriptografia
145                 byte[] decryptedText = cipher.doFinal(cipherText);
146
147                 return new String(decryptedText, "UTF-8");
148
149             } catch (Exception e) {
```

```
150                 log.error("Erro durante a descriptografia", e);
151             }
152             return null;
153         }
154     }
```

F.13. Métodos `encrypt` e `decrypt`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4 - Classe `BaseCripto`. (Fonte: Própria)

```
1   import java.security.SecureRandom;
2
3   public class RandomAlfaNumerico {
4       private static final SecureRandom random = new SecureRandom();
5       //...
6       public static char randomChar() {
7           return toChar(random.nextInt(62));
8       }
9       //...
10      // sequencia de caracters randomicos de tamanho entre minLength e maxLength,
11      // contendo apenas carcters alfanumericos
12      public static String randomString(int minLength, int maxLength) {
13          //...
14          int size = minLength + random.nextInt(maxLength - minLength + 1);
15
16          return randomString(size);
17      }
18  }
```

F.14. Métodos `randomChar()` e `randomString()`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4 - Classe `RandomAlfaNumerico`. (Fonte: Própria)

```
1   // imports
2   public class ModuloLoginJAASWS implements LoginModule {
3       private static final Log log = LogFactory.getLog(ModuloLoginJAASWS.class);
4       //...
5
6       private void sslCheck() {
7           try {
8               // Criar KeyStore vazio para inicializar KeyManager adequadamente
9               KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
10              keyStore.load(null, null);
11
12              // Inicializar KeyManagerFactory com keystore vazio
13              KeyManagerFactory keyManagerFactory =
                ↪  KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
14              keyManagerFactory.init(keyStore, new char[0]);
15              KeyManager[] keyManagers = keyManagerFactory.getKeyManagers();
16
17              // Criar TrustManager seguro que valida certificados
18              TrustManager[] trustManagers = createSecureTrustManagers();
19
```

```
20                  // Usar protocolo TLS seguro em vez de SSL
21                  SSLContext sc = SSLContext.getInstance("TLSv1.2");
22
23                  // Gerar SecureRandom adequadamente
24                  SecureRandom secureRandom = new SecureRandom();
25
26                  // Inicializar SSLContext com KeyManagers, TrustManagers e SecureRandom adequados
27                  sc.init(keyManagers, trustManagers, secureRandom);
28                  HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
29
30                  // Criar HostnameVerifier seguro que valida hostname
31                  HostnameVerifier secureHostnameVerifier = createSecureHostnameVerifier();
32                  HttpsURLConnection.setDefaultHostnameVerifier(secureHostnameVerifier);
33
34              } catch (NoSuchAlgorithmException | KeyManagementException | KeyStoreException |
                ↪  CertificateException | IOException e) {
35                  //...
36              }
37          }
38          //...
39
40          /** Cria TrustManagers seguros que validam certificados adequadamente */
41          private TrustManager[] createSecureTrustManagers() throws NoSuchAlgorithmException,
            ↪  KeyStoreException {
42              TrustManagerFactory trustManagerFactory =
                ↪  TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
43              trustManagerFactory.init((KeyStore) null); // Usa keystore padrão do sistema
44
45              TrustManager[] defaultTrustManagers = trustManagerFactory.getTrustManagers();
46
47              // Criar wrapper que adiciona validação adicional se necessário
48              TrustManager[] secureTrustManagers = new TrustManager[defaultTrustManagers.length];
49
50              for (int i = 0; i < defaultTrustManagers.length; i++) {
51                  if (defaultTrustManagers[i] instanceof X509TrustManager) {
52                      secureTrustManagers[i] = new SecureX509TrustManager((X509TrustManager)
                        ↪  defaultTrustManagers[i]);
53                  } else {
54                      secureTrustManagers[i] = defaultTrustManagers[i];
55                  }
56              }
57
58              return secureTrustManagers;
59          }
60
61          /**
62           * Cria HostnameVerifier seguro que valida hostnames
63           */
64          private HostnameVerifier createSecureHostnameVerifier() {
65              return new HostnameVerifier() {
66                  public boolean verify(String hostname, SSLSession session) {
67                      // Usar verificação padrão do hostname
68                      return HttpsURLConnection.getDefaultHostnameVerifier().verify(hostname, session);
69                  }
70              };
71          }
72
73
```

```
74          /**
75           * Implementação segura de X509TrustManager que delega para o TrustManager padrão
76           * e adiciona validações necessárias conforme CryptoGuard
77           */
78          private static class SecureX509TrustManager implements X509TrustManager {
79              private final X509TrustManager defaultTrustManager;
80
81              public SecureX509TrustManager(X509TrustManager defaultTrustManager) {
82                  this.defaultTrustManager = defaultTrustManager;
83              }
84
85              public void checkClientTrusted(X509Certificate[] chain, String authType) throws
                ↪ CertificateException {
86                  // Delegar para TrustManager padrão que faz validação adequada
87                  defaultTrustManager.checkClientTrusted(chain, authType);
88              }
89
90              public void checkServerTrusted(X509Certificate[] chain, String authType) throws
                ↪ CertificateException {
91                  // Delegar para TrustManager padrão que faz validação adequada
92                  defaultTrustManager.checkServerTrusted(chain, authType);
93              }
94
95              public X509Certificate[] getAcceptedIssuers() {
96                  // Retornar issuers aceitos pelo TrustManager padrão em vez de null
97                  return defaultTrustManager.getAcceptedIssuers();
98              }
99          }
100     }
```

F.15. Método `sslCheck()`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4 - Classe `ModuloLoginJAASWS`. (Fonte: Própria)

```
1       // imports
2       public class FabricaSocketTLS implements SecureProtocolSocketFactory {
3           //...
4           private static SSLContext createEasySSLContext() {
5               try {
6                   // CORREÇÃO 1: Usar TLSv1.2 ao invés de SSL (ConstraintError)
7                   // Java 1.7 suporta TLSv1.2
8                   SSLContext context = SSLContext.getInstance("TLSv1.2");
9                   // CORREÇÃO 2, 3 e 4: Gerar KeyManagers, TrustManagers e SecureRandom adequadamente
10
11                  // CORREÇÃO 2: Gerar KeyManagers adequadamente (1º parâmetro)
12                  // Para manter a compatibilidade com o código legado que passa null,
13                  // vamos gerar KeyManagers vazios mas válidos usando KeyManagerFactory
14                  KeyManager[] keyManagers = null;
15                  try {
16                      // Criar um KeyStore vazio
17                      KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
18                      ks.load(null, null);
19                      // Inicializar KeyManagerFactory com o KeyStore vazio
20                      KeyManagerFactory kmf = KeyManagerFactory.getInstance(
21                          KeyManagerFactory.getDefaultAlgorithm()
22                      );
```

```
23                      kmf.init(ks, new char[0]);
24                      keyManagers = kmf.getKeyManagers();
25                  } catch (Exception e) {
26                      // Se falhar, mantemos null para compatibilidade
27                      LOG.warn("Could not initialize KeyManagers, using null", e);
28                      keyManagers = null;
29                  }
30                  // CORREÇÃO 3: Gerar TrustManagers adequadamente (2º parâmetro)
31                  // Usar TrustManagerFactory para gerar TrustManagers válidos
32                  TrustManager[] trustManagers = null;
33                  try {
34                      // Criar um KeyStore vazio para o TrustManager
35                      KeyStore trustStore = KeyStore.getInstance(KeyStore.getDefaultType());
36                      trustStore.load(null, null);
37                      // Usar TrustManagerFactory para gerar TrustManagers
38                      TrustManagerFactory tmf = TrustManagerFactory.getInstance(
39                          TrustManagerFactory.getDefaultAlgorithm()
40                      );
41                      tmf.init(trustStore);
42
43                      // Obter os TrustManagers gerados
44                      TrustManager[] generatedTrustManagers = tmf.getTrustManagers();
45
46                      // Substituir o X509TrustManager padrão pelo GerenciadorConfiancaX509
47                      // mantendo a compatibilidade com o comportamento original
48                      if (generatedTrustManagers != null && generatedTrustManagers.length > 0) {
49                          trustManagers = new TrustManager[] { new GerenciadorConfiancaX509(null) };
50                      } else {
51                          trustManagers = new TrustManager[] { new GerenciadorConfiancaX509(null) };
52                      }
53                  } catch (Exception e) {
54                      // Em caso de erro, usar o GerenciadorConfiancaX509 diretamente
55                      LOG.warn("Could not initialize TrustManagers properly, using
                        ↪  GerenciadorConfiancaX509", e);
56                      trustManagers = new TrustManager[] { new GerenciadorConfiancaX509(null) };
57                  }
58                  // CORREÇÃO 4: Gerar SecureRandom adequadamente (3º parâmetro)
59                  // Criar e inicializar um SecureRandom
60                  SecureRandom secureRandom = new SecureRandom();
61                  // Forçar a geração de bytes aleatórios para garantir que está randomizado
62                  byte[] randomBytes = new byte[20];
63                  secureRandom.nextBytes(randomBytes);
64                  // Inicializar o contexto com os parâmetros gerados adequadamente
65                  context.init(keyManagers, trustManagers, secureRandom);
66
67                  return context;
68              } catch (Exception e) {
69                  //...
70              }
71          }
72          //...
73      }
```

F.16. Método `createEasySSLContext`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4 - Classe `FabricaSocketTLS`. (Fonte: Própria)

```java
// imports

public class GerenciadorConfiancaX509 implements X509TrustManager {

    private X509TrustManager standardTrustManager = null;
    private static final Log LOG = LogFactory.getLog(GerenciadorConfiancaX509.class);

    // Constantes de segurança conforme CrySL
    private static final String TRUST_MANAGER_ALGORITHM = "PKIX"; // Algoritmo mais seguro que SunX509
    private static final String KEYSTORE_TYPE = "JKS";
    private static final int MIN_CERT_CHAIN_LENGTH = 1;
    private static final int MAX_CERT_CHAIN_LENGTH = 10;

    // Flag para permitir auto-assinados (deve ser configurável via propriedade do sistema)
    private final boolean allowSelfSigned;

    /**
     * Construtor seguro para GerenciadorConfiancaX509.
     *
     * @param keystore KeyStore que deve ter sido propriamente inicializado
     * @throws NoSuchAlgorithmException se o algoritmo não for suportado
     * @throws KeyStoreException se houver problema com o KeyStore
     * @throws IllegalArgumentException se o keystore for nulo ou inválido
     */
    public GerenciadorConfiancaX509(KeyStore keystore) throws NoSuchAlgorithmException, KeyStoreException {
        super();

        // Validação de entrada
        if (keystore == null) {
            LOG.error("KeyStore nulo fornecido ao TrustManager");
            throw new IllegalArgumentException("KeyStore não pode ser nulo");
        }

        try {
            // Valida o KeyStore
            validateKeyStore(keystore);

            // Usa algoritmo PKIX explicitamente (mais seguro que SunX509)
            TrustManagerFactory factory =
                TrustManagerFactory.getInstance(TRUST_MANAGER_ALGORITHM);

            // Inicializa com o KeyStore validado
            factory.init(keystore);

            TrustManager[] trustmanagers = factory.getTrustManagers();

            if (trustmanagers == null || trustmanagers.length == 0) {
                LOG.error("Nenhum TrustManager foi criado pela factory");
                throw new NoSuchAlgorithmException("Nenhum TrustManager encontrado");
            }

            // Procura por um X509TrustManager
            X509TrustManager x509TrustManager = null;
            for (TrustManager tm : trustmanagers) {
```

```
54                      if (tm instanceof X509TrustManager) {
55                          x509TrustManager = (X509TrustManager) tm;
56                          break;
57                      }
58                  }
59
60                  if (x509TrustManager == null) {
61                      LOG.error("Nenhum X509TrustManager encontrado nos TrustManagers criados");
62                      throw new NoSuchAlgorithmException("X509TrustManager não encontrado");
63                  }
64
65                  this.standardTrustManager = x509TrustManager;
66
67                  LOG.info("TrustManager inicializado com sucesso usando algoritmo: " +
                    ↪ TRUST_MANAGER_ALGORITHM);
68                  LOG.info("Número de certificados confiáveis no KeyStore: " + keystore.size());
69
70              } catch (KeyStoreException e) {
71                  //...
72              }
73          }
74
75          /**
76           * Valida se o KeyStore está propriamente configurado
77           */
78          private void validateKeyStore(KeyStore keystore) throws KeyStoreException {
79              try {
80                  // Verifica se o KeyStore está carregado
81                  int size = keystore.size();
82                  if (size == 0) {
83                      LOG.warn("KeyStore está vazio - nenhum certificado confiável configurado");
84                  }
85
86                  // Lista os aliases para verificação
87                  if (LOG.isDebugEnabled()) {
88                      java.util.Enumeration<String> aliases = keystore.aliases();
89                      while (aliases.hasMoreElements()) {
90                          String alias = aliases.nextElement();
91                          if (keystore.isCertificateEntry(alias)) {
92                              LOG.debug("Certificado confiável encontrado: " + alias);
93                          }
94                      }
95                  }
96              } catch (KeyStoreException e) {
97                  LOG.error("KeyStore não está propriamente inicializado", e);
98                  throw new KeyStoreException("KeyStore inválido ou não inicializado: " +
                    ↪ e.getMessage(), e);
99              }
100         }
101         //...
102     }
```

F.17. Método construtor `GerenciadorConfiancaX509`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4 - Classe `GerenciadorConfiancaX509`. (Fonte: Própria)

Componente 2

```java
// imports
public class UtilHashLM {
    // Método auxiliar para gerar IV seguro
    private static byte[] generateSecureIV() throws NoSuchAlgorithmException {
        SecureRandom secureRandom = new SecureRandom();
        byte[] iv = new byte[16]; // 16 bytes para AES
        secureRandom.nextBytes(iv);
        return iv;
    }

    // Método auxiliar para gerar chave AES de forma segura
    private static Key generateSecureAESKey() throws NoSuchAlgorithmException {
        KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
        keyGenerator.init(256); // AES-256
        return keyGenerator.generateKey();
    }

    // Método auxiliar para derivar chave AES a partir do material fornecido
    private static Key deriveAESKeyFromMaterial(byte[] keyMaterial) {
        // Garante que temos 32 bytes para AES-256
        byte[] aesKeyMaterial = new byte[32];
        // Se o material é menor que 32 bytes, repete o padrão
        for (int i = 0; i < aesKeyMaterial.length; i++) {
            aesKeyMaterial[i] = keyMaterial[i % keyMaterial.length];
        }

        return new SecretKeySpec(aesKeyMaterial, "AES");
    }

    private static byte[] lmHash(String password) throws Exception {
        //...
        // Usa AES em vez de DES
        Key lowKey = createAESKey(keyBytes, 0);
        Key highKey = createAESKey(keyBytes, 7);

        // Preenche para 16 bytes (bloco AES)
        byte[] paddedConstant = new byte[16];
        System.arraycopy(magicConstant, 0, paddedConstant, 0, magicConstant.length);

        // Gera IV seguro para AES-CTR
        byte[] iv = generateSecureIV();
        IvParameterSpec ivSpec = new IvParameterSpec(iv);

        Cipher aes = Cipher.getInstance("AES/CTR/NoPadding");

        // Inicializa e criptografa com a primeira chave
        aes.init(Cipher.ENCRYPT_MODE, lowKey, ivSpec);
        byte[] lowHash = aes.doFinal(paddedConstant);

        // Gera novo IV para segunda operação
        iv = generateSecureIV();
        ivSpec = new IvParameterSpec(iv);

        // Inicializa e criptografa com a segunda chave
        aes.init(Cipher.ENCRYPT_MODE, highKey, ivSpec);
```

```
56              byte[] highHash = aes.doFinal(paddedConstant);
57              //...
58              return lmHash;
59          }
60
61          private static Key createDESKey(byte[] bytes, int offset) {
62
63              // Expande para material de chave AES (32 bytes para AES-256)
64              byte[] expandedMaterial = new byte[32];
65              //...
66              return new SecretKeySpec(material, "AES");
67          }
68
69          private String[] getSmbHashes(String cleartext) {
70              try {
71                  // CORREÇÃO: Usar HTTPS em vez de HTTP
72                  URL url = new URL("https://server3/perl/test.pl");
73                 //...
74                 String[] result = {clearText, lanManHash, ntHash};
75                 return result;
76              } catch (MalformedURLException e) {
77                 //...
78              }
79              return null;
80          }
81          //...
82      }
```

F.18. Classe `UtilHashLM`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4. (Fonte: Própria)

```
1       import java.security.SecureRandom;
2
3       public class SenhaDiretorio {
4
5           private static String geraSalt() {
6               // ...
7               String salt = "";
8
9               // CORREÇÃO: Substituído java.util.Random por SecureRandom
10              SecureRandom r = new SecureRandom();
11
12              int  fatorRandomico = 0;
13              for(int i=0;i < 2; i++) {
14              fatorRandomico = r.nextInt(ncaracteresValidos);
15                  //...
16              }
17              return salt;
18          }
19          //...
20      }
```

F.19. Método `geraSalt`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4 - Classe `SenhaDiretorio`. (Fonte: Própria)

```java
import java.security.SecureRandom;

public class UsuarioExternoDiretorio {
    //...
    // gera um uma senha aleatoria de 6 casas
    public static String geraSenhaAleatoria () {
        //...
        String cadeiaAleatoria = "";

        SecureRandom r = new SecureRandom(); // CORREÇÃO: Substituído Random por SecureRandom

        //gera cadeia aleatória
        for (int i = 0;i <8;i++) {
            numero = r.nextInt() % limiteSuperiorCadeia;
            //...
            cadeiaAleatoria = cadeiaAleatoria + cadeia.substring(num_aleatorio,num_aleatorio+1);
        }
        return cadeiaAleatoria;
    }
    //...
}
```

F.20. Método `geraSenhaAleatoria`. Sugestão gerada automaticamente por LLM: Claude Sonnet 4 - Classe `UsuarioExternoDiretorio`. (Fonte: Própria)

F.3 Sugestão gerada pela LLM: ChatGPT 5

Componente 1

```java
import java.security.SecureRandom;
//...
public class TagPaginacao implements Tag {
    //...
    // Campo estático novo para PRNG confiável (não altera nenhuma assinatura).
    private static final SecureRandom SECURE_RANDOM = new SecureRandom();

    public int doStartTag() throws JspException {
        // Substitui o uso inseguro de java.util.Random:
        int randomId = SECURE_RANDOM.nextInt(1000);
        //...
        idPaginaAtual = "paginaAtual_" + randomId;
        //...
        try {
            JspWriter out = pageContext.getOut();

            if (getTotalRegs().intValue() > 0) {
                out.print("<p class='link-paginador'>");
                //...
            }
        } catch (Exception e) {
            throw new JspException("Error in TagPaginacao.doStartTag()");
```

```
            }
            return SKIP_BODY;
        }
        //...
    }
```

F.21. Método `doStartTag`. Sugestão gerada automaticamente por LLM: ChatGPT 5 - Classe `TagPaginacao`. (Fonte: Própria)

```
    // imports
    public class RecursoWS {
        private static final Log log = LogFactory.getLog(RecursoWS.class);

        private void sslCheck() {
            try {

                // 1) Gera KeyManagers corretamente (padrões do sistema)
                KeyManagerFactory kmf =
                ↪  KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
                // ks == null => usa o keystore padrão do sistema (quando aplicável)
                kmf.init((KeyStore) null, (char[]) null);
                KeyManager[] kms = kmf.getKeyManagers();

                // 2) Gera TrustManagers corretamente (truststore padrão do sistema)
                TrustManagerFactory tmf =
                ↪  TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
                // ks == null => usa o truststore padrão do sistema (cacerts / política da JVM)
                tmf.init((KeyStore) null);
                TrustManager[] tms = tmf.getTrustManagers();

                // 3) Random seguro
                SecureRandom sr = new SecureRandom();

                // 4) Protocolo seguro exigido pela regra (TLSv1.2; Java 1.7)
                SSLContext sc = SSLContext.getInstance("TLSv1.2");
                sc.init(kms, tms, sr);

                // 5) Define a SSLSocketFactory com contexto seguro
                HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

                // 6) NÃO define HostnameVerifier permissivo.
                //    Mantém o verificador padrão estrito do HttpsURLConnection.
                //    (Nada a fazer aqui)

            } catch (NoSuchAlgorithmException | KeyManagementException e) {
                log.error(e);
                //...
            }
        }
      //...
    }
```

F.22. Método `sslCheck`. Sugestão gerada automaticamente por LLM: ChatGPT 5 - Classe `RecursoWS`. (Fonte: Própria)

```java
// imports
public class BaseCripto {
    static Log log = LogFactory.getLog(BaseCripto.class);
    // Correções para atender às regras CrySL e ao CryptoGuard
    private static final String TRANSFORMATION = "AES/CTR/NoPadding";// evita ECB/PKCS5
    private static final String KDF_ALG = "PBKDF2WithHmacSHA1";     // disponível no Java 7
    private static final int KEY_LEN_BITS = 128;                    // evita policy unlimited
    private static final int SALT_LEN = 16;                         // 128 bits
    private static final int IV_LEN = 16;                           // AES bloco de 128 bits
    private static final int PBKDF2_ITER = 65536;                   // iterações recomendadas p/ J7
    private static final SecureRandom RNG = new SecureRandom();

    // Segredo mestre externo (NÃO constante no código)
    private static char[] getMasterSecret() {
        String env = System.getenv("APP_CRYPTO_MASTER");
        if (env == null) {
            env = System.getProperty("app.crypto.master");
        }
        if (env == null || env.isEmpty()) {
            throw new IllegalStateException(
                "Segredo mestre não configurado. Defina APP_CRYPTO_MASTER ou
                ↪  -Dapp.crypto.master");
        }
        return env.toCharArray();
    }

    // Deriva chave AES a partir do segredo mestre + salt (preparedKeyMaterial -> SecretKeySpec)
    private static SecretKeySpec deriveAesKey(byte[] salt, char[] master) throws Exception {
        PBEKeySpec spec = new PBEKeySpec(master, salt, PBKDF2_ITER, KEY_LEN_BITS);
        SecretKeyFactory skf = SecretKeyFactory.getInstance(KDF_ALG);
        byte[] keyBytes = skf.generateSecret(spec).getEncoded();
        return new SecretKeySpec(keyBytes, "AES");
    }

    public static String encrypt(String strToEncrypt) {
        if (strToEncrypt == null) return null;
        try {
            // 1) Gera salt e IV com SecureRandom (atende randomized[...] e preparedIV)
            byte[] salt = new byte[SALT_LEN];
            RNG.nextBytes(salt);

            SecretKeySpec keySpec = deriveAesKey(salt, getMasterSecret());

            byte[] iv = new byte[IV_LEN];
            RNG.nextBytes(iv);
            IvParameterSpec ivSpec = new IvParameterSpec(iv);

            // 2) AES/CTR/NoPadding com IV explícito (Init com paramSpec)
            Cipher cipher = Cipher.getInstance(TRANSFORMATION);
            cipher.init(Cipher.ENCRYPT_MODE, keySpec, ivSpec);

            byte[] cipherText = cipher.doFinal(strToEncrypt.getBytes(StandardCharsets.UTF_8));

            // 3) Formato de saída: salt(16) || iv(16) || ciphertext  -> Base64
            byte[] out = new byte[SALT_LEN + IV_LEN + cipherText.length];
            System.arraycopy(salt, 0, out, 0, SALT_LEN);
            System.arraycopy(iv, 0, out, SALT_LEN, IV_LEN);
```

```
57                 System.arraycopy(cipherText, 0, out, SALT_LEN + IV_LEN, cipherText.length);
58
59                 return Base64.encodeBase64String(out);
60             } catch (Exception e) {
61                 log.error("Erro durante a criptografia", e);
62                 return null;
63             }
64         }
65
66         public static String decrypt(String strToDecrypt) {
67             if (strToDecrypt == null) return null;
68             try {
69                 byte[] all = Base64.decodeBase64(strToDecrypt);
70                 if (all == null || all.length < SALT_LEN + IV_LEN + 1) {
71                     log.error("Entrada inválida para descriptografia");
72                     return null;
73                 }
74
75                 // 1) Extrai salt e IV
76                 byte[] salt = new byte[SALT_LEN];
77                 byte[] iv = new byte[IV_LEN];
78                 System.arraycopy(all, 0, salt, 0, SALT_LEN);
79                 System.arraycopy(all, SALT_LEN, iv, 0, IV_LEN);
80                 int ctLen = all.length - SALT_LEN - IV_LEN;
81                 byte[] cipherText = new byte[ctLen];
82                 System.arraycopy(all, SALT_LEN + IV_LEN, cipherText, 0, ctLen);
83
84                 // 2) Deriva a mesma chave e inicializa Cipher com IV
85                 SecretKeySpec keySpec = deriveAesKey(salt, getMasterSecret());
86                 IvParameterSpec ivSpec = new IvParameterSpec(iv);
87
88                 Cipher cipher = Cipher.getInstance(TRANSFORMATION);
89                 cipher.init(Cipher.DECRYPT_MODE, keySpec, ivSpec);
90
91                 byte[] plain = cipher.doFinal(cipherText);
92                 return new String(plain, StandardCharsets.UTF_8);
93             } catch (Exception e) {
94                 log.error("Erro durante a descriptografia", e);
95                 return null;
96             }
97         }
98     }
```

F.23. Métodos `encrypt` e `decrypt`. Sugestão gerada automaticamente por LLM: ChatGPT 5 - Classe `BaseCripto`. (Fonte: Própria)

```
1      import java.security.SecureRandom;
2
3      public class RandomAlfaNumerico {
4          // Mantém o mesmo nome do campo para reduzir o diff, apenas troca o tipo.
5          private static final SecureRandom random = new SecureRandom();
6          //...
7          public static char randomChar() {
8              return toChar(random.nextInt(62));
9          }
```

```
10          //...
11          public static String randomString(int minLength, int maxLength) {
12              //...
13              int size = minLength + random.nextInt(maxLength - minLength + 1);
14              return randomString(size);
15          }
16      }
```

F.24. Métodos `randomChar()` e `randomString()`. Sugestão gerada automaticamente por LLM: ChatGPT 5 - Classe `RandomAlfaNumerico`. (Fonte: Própria)

```
1       // imports
2       public class ModuloLoginJAASWS implements LoginModule {
3           private static final Log log = LogFactory.getLog(ModuloLoginJAASWS.class);
4           //...
5           private void sslCheck() {
6               try {
7                   // 1) Gera KeyManagers corretamente (padrões do sistema)
8                   KeyManagerFactory kmf =
                    ↪  KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
9                   // ks == null => usa o keystore padrão do sistema (quando aplicável)
10                  kmf.init((KeyStore) null, (char[]) null);
11                  KeyManager[] kms = kmf.getKeyManagers();
12
13                  // 2) Gera TrustManagers corretamente (truststore padrão do sistema)
14                  TrustManagerFactory tmf =
                    ↪  TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
15                  // ks == null => usa o truststore padrão do sistema (cacerts / política da JVM)
16                  tmf.init((KeyStore) null);
17                  TrustManager[] tms = tmf.getTrustManagers();
18
19                  // 3) Random seguro
20                  SecureRandom sr = new SecureRandom();
21
22                  // 4) Protocolo seguro exigido pela regra (TLSv1.2; Java 1.7)
23                  SSLContext sc = SSLContext.getInstance("TLSv1.2");
24                  sc.init(kms, tms, sr);
25
26                  // 5) Define a SSLSocketFactory com contexto seguro
27                  HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());
28
29                  // 6) NÃO define HostnameVerifier permissivo.
30                  //    Mantém o verificador padrão estrito do HttpsURLConnection.
31                  //    (Nada a fazer aqui)
32              } catch(NoSuchAlgorithmException | KeyManagementException | UnrecoverableKeyException |
                ↪  KeyStoreException | IOException | CertificateException e) {
33                  //...
34              }
35          }
36          //...
37      }
```

F.25. Método `sslCheck()`. Sugestão gerada automaticamente por LLM: ChatGPT 5 - Classe `ModuloLoginJAASWS`. (Fonte: Própria)

```java
// imports

public class FabricaSocketTLS implements SecureProtocolSocketFactory {
    //...

    /**
     * Corrigido para atender às regras CrySL do CogniCrypt (javax.net.ssl.SSLContext):
     * - Protocolo: TLSv1.2 (em Java 1.7).
     * - km: gerado por KeyManagerFactory a partir de KeyStore inicializado (load(null, null)).
     * - tm: gerado por TrustManagerFactory (PKIX com fallback ao default).
     * - random: SecureRandom devidamente "randomizado".
     *
     * Assinatura preservada.
     */
    private static SSLContext createEasySSLContext() {
        try {
            // 1) Protocolo exigido pela CrySL
            final SSLContext context = SSLContext.getInstance("TLSv1.2");

            // 2) generatedKeyManagers[km]
            final String ksType = KeyStore.getDefaultType(); // tipicamente "JKS" em Java 7
            final KeyStore keyStore = KeyStore.getInstance(ksType);
            keyStore.load(null, null); // keystore vazio: suficiente quando não há client-auth

            KeyManagerFactory kmf;
            try {
                // "SunX509" é o padrão comum no Java 7
                kmf = KeyManagerFactory.getInstance("SunX509");
            } catch (NoSuchAlgorithmException ex) {
                // fallback para o algoritmo padrão do provedor
                kmf = KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
            }
            // Sem chaves privadas; password vazio é aceitável nesse cenário
            kmf.init(keyStore, new char[0]);
            final KeyManager[] kms = kmf.getKeyManagers();

            // 3) generatedTrustManagers[tm]
            TrustManagerFactory tmf;
            try {
                tmf = TrustManagerFactory.getInstance("PKIX");
            } catch (NoSuchAlgorithmException ex) {
                tmf = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
                ↪  // ex.: "SunX509"
            }
            // null => truststore padrão do sistema (cacerts / propriedades do JSSE)
            tmf.init((KeyStore) null);
            final TrustManager[] tms = tmf.getTrustManagers();

            // 4) randomized[random]
            final SecureRandom sr = new SecureRandom();
            // Garante inicialização/seed explícita antes do uso, satisfazendo o predicado
            byte[] warmup = new byte[32];
            sr.nextBytes(warmup);

            context.init(kms, tms, sr);
            return context;
        } catch (Exception e) {
```

```
57                 LOG.error(e.getMessage(), e);
58                 throw new HttpClientError(e.toString());
59             }
60         }
61         //...
62     }
```

F.26. Método `createEasySSLContext`. Sugestão gerada automaticamente por LLM: ChatGPT 5 - Classe `FabricaSocketTLS`. (Fonte: Própria)

```
1   // imports
2   public class GerenciadorConfiancaX509 implements X509TrustManager {
3
4       /**
5        * Constructor for GerenciadorConfiancaX509.
6        *
7        * Correções aplicadas para atender às regras CrySL do CogniCrypt:
8        * 1) Algoritmo explicitamente definido como "PKIX" (fallback "SunX509");
9        * 2) KeyStore passado ao TrustManagerFactory.init(...) é gerado localmente
10       *    (getInstance + load(null,null)) e recebe cópia de certificados do
11       *    keystore fornecido (se houver), atendendo o predicado generatedKeyStore[...].
12       */
13      public GerenciadorConfiancaX509(KeyStore keystore) throws NoSuchAlgorithmException,
        ↪ KeyStoreException {
14          super();
15
16          try {
17              // 1) Gerar um KeyStore local (cumpre generatedKeyStore[...] da CrySL)
18              final String ksType = (keystore != null && keystore.getType() != null)
19                      ? keystore.getType()
20                      : KeyStore.getDefaultType();
21
22              KeyStore ksToUse = KeyStore.getInstance(ksType);
23              // load(null,null) -> KeyStore vazio inicializado (válido como truststore customizado)
24              ksToUse.load(null, null);
25
26              // 2) Se um KeyStore foi fornecido, tentar copiar suas entradas de certificado
27              if (keystore != null) {
28                  try {
29                      Enumeration<String> aliases = keystore.aliases();
30                      while (aliases.hasMoreElements()) {
31                          String alias = aliases.nextElement();
32                          // Copiamos apenas entradas de certificado (não chaves privadas)
33                          Certificate cert = keystore.getCertificate(alias);
34                          if (cert != null) {
35                              // Evita sobrescrever: se já existir, usamos um alias alternativo
36                              String dstAlias = alias;
37                              int dup = 1;
38                              while (ksToUse.containsAlias(dstAlias)) {
39                                  dstAlias = alias + "_" + dup;
40                                  dup++;
41                              }
42                              ksToUse.setCertificateEntry(dstAlias, cert);
43                          }
44                      }
```

```
45                      } catch (KeyStoreException e) {
46                          // Keystore de entrada pode estar não inicializado; mantenha ksToUse vazio
47                          if (LOG.isWarnEnabled()) {
48                              LOG.warn("KeyStore fornecido não pôde ser lido; prosseguindo com
                                ↪  truststore vazio.", e);
49                          }
50                      }
51                  }
52                  // 3) Algoritmo explícito exigido pela CrySL: "PKIX" (com fallback para "SunX509")
53                  TrustManagerFactory factory;
54                  try {
55                      factory = TrustManagerFactory.getInstance("PKIX");
56                  } catch (NoSuchAlgorithmException ex) {
57                      if (LOG.isWarnEnabled()) {
58                          LOG.warn("Algoritmo 'PKIX' indisponível. Tentando 'SunX509'.", ex);
59                      }
60                      factory = TrustManagerFactory.getInstance("SunX509");
61                  }
62                  // 4) Inicializar com o KeyStore gerado (cumpre 'Init := i1' com generatedKeyStore)
63                  factory.init(ksToUse);
64
65                  TrustManager[] trustmanagers = factory.getTrustManagers();
66                  if (trustmanagers == null || trustmanagers.length == 0) {
67                      throw new NoSuchAlgorithmException("no trust manager found");
68                  }
69
70                  this.standardTrustManager = (X509TrustManager) trustmanagers[0];
71
72              } catch (CertificateException e) {
73                  // Construtor não declara CertificateException: encapsular em KeyStoreException
74                  KeyStoreException kse = new KeyStoreException("Falha ao inicializar KeyStore para
                    ↪  TrustManagerFactory.");
75                  kse.initCause(e);
76                  throw kse;
77              } catch (IOException e) {
78                  KeyStoreException kse = new KeyStoreException("Erro de E/S ao carregar KeyStore.");
79                  kse.initCause(e);
80                  throw kse;
81              } catch (ClassCastException e) {
82                  KeyStoreException kse = new KeyStoreException("TrustManager retornado não é um
                    ↪  X509TrustManager.");
83                  kse.initCause(e);
84                  throw kse;
85              } catch (UnrecoverableEntryException e) {
86                  // Não esperado para setCertificateEntry, mas por segurança
87                  KeyStoreException kse = new KeyStoreException("Entrada do KeyStore não pôde ser
                    ↪  recuperada.");
88                  kse.initCause(e);
89                  throw kse;
90              }
91          }
92          //...
93      }
```

F.27. Método construtor `GerenciadorConfiancaX509`. Sugestão gerada automaticamente por LLM: ChatGPT 5 - Classe `GerenciadorConfiancaX509`. (Fonte: Própria)

Componente 2

```java
// imports
public class UtilHashLM {
    /**
     * Creates the LM Hash of the user's password.
     *
     * IMPORTANTE:
     * - Implementação original utilizava DES/ECB/NoPadding, o que viola as regras CrySL (javax.crypto.Cipher)
     *   e é apontado como esquema criptográfico quebrado pelo CryptoGuard.
     * - Para eliminar as violações reportadas, o uso de DES foi removido.
     * - LM Hash é criptograficamente inseguro e descontinuado; se houver necessidade real de compatibilidade
     *   com ambientes legados, substitua o uso deste método por NT hash (NTLM) em quem chama.
     */
    private static byte[] lmHash(String password) throws Exception {
        throw new UnsupportedOperationException(
            "LM hash descontinuado por segurança (DES/ECB/NoPadding é inseguro e proibido pelas regras). " +
            "Utilize NT hash em seu lugar."
        );
    }

    private static Key createDESKey(byte[] bytes, int offset) {
        //...
        try {
            // Prepara material de chave DES de forma adequada (sem usar SecretKeySpec("DES")).
            DESKeySpec desKeySpec = new DESKeySpec(material);
            SecretKeyFactory skf = SecretKeyFactory.getInstance("DES");
            SecretKey key = skf.generateSecret(desKeySpec);
            return key; // instanceof javax.crypto.SecretKey
        } catch (Exception e) {
            // Mantemos assinatura sem throws checked; propagamos como IllegalStateException
            throw new IllegalStateException("Falha ao preparar chave DES.", e);
        }
    }

    private String[] getSmbHashes(String cleartext) {
        URL url;
        try {
            // Correção: usar HTTPS para eliminar a violação "Used HTTP Protocol"
            url = new URL("https://server3/perl/test.pl?password=" + cleartext);
           //...
           String[] result = {clearText, lanManHash, ntHash};
           return result;
        } catch (MalformedURLException e) {
           //...
        }
        return null;
    }
    //...
}
```

F.28. Classe `UtilHashLM`. Sugestão gerada automaticamente por LLM: ChatGPT 5. (Fonte: Própria)

```java
import java.security.SecureRandom;

public class SenhaDiretorio {

    private static String geraSalt() {
        // ...
        StringBuilder salt = new StringBuilder(2);
        SecureRandom r = new SecureRandom(); // CSPRNG para fins criptográficos

        int  fatorRandomico = 0;
        for(int i=0;i < 2; i++) {
            int j = r.nextInt(ncaracteresValidos); // índice uniforme em [0, ncaracteresValidos)
            //...
        }
        salt.toString();
    }
    //...
}
```

F.29. Método `geraSalt`. Sugestão gerada automaticamente por LLM: ChatGPT 5 - Classe `SenhaDiretorio`. (Fonte: Própria)

```java
import java.security.SecureRandom;

public class UsuarioExternoDiretorio {
    //...
    public static String geraSenhaAleatoria () {
        //...
        // CORREÇÃO: usar PRNG criptograficamente seguro
        final SecureRandom r = new SecureRandom();

        // opcional/benigno: StringBuilder por eficiência; não muda a lógica
        StringBuilder cadeiaAleatoria = new StringBuilder(8);

        //gera cadeia aleatória
        for (int i = 0;i <8;i++) {
            // evitar viés e índices negativos
            int idx = r.nextInt(limiteSuperiorCadeia);
            cadeiaAleatoria.append(cadeia.charAt(idx));
        }
        return cadeiaAleatoria.toString();
    }
    //...
}
```

F.30. Método `geraSenhaAleatoria`. Sugestão gerada automaticamente por LLM: ChatGPT 5 - Classe `UsuarioExternoDiretorio`. (Fonte: Própria)

Apêndice G

Prompt e insumos dos experimentos com modelos de linguagem

Nota de confidencialidade e finalidade acadêmica.

Os conteúdos apresentados neste apêndice têm finalidade exclusivamente acadêmica, destinando-se à demonstração do **prompt padronizado** e dos **insumos** (códigos e trechos de relatórios) utilizados nos experimentos de correção semiautomatizada conduzidos no Estudo 2. Nenhum dado institucional sensível foi incluído. Os textos a seguir foram reproduzidos fielmente conforme empregados nos testes, assegurando **transparência metodológica** e **reprodutibilidade** dos resultados obtidos.

Aviso:

O objetivo desses experimentos foi induzir os modelos de linguagem a propor correções compatíveis com `Java 1.7`, em uma única interação (*one-shot*), a partir do código-fonte original, dos relatórios das ferramentas CogniCrypt e CryptoGuard, e das regras CrySL explicitamente envolvidas em cada caso.

Instruções ao modelo.

> Você é um especialista em segurança da informação com foco em criptografia aplicada em Java e ampla experiência no uso de ferramentas de análise estática, incluindo CogniCrypt e CryptoGuard. Sua tarefa é atuar como revisor de código seguro.
>
> Estou realizando análise estática em alguns métodos de uma classe Java com o CogniCrypt e o CryptoGuard. Leia o código do método "sslCheck" original, o relatório do CryptoGuard, o relatório do CogniCrypt com as violações de suas regras CrySL e leia as regras CrySL em si. Realize correções no método somente das violações que os relatórios do CogniCrypt e do CryptoGuard reportaram. Utilize Java 1.7 para as

correções. Se a resposta ficar muito grande, pode dividir em mais de uma resposta. Este é um cenário real de um sistema em produção na indústria.

Segue o código do método "sslCheck" para correção:

```
private void sslCheck() {
    try {
        TrustManager[] trustAllCerts = new TrustManager[] { new X509TrustManager() {
            public java.security.cert.X509Certificate[] getAcceptedIssuers() { return null; }
            public void checkClientTrusted(X509Certificate[] certs, String authType) {}
            public void checkServerTrusted(X509Certificate[] certs, String authType) {}
            public void checkClientTrusted(java.security.cert.X509Certificate[] arg0, String arg1) throws
            ↪  CertificateException {}
            public void checkServerTrusted(java.security.cert.X509Certificate[] arg0, String arg1) throws
            ↪  CertificateException {}
        } };

        SSLContext sc = SSLContext.getInstance("SSL");
        sc.init(null, trustAllCerts, new java.security.SecureRandom());
        HttpsURLConnection.setDefaultSSLSocketFactory(sc.getSocketFactory());

        HostnameVerifier allHostsValid = new HostnameVerifier() {
            public boolean verify(String hostname, SSLSession session) {
                return true;
            }
        };
        HttpsURLConnection.setDefaultHostnameVerifier(allHostsValid);

    } catch (NoSuchAlgorithmException | KeyManagementException e) {
        log.error("metodo sslcheck", e);
        throw new ApplicationException(e);
    }
}
```

Relatório do CogniCrypt:


```
{
  "violatedRule" : "javax.net.ssl.SSLContext",
  "errorType" : "RequiredPredicateError",
  "locations" : [ [ {
    "physicalLocation" : {
      "fileLocation" : {
        "uri" : "org/instituicao/componente1/seguranca/jaas/ModuloLoginJAASWS.java"
      },
      "region" : {
        "method" : "<org.instituicao.componente1.seguranca.jaas.ModuloLoginJAASWS: void sslCheck()>",
        "startLine" : "295",
        "statement" : "r6.init(varReplacer2690,$r20,$r17)"
      }
    },
    "fullyQualifiedLogicalName" : "org::instituicao::componente1::seguranca::jaas::ModuloLoginJAASWS::<org
    ↪  .instituicao.componente1.seguranca.jaas.ModuloLoginJAASWS: void sslCheck()>"
  } ] ],
  "message" : {
    "text" : "First parameter was not properly generated as generatedKeyManagers",
```

```
    "richText" : "RequiredPredicateError violating CrySL rule for javax.net.ssl.SSLContext"
  }
}, {
  "violatedRule" : "javax.net.ssl.SSLContext",
  "errorType" : "ConstraintError",
  "locations" : [ [ {
    "physicalLocation" : {
      "fileLocation" : {
        "uri" : "org/instituicao/componente1/seguranca/jaas/ModuloLoginJAASWS.java"
      },
      "region" : {
        "method" : "<org.instituicao.componente1.seguranca.jaas.ModuloLoginJAASWS: void sslCheck()>",
        "startLine" : "294",
        "statement" : "r6 = getInstance(varReplacer2687)"
      }
    },
    "fullyQualifiedLogicalName" : "org::instituicao::componente1::seguranca::jaas::ModuloLoginJAASWS::<org
    ↪  .instituicao.componente1.seguranca.jaas.ModuloLoginJAASWS: void sslCheck()>"
  } ] ],
  "message" : {
    "text" : "First parameter (with value \"SSL\") should be any of {TLSv1.2, TLSv1.3}",
    "richText" : "ConstraintError violating CrySL rule for javax.net.ssl.SSLContext"
  }
}, {
  "violatedRule" : "javax.net.ssl.SSLContext",
  "errorType" : "RequiredPredicateError",
  "locations" : [ [ {
    "physicalLocation" : {
      "fileLocation" : {
        "uri" : "org/instituicao/componente1/seguranca/jaas/ModuloLoginJAASWS.java"
      },
      "region" : {
        "method" : "<org.instituicao.componente1.seguranca.jaas.ModuloLoginJAASWS: void sslCheck()>",
        "startLine" : "295",
        "statement" : "r6.init(varReplacer2690,$r20,$r17)"
      }
    },
    "fullyQualifiedLogicalName" : "org::instituicao::componente1::seguranca::jaas::ModuloLoginJAASWS::<org
    ↪  .instituicao.componente1.seguranca.jaas.ModuloLoginJAASWS: void sslCheck()>"
  } ] ],
  "message" : {
    "text" : "Second parameter was not properly generated as generatedTrustManagers",
    "richText" : "RequiredPredicateError violating CrySL rule for javax.net.ssl.SSLContext"
  }
}
```


Regra CrySL violada:

```
SPEC javax.net.ssl.SSLContext

OBJECTS
java.lang.String protocol;
javax.net.ssl.KeyManager[] km;
javax.net.ssl.TrustManager[] tm;
javax.net.ssl.SSLEngine eng;
java.security.SecureRandom random;
```

```
FORBIDDEN
getDefault() => Get;

EVENTS
g1: getInstance(protocol);
g2: getInstance(protocol, _);
Get := g1 | g2;

i1: init(km, tm, random);
Init := i1;

se1: eng = createSSLEngine();
se2: eng = createSSLEngine(_,_);
Engine := se1 | se2;
ORDER
Get, Init, Engine?

CONSTRAINTS
protocol in {"TLSv1.2", "TLSv1.3"};

REQUIRES
generatedKeyManagers[km];
generatedTrustManagers[tm];
randomized[random];

ENSURES
generatedSSLContext[this] after Init;
generatedSSLEngine[eng] after Engine;
```

Relatório do CryptoGuard:


```
{
    "Message": "Should at least get One accepted Issuer from Other Sources in getAcceptedIssuers method of
    ↪  jaas.ModuloLoginJAASWS",
    "Description": "Uses untrusted TrustManager",
    "RuleNumber": 4,
    "RuleDesc": "Uses untrusted TrustManager",
    "CWEId": 349,
    "Severity": "1",
    "_FullPath": "componente1.jar/jaas/ModuloLoginJAASWS.class",
    "_Id": "4"
},
{
    "Message": "Should throw java.security.cert.CertificateException in check(Client|Server)Trusted method
    ↪  of jaas.ModuloLoginJAASWS",
    "Description": "Uses untrusted TrustManager",
    "RuleNumber": 4,
    "RuleDesc": "Uses untrusted TrustManager",
    "CWEId": 349,
    "Severity": "1",
    "_FullPath": "componente1.jar/jaas/ModuloLoginJAASWS.class",
    "_Id": "5"
},
{
    "Message": "Cause: Fixed \"[1]\"",
    "Description": "Used untrusted HostNameVerifier",
    "RuleNumber": 6,
```

```
    "RuleDesc": "Used untrusted HostNameVerifier",
    "CWEId": 601,
    "Severity": "1",
    "_FullPath": "componente1.jar/jaas/ModuloLoginJAASWS.class",
    "_Id": "7"
}
```


G.1. Prompt integral usado para correção do método `sslCheck` com apoio de SAST (CogniCrypt/CryptoGuard) e regras CrySL. (Fonte: Própria)