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

Análise de uso incorreto de APIs de Criptografia Java em uma Instituição Financeira

Carine da Silva Ferreira

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)

Da Silva Ferreira, Carine
DF383a Análise de uso incorreto de APIs de Criptografia Java em uma Instituição Financeira / Carine Da Silva Ferreira; orientador Rodrigo Bonifácio de Almeida. Brasília, 2025.
48 p.

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

1. Análise Estática (SAST). 2. Consumo de Energia e CO2 (CodeCarbon). 3. Uso indevido de API de Criptografia. 4. CamBench. 5. DevSecOps Verde. I. Bonifácio de Almeida, Rodrigo, orient. II. Título.

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

Análise de uso incorreto de APIs de Criptografia Java em uma Instituição Financeira

Carine da Silva Ferreira

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

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

Prof. Dra. Paola Rodrigues de Godoy Accioly
CIN/UFPE

Dr. Eduardo Adilo Pelinson Alchieri
CIC/UnB

Prof. Dra. Edna Dias Canedo
Coordenador do Programa de Pós-graduação em Computação Aplicada

Brasília, 08 de Dezembro de 2025

Dedicatória

Dedico este trabalho à minha filha e à minha família, em especial à minha mãe, e também aos meus amigos que me apoiaram com convicção, mesmo nos momentos de minha desconfiança em mim, e que estiveram presentes em todos os momentos dessa travessia.

Agradecimentos

A Deus, pela força e pela resiliência que me sustentaram ao longo de toda esta caminhada.

À minha filha, Ana Clara, pela paciência, compreensão e apoio, sem os quais este trabalho não teria sido possível.

Aos meus pais, Joacir e Dalva, pelo incentivo constante e pela sustentação para prosseguir, mesmo quando tudo parecia difícil; caminharam comigo com firmeza e equilíbrio, sempre alinhados ao meu propósito maior.

Ao meu irmão, Diego, à minha cunhada, Adriane, e aos meus sobrinhos, João Pedro e Pedro Henrique, que me fazem acreditar que é possível crescer e prosperar amparada pelo carinho e pelo amor da família.

Ao meu orientador, Prof. Dr. Rodrigo Bonifácio de Almeida, por toda a atenção, compreensão e cuidado comigo e com seus alunos. O compartilhamento de seus conhecimentos, promovendo trocas contínuas de aprendizado, capacitou-nos a nos tornarmos pesquisadores ainda melhores.

À Profa. Dra. Paola Accioly e ao Prof. Dr. Eduardo Alchieri, que compuseram esta trajetória como membros da banca examinadora, pelo tempo dedicado à avaliação deste trabalho; e a todo o corpo docente do PPCA, pelos ensinamentos fundamentais, em especial ao Prof. Dr. Marcelo Ladeira, pelo carinho e compreensão.

Ao meu chefe atual, Eduardo Iwakawa, e aos meus antigos chefes, Leonardo Delogo, Everton Rocha, Marcelo André e Rodrigo Galvão, pelo incentivo e apoio ao projeto de pesquisa; em especial ao especialista e chefe César Dantas, que acreditou (*Believe*) na minha capacidade e me orientou na elaboração da pesquisa.

Às minhas amigas, queridas e quase irmãs, Aline, Daiana, Daniela, Helena, Isabel, Marli, Luciana e Rose, pela empatia e pelo cuidado que somente verdadeiras amizades oferecem.

À minha querida amiga e mentora, Ana Paula, que me ajudou a atravessar angústias e preocupações, desafiando-me a seguir adiante, apesar do medo, e a nunca desistir do meu objetivo.

Às profissionais que cuidam da minha saúde mental e espiritual, minha psicóloga, Mairlene e minha professora de yoga, Luanda, cujas terapias reorganizaram e equilibraram

meu bem-estar; este apoio foi fundamental para a conclusão deste trabalho.

À minha médica neurologista, Dra. Fernanda Ferraz, pelo cuidado atento e integral, que me permitiu manter o rumo saudável desta jornada.

Aos amigos e colegas de trabalho, Richard Barroso e Willian Carnauba, pelos ensinamentos, contribuições, sugestões e melhorias no meu notebook; e, em especial, ao amigo e colega de trabalho Wesley Pereira, que me ensinou e auxiliou em todo o processo inicial de configuração dos experimentos, essenciais para conclusão da pesquisa.

E aos queridos amigos de pesquisa, Luís Amaral e Joilton Almeida, pela parceria acadêmica e pelo suporte, técnico e emocional, indispensáveis à finalização deste trabalho.

Resumo

As vulnerabilidades decorrentes do uso incorreto de APIs de criptografia figuram entre as mais recorrentes segundo a OWASP. Para mitigar esse problema, ferramentas comerciais de análise estática têm sido adotadas de forma cada vez mais ampla pela indústria, enquanto a academia desenvolve soluções especializadas como CogniCrypt e CryptoGuard. Embora existam estudos avaliando essas ferramentas em benchmarks sintéticos ou derivados de projetos open-source, ainda há pouca evidência sobre seu desempenho em cenários industriais reais.

Diante dessa lacuna, este trabalho investiga o uso de ferramentas de análise estática no contexto da segurança de software de uma instituição financeira, com foco na detecção de vulnerabilidades relacionadas ao uso inadequado de APIs de criptografia em aplicações Java. Além de analisar a eficácia dessas ferramentas, o estudo incorpora a perspectiva de TI Verde e metas ESG/ASG, avaliando o custo operacional e energético de integrar verificações de segurança em pipelines DevSecOps.

Para atender a esse objetivo, conduzimos uma avaliação sistemática da acurácia e do desempenho de quatro ferramentas SAST—CogniCrypt, CryptoGuard, FindSecBugs/SecBugs e Horusec—na identificação de usos incorretos de criptografia, bem como de seu consumo energético e emissões de CO_2 quando orquestradas em um pipeline DevSecOps Verde. A investigação combinou duas etapas empíricas: (i) avaliação de acurácia utilizando o benchmark CamBench (subconjunto Cap); e (ii) análise de escalabilidade em 211 artefatos JAR de sistemas reais. As métricas consideradas incluíram precisão, recall, F1, tempo de execução, pico de memória, uso de CPU e estimativas de energia (kWh/Wh) e emissões de CO_2.

Entre os achados, identificamos que as ferramentas se comportaram de modo complementar: no benchmark CamBench, CogniCrypt e Horusec apresentaram melhor equilíbrio entre precisão e cobertura, o CryptoGuard teve desempenho intermediário, e o SecBugs exibiu o menor desempenho. Já nos sistemas reais, o SecBugs reportou o maior número de potenciais problemas, enquanto CogniCrypt e CryptoGuard tiveram comportamento mais conservador; a concordância entre ferramentas para um mesmo caso foi rara, exigindo triagem manual. Quanto ao custo operacional, o CogniCrypt foi o mais rápido e leve;

CryptoGuard e Horusec ficaram na faixa intermediária; e o SecBugs foi o mais oneroso em tempo e memória. No consumo de energia, observou-se que, quanto maior o tempo de execução, maior o consumo, e que a ferramenta escolhida e o porte do sistema também modulam esse custo. Concluímos que a seleção de ferramentas deve considerar o objetivo da instituição (velocidade versus cobertura), adotar limites de tempo e priorizar alertas com consenso para reduzir falsos positivos, favorecendo uma esteira DevSecOps mais eficiente e alinhada à TI Verde.

Palavras-chave: Segurança de Software; Análise Estática (SAST); Uso indevido de API de Criptografia; CamBench; DevSecOps Verde; Consumo de Energia e CO_2 (CodeCarbon)

Abstract

Cryptography-API misuse vulnerabilities rank among the most recurrent issues according to OWASP. To mitigate this problem, commercial static analysis tools have been increasingly adopted across industry, while academia has developed specialized solutions such as CogniCrypt and CryptoGuard. Although prior studies evaluate these tools on synthetic benchmarks or datasets derived from open-source projects, there remains limited evidence of their performance in real industrial settings.

Addressing this gap, this work investigates the use of static analysis tools in the software-security context of a financial institution, focusing on the detection of vulnerabilities related to improper use of cryptography APIs in Java applications. Beyond assessing tool effectiveness, the study incorporates a Green IT and ESG/ASG perspective by evaluating the operational and energy costs of integrating security checks into DevSecOps pipelines.

To this end, we conducted a systematic assessment of the accuracy and performance of four SAST tools—CogniCrypt, CryptoGuard, FindSecBugs/SecBugs, and Horusec—in identifying cryptography misuse, as well as of their energy consumption and CO_2 emissions when orchestrated within a Green DevSecOps pipeline. The investigation combined two empirical stages: (i) an accuracy evaluation using the *CamBench* benchmark (Cap subset); and (ii) a scalability analysis over 211 JAR artifacts from real systems. The metrics considered included precision, recall, F1, execution time, peak memory, CPU usage, and energy (kWh/Wh) and CO_2-emission estimates.

Among the findings, we observed complementary tool behavior: on *CamBench*, CogniCrypt and Horusec achieved a better balance between precision and coverage, CryptoGuard showed intermediate performance, and SecBugs exhibited the lowest performance. In real systems, SecBugs reported the largest number of potential issues, whereas CogniCrypt and CryptoGuard behaved more conservatively; cross-tool agreement on the same case was rare, requiring manual triage. Regarding operational cost, CogniCrypt was the fastest and lightest; CryptoGuard and Horusec fell in the intermediate range; and SecBugs was the most demanding in time and memory. For energy consumption, longer execution times were associated with higher consumption, and both the chosen tool and system size

also modulated this cost. We conclude that tool selection should consider the institution's objective (speed versus coverage), adopt time limits, and prioritize alerts with cross-tool consensus to reduce false positives, fostering a more efficient DevSecOps pipeline aligned with Green IT.

Keywords: Software Security; Static Application Security Testing (SAST); Cryptographic API Misuse; CamBench; Green DevSecOps; Energy and CO_2 Consumption (CodeCarbon)

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

API Application Programming Interface (Interface de Programação de Aplicações).

ASG Ambiental, Social e Governança.

CBC Cipher Block Chaining (Encadeamento de Blocos de Cifra).

CD Continuous Delivery (Entrega Contínua).

CI Continuous Integration (Integração Contínua).

CI/CD Continuous Integration / Continuous Deployment (Integração Contínua / Entrega Contínua).

CLI Command Line Interface (Interface de Linha de Comando).

CSRF Cross-Site Request Forgery (Falsificação de Requisição entre Sites).

CWE Common Weakness Enumeration (Enumeração Comum de Fraquezas).

DevSecOps Development, Security and Operations (Integração de Desenvolvimento, Segurança e Operações).

ESG *Environmental, Social and Governance.*

JavaEE Java Platform, Enterprise Edition (Plataforma Java, Edição Corporativa).

MD5 Message Digest 5 (Função Resumo de Mensagem 5).

OWASP Open Web Application Security Project (Projeto Aberto para Segurança em Aplicações Web).

SAST Static Application Security Testing (Teste Estático de Segurança de Aplicações).

SDL Security Development Lifecycle (Ciclo de Vida do Desenvolvimento de Software Seguro).

SDLC Software Development Lifecycle (Ciclo de Vida do Desenvolvimento de Software).

SSL Secure Sockets Layer (Camada de Soquetes Seguros).

Capítulo 1

Introdução

A segurança da informação bancária possui um histórico de constantes desafios e tem se tornado cada vez mais importante e relevante diante de novas ameaças e ataques. Dado o grande volume de transações que são executadas diariamente, faz-se necessário que a segurança seja essencial em todas as etapas do ciclo de vida do software. Um dos grandes desafios são as mudanças constantes de versões, arquiteturais ou regulatórias, com prazos apertados e entregas com falhas no código-fonte e ausência de testes adequados de segurança nas soluções destinadas a clientes internos e externos. O grande aumento no uso de aplicativos móveis, *internet banking* e processamento de transações requer agilidade e compromisso da instituição financeira com a realização de auditorias, treinamentos de conscientização em segurança para funcionários e a implementação de práticas de desenvolvimento seguro (*Development, Security and Operations (Integração de Desenvolvimento, Segurança e Operações) (DevSecOps))* para mitigar vulnerabilidades de forma proativa.

Além das falhas gerais de software, o uso inadequado de Application Programming Interface (Interface de Programação de Aplicações) (API)s de criptografia representa uma vulnerabilidade crítica em sistemas financeiros. Essas são amplamente utilizadas para proteger dados sensíveis, como informações de transações bancárias e dados pessoais de clientes. No entanto, estudos recentes identificam o uso incorreto como uma das principais causas de vulnerabilidades exploradas por atacantes. Práticas inadequadas, como o uso de algoritmos de criptografia obsoletos (e.g., Message Digest 5 (Função Resumo de Mensagem 5) (MD5)), chaves de criptografia fracas ou a omissão de configurações obrigatórias (e.g., Cipher Block Chaining (Encadeamento de Blocos de Cifra) (CBC) com *padding*), frequentemente resultam na exposição de dados sensíveis. Como destacado por [1], erros como a falta de verificação de condições ou o manuseio incorreto de exceções podem levar a *bugs*, *crashes* e até mesmo brechas de segurança. O trabalho de [19] reforça que muitos desenvolvedores enfrentam dificuldades ao utilizá-las, frequentemente devido à complexidade da documentação e à ausência de exemplos claros. Assim, identificar e

corrigir esses erros durante o desenvolvimento é crucial para garantir a segurança e a confiabilidade dos sistemas financeiros.

Além do vetor tradicional de *segurança*, instituições financeiras têm perseguido metas de *sustentabilidade* alinhadas a estratégias *Environmental, Social and Governance* (ESG)/Ambiental, Social e Governança (ASG). Nesse contexto, a área de TI passa a considerar, de maneira sistemática, o consumo de energia e a pegada de carbono das atividades de desenvolvimento e operação de software. Práticas de *TI Verde* conectam esses objetivos corporativos com o ciclo de vida do software, promovendo medições transparentes e decisões técnicas que reduzam custo ambiental sem comprometer a proteção de dados nem os requisitos regulatórios do setor bancário.

No âmbito desta dissertação, a abordagem DevSecOps é estendida com um enfoque verde: as verificações de segurança por Static Application Security Testing (Teste Estático de Segurança de Aplicações) (SAST) são integradas à esteira CI/CD juntamente a um medidor de energia e emissões (p. ex., *CodeCarbon*), possibilitando observar, por projeto e por ferramenta, o custo operacional associado às análises estáticas. Essa integração, aqui referida como *Pipeline DevSecOps Verde*, mantém o foco em segurança (detecção de uso inadequado de APIs de criptografia) e adiciona visibilidade a *tempo, memória, CPU, energia e* CO_2e como insumos para decisões técnicas e de conformidade ESG.

No entanto, apesar desses esforços, a complexidade e a constante evolução das ameaças cibernéticas exigem uma abordagem proativa e dinâmica. A dependência crescente de APIs de criptografia em sistemas bancários introduz novas vulnerabilidades que podem ser exploradas por agentes maliciosos. Essas implementações inadequadas podem comprometer a segurança dos dados e também ampliar a superfície de ataque, criando oportunidades para exploração por agentes mal-intencionados. Portanto, as instituições financeiras implementam diversas soluções de segurança, realizam análises contínuas de suas infraestruturas tecnológicas e de softwares para identificar e corrigir possíveis brechas de vulnerabilidade.

A análise do uso inadequado de APIs de criptografia fortalece a segurança no setor bancário, prevenindo vulnerabilidades e protegendo a integridade dos dados financeiros. Corrigir implementações inseguras contribui diretamente para a mitigação de riscos e intensifica a resiliência das instituições financeiras contra ataques cibernéticos, reforçando a confiança no sistema bancário.

1.1 Caracterização do Problema

A instituição financeira objeto desta pesquisa é uma das maiores do país. Historicamente, seus sistemas bancários foram desenvolvidos e operam em plataformas *mainframe*, reconhecidas por sua robustez, capacidade de processamento e segurança. Esses sistemas centrais

são responsáveis por processar um volume massivo de transações financeiras, garantindo a integridade e a disponibilidade dos serviços oferecidos aos clientes.

Com o avanço tecnológico e a crescente demanda por serviços digitais, a instituição expandiu sua atuação para além dos sistemas *mainframe* tradicionais. Atualmente, oferece aplicativos móveis, integrações com sistemas externos e disponibiliza APIs para consumo de informações internas. Essa modernização visa atender às expectativas dos clientes por conveniência e acessibilidade, permitindo que realizem transações bancárias a qualquer hora e lugar. No entanto, essa expansão tecnológica introduz novos desafios de segurança, especialmente no que diz respeito à proteção de dados sensíveis e à prevenção de acessos não autorizados.

A crescente complexidade dos sistemas e a integração com múltiplas plataformas aumentam a superfície de ataque, tornando a instituição mais suscetível a vulnerabilidades de software. Estudos indicam que aplicativos bancários podem apresentar deficiências de segurança, como a falta de uso de mecanismos de proteção disponíveis para aplicativos móveis e plataformas bancárias. Além disso, a disponibilização de APIs expõe pontos de entrada que, se não devidamente protegidos, podem ser explorados por agentes mal-intencionados.

Para enfrentar esses desafios, a instituição implementa um conjunto de controles de segurança, incluindo políticas de conformidade, auditorias regulares e monitoramento contínuo de suas infraestruturas. Essas medidas visam identificar e mitigar riscos associados ao desenvolvimento e à operação de seus sistemas. No entanto, a eficácia desses controles depende da capacidade de detectar e corrigir vulnerabilidades de forma proativa, antes que possam ser exploradas.

Uma abordagem adotada pela instituição é a integração de ferramentas de análise estática de código em sua esteira de desenvolvimento. Ferramentas como Checkmarx e SonarQube são utilizadas para inspecionar o código-fonte em busca de vulnerabilidades conhecidas, permitindo que os desenvolvedores corrijam problemas de segurança durante as fases iniciais do desenvolvimento. Essa prática reduz o custo de correção de falhas e contribui para a construção de software mais seguro e resiliente.

Diante deste cenário, aumentar a segurança durante o processo de desenvolvimento de software é crucial para a instituição. Este trabalho visa analisar o uso incorreto de APIs de criptografia, uma vulnerabilidade comum que pode comprometer a confidencialidade e a integridade dos dados. Ao identificar e corrigir essas falhas, busca-se fortalecer a postura de segurança da instituição e garantir a proteção dos dados de seus clientes.

Dessa forma, esta pesquisa investiga, em sistemas bancários reais, a relação entre detecção de mau uso de APIs de criptografia e o respectivo custo operacional e energético de ferramentas SAST quando orquestradas em um *Pipeline DevSecOps Verde*. O estudo é

empírico e quantitativo, mantendo separação clara entre *caracterização e método* (Cap. 3) e *avaliação empírica* (Cap. 4), em alinhamento às diretrizes internas de segurança e às metas ESG/ASG da instituição.

1.2 Objetivos

O objetivo principal deste trabalho é analisar o uso inadequado de APIs de criptografia na instituição financeira, identificando vulnerabilidades que possam comprometer a segurança dos sistemas. A pesquisa avaliará a ocorrência dessas falhas em projetos reais e a eficácia das ferramentas de análise estática na sua detecção, além de examinar o impacto do uso dessas ferramentas no consumo de recursos computacionais.

Para alcançar esse objetivo geral, este estudo se propõe a atingir os seguintes objetivos específicos:

1. Revisar a literatura sobre vulnerabilidades e uso incorreto de APIs de criptografia.
2. Identificar projetos que utilizam inadequadamente essas APIs de criptografia.
3. Aplicar ferramentas de análise estática para detectar falhas de APIs de criptografia no código-fonte.
4. Avaliar a acurácia das ferramentas com base no *benchmark* Cambench.
5. Analisar o consumo de recursos computacionais das ferramentas utilizadas.
6. Integrar as ferramentas SAST à esteira CI/CD com medição de energia e emissões (p. ex., *CodeCarbon*), mantendo rastreabilidade por projeto e por ferramenta.
7. Analisar o consumo de recursos computacionais e o consumo de energia/emissões das ferramentas utilizadas, subsidiando decisões técnicas e metas ESG/ASG da instituição.

1.3 Justificativa

O tema desta pesquisa é altamente relevante, pois a segurança da informação é uma prioridade para instituições financeiras, que lidam com dados sensíveis de milhões de clientes. O uso incorreto de APIs de criptografia pode comprometer a confidencialidade e integridade dessas informações, tornando os sistemas vulneráveis a ataques cibernéticos. Além disso, os projetos selecionados para análise nesta pesquisa são estratégicos para a instituição, pois envolvem dados críticos, como informações relacionadas a recursos humanos

e folha de pagamento. Dessa forma, a identificação e mitigação de vulnerabilidades nesses sistemas podem contribuir diretamente para a proteção dos ativos digitais da organização.

Uma análise detalhada do uso inadequado de APIs de criptografia, realizada com ferramentas de análise estática de segurança amplamente reconhecidas no meio acadêmico, permitirá avaliar a eficácia dessas soluções na detecção de vulnerabilidades. Essa pesquisa contribuirá para o aprimoramento dos processos de desenvolvimento seguro na instituição, fornecerá subsídios para a comunidade científica e para outras organizações interessadas nessa segurança. Espera-se que os resultados obtidos auxiliem na implementação de melhores práticas, reduzam riscos operacionais e promovam maior confiança na adoção de ferramentas de análise estática para prevenir falhas de criptografia em sistemas financeiros.

Ao incorporar medições de energia e emissões na esteira de desenvolvimento — em complemento às verificações SAST — a instituição alinha segurança de software às metas corporativas de ESG/ASG. Essa visibilidade operacional permite comparar alternativas técnicas com base não somente em eficácia de detecção, mas também em custo computacional e pegada ambiental, apoiando uma *TI Verde* relevante no contexto bancário.

Capítulo 2

Revisão de Literatura

A revisão de literatura desta pesquisa tem como objetivo contextualizar e fundamentar a análise de vulnerabilidades decorrentes do uso inadequado de APIs de criptografia e como essas falhas podem ser detectadas por ferramentas de análise estática de código. Para isso, inicialmente, são discutidos conceitos fundamentais sobre segurança de software e o papel das APIs de criptografia na proteção de dados. Em seguida, são apresentados estudos que investigam falhas na implementação dessas APIs e os desafios enfrentados por desenvolvedores ao utilizá-las corretamente. Posteriormente, são descritas as ferramentas de análise estática utilizadas para detectar essas vulnerabilidades. Por fim, são analisados trabalhos relacionados que discutem *benchmarks*, eficácia e desempenho dessas ferramentas, permitindo identificar lacunas na literatura e justificar a abordagem adotada nesta pesquisa.

2.1 Segurança de Software

A segurança de software constitui uma área da segurança da informação, pois trata da proteção dos sistemas computacionais contra falhas e ameaças que possam comprometer seus ativos. Segundo Mead [15], software seguro é aquele que preserva os três pilares clássicos da segurança da informação: confidencialidade, integridade e disponibilidade. Esses princípios podem ser violados por vulnerabilidades introduzidas durante o processo de desenvolvimento, seja por práticas inadequadas de codificação, decisões de arquitetura inseguras ou ausência de testes apropriados.

As vulnerabilidades de software podem originar-se tanto de falhas humanas quanto de pressões por entregas rápidas, que priorizam prazos curtos em detrimento de práticas essenciais de segurança. O desenvolvimento de sistemas geralmente segue um ciclo estruturado, conhecido como *Software Development Lifecycle* (Security Development Lifecycle (Ciclo de Vida do Desenvolvimento de Software Seguro) (SDL)), composto por fases como levantamento de requisitos, codificação e testes. No entanto, assegurar a

segurança requer a integração de práticas específicas em cada etapa. Para isso, propõe-se o uso do *Security Development Lifecycle* Software Development Lifecycle (Ciclo de Vida do Desenvolvimento de Software) (SDLC), um modelo complementar voltado à inserção de atividades de segurança no processo. Segundo Ransome [21], o SDLC visa construir segurança desde as fases iniciais. Essa visão é reforçada por Mcgraw [14], que destaca a importância de "construir segurança" no código, e por Mead [15], ao enfatizar que práticas contínuas reduzem riscos e custos de correção.

Parte das falhas exploradas em ataques reais resulta de vulnerabilidades evitáveis, como ausência de validação de entrada, configurações permissivas e uso de bibliotecas inseguras. Para Mcgraw [14], muitos desses erros são consequências da confusão entre segurança de aplicações e segurança de software. Enquanto a primeira foca na proteção do ambiente de execução, a segunda busca prevenir falhas desde o código-fonte, integrando a segurança de forma estrutural no produto.

Confiar somente na qualidade aparente do código não é suficiente. Embora código bem estruturado contribua para a manutenibilidade, ele não garante, por si só, a segurança. A ausência de práticas formais e automatizadas de verificação pode permitir que falhas críticas passem despercebidas. Como destacam Mead [15], o uso de ferramentas de análise estática, testes dinâmicos e avaliações automatizadas desempenha um papel essencial na detecção precoce de vulnerabilidades, reduzindo o custo de correções e melhorando a confiabilidade do sistema antes da implantação.

Além disso, segundo Ransome [21], o processo de desenvolvimento seguro requer mais do que tecnologia: envolve o fortalecimento da cultura organizacional, a capacitação de equipes e a definição clara de papéis de segurança, como arquitetos e desenvolvedores especializados. A atuação coordenada de profissionais com visão de todo o processo é indispensável para aplicar corretamente princípios como o de privilégio mínimo, defesa em profundidade e validação contínua da superfície de ataque. Como defendido por Mcgraw [14], construir segurança é mais eficiente e sustentável do que remediar falhas após o produto estar em produção.

2.2 Desafios com o Uso de APIs de Criptografia

A criptografia é utilizada para garantir a confidencialidade, integridade e autenticidade dos dados em sistemas computacionais. Para facilitar sua implementação, bibliotecas e APIs de criptografia foram desenvolvidas, permitindo que os programadores integrem funcionalidades de criptografia em seus códigos.

No entanto, a segurança de uma aplicação que utiliza criptografia não depende apenas da robustez dos algoritmos, mas também da forma como as APIs são usadas. Cito and

Gall [6] destacam que o uso incorreto dessas APIs pode resultar em vulnerabilidades. Tais usos incorretos envolvem a utilização de algoritmos desatualizados, geração de chaves fracas e configurações incorretas de parâmetros de segurança. Essas vulnerabilidades podem ser exploradas por atacantes para comprometer a integridade dos sistemas e acessar informações sigilosas [5].
A literatura converge para um conjunto recorrente de usos inadequados de APIs de criptografia em Java — p. ex., algoritmos obsoletos ou quebrados (*brokencrypto*), funções de *hash* inseguras (*brokenhash*), modos frágeis como ECB (*ecbmode*), geradores de números aleatórios previsíveis (*insecurerandom*), parâmetros inadequados de derivação de chaves/-senhas (*pbeparameters*), chaves abaixo do recomendado (*smallkeysize*) e IVs estáticos (*staticiv*) [5, 6, 20, 23]. *benchmarks* como o CamBench organizam casos nessas categorias para permitir avaliação comparável entre ferramentas.

Nadi et al. conduziram um estudo empírico para entender os desafios enfrentados por desenvolvedores ao utilizar APIs de criptografia em Java [19]. As fontes de dados para o estudo foram a análise de 100 postagens no StackOverflow, revisão de 100 repositórios no GitHub, um estudo piloto com 11 desenvolvedores e uma pesquisa ampliada com 37 desenvolvedores. Os resultados mostraram que a principal dificuldade está na complexidade das APIs e na falta de documentação adequada, levando a erros de implementação e insegurança no uso de criptografia. Além disso, os desenvolvedores relataram dificuldades na escolha de algoritmos adequados, configuração do ambiente e compreensão das mensagens de erro. Como solução, os autores recomendam a criação de APIs mais abstratas, documentação baseada em tarefas comuns e ferramentas que auxiliem na configuração e no uso seguro das bibliotecas de criptografia.

Meng et al. investigaram desafios e vulnerabilidades em práticas de codificação segura em Java, analisando 503 postagens no StackOverflow [16]. O objetivo foi identificar preocupações comuns dos desenvolvedores, obstáculos na implementação de segurança e vulnerabilidades recorrentes. A metodologia envolveu a extração automatizada e análise manual das postagens, classificando questões sobre APIs de criptografia, *Spring Security* e segurança em Java Platform, Enterprise Edition (Plataforma Java, Edição Corporativa) (JavaEE). Os achados indicam que erros frequentes decorrem da falta de suporte adequado para o uso seguro de APIs, levando desenvolvedores a adotarem práticas inseguras, como a desativação de proteções contra *Cross-Site Request Forgery* (Cross-Site Request Forgery (Falsificação de Requisição entre Sites) (CSRF)) e a aceitação de certificados Secure Sockets Layer (Camada de Soquetes Seguros) (SSL) sem validação. O estudo destaca a necessidade de ferramentas automatizadas para a prevenção de vulnerabilidades e melhor documentação para evitar falhas de segurança em aplicações Java.

O estudo de Rauf et al. investigou os fatores que impedem os desenvolvedores de

implementar código seguro, apesar da disponibilidade de ferramentas de segurança [22]. A pesquisa empregou uma abordagem interdisciplinar, combinando revisão de literatura sobre segurança em software com teorias da psicologia cognitiva e social. A metodologia incluiu uma meta-análise de estudos prévios e a caracterização das principais intervenções de segurança disponíveis. Os autores identificaram que as dificuldades enfrentadas pelos desenvolvedores se devem à falta de conhecimento, atenção e intenção, os quais impactam negativamente a adoção de práticas seguras. O estudo sugere o conceito de *adaptive security interventions*, que ajustam as recomendações de segurança ao contexto e às necessidades individuais dos desenvolvedores, tornando a segurança mais acessível e eficaz. Os resultados destacam que intervenções que integram fatores técnicos, cognitivos e contextuais podem aprimorar significativamente a segurança do software, facilitando sua adoção e reduzindo vulnerabilidades no desenvolvimento de aplicações.

Em resumo, a literatura indica que o uso inadequado de APIs de criptografia é, em grande parte, consequência da complexidade das bibliotecas e da falta de conhecimento especializado por parte dos desenvolvedores. Vulnerabilidades recorrentes incluem o uso de algoritmos obsoletos, chaves fracas e configurações inadequadas de parâmetros de segurança, o que pode comprometer aplicações financeiras e outros sistemas críticos. Para lidar com esses desafios, ferramentas de análise estática têm sido desenvolvidas para detectar esses problemas de forma automatizada, permitindo uma abordagem proativa na mitigação de riscos de segurança.

2.3 Uso de Análise Estática para Identificar Vulnerabilidades

Ferramentas de análise estática são utilizadas para detectar vulnerabilidades sem a necessidade de executar o código-fonte. Nesta pesquisa, quatro ferramentas serão empregadas: Cognicrypt, CryptoGuard, Horusec e SecBugs. Estudos anteriores, como [12], [9], [20] avaliaram a eficácia dessas ferramentas em detectar falhas em aplicações Java. Cada uma dessas soluções apresenta diferentes abordagens e técnicas, como análise de fluxo de dados e regras baseadas em padrões.

2.3.1 Análise Estática

A análise estática examina código-fonte ou *bytecode* sem execução para identificar padrões de risco, más práticas e vulnerabilidades antes da implantação. Ferramentas de *Static Application Security Testing* (Teste Estático de Segurança de Aplicações — SAST) apoiam a detecção precoce e a redução do custo de correção ao longo do ciclo de desenvolvimento [26].

Em paralelo, referências como OWASP Top 10 e CWE/SANS Top 25 orientam a priorização de riscos e o mapeamento entre fraquezas conhecidas e regras das ferramentas, favorecendo triagem e correção sistemáticas [11]. Em ambientes corporativos, o uso recorrente de SAST também implica custos operacionais (tempo, memória/CPU) e impacto em Continuous Integration (Integração Contínua) (CI)/Continuous Delivery (Entrega Contínua) (CD), dimensão tratada nesta dissertação ao lado de acurácia e cobertura.

Além de aspectos técnicos, a literatura recente também tem destacado o *custo operacional* associado ao uso de ferramentas SAST em escala, incluindo tempo de execução, consumo de memória e impacto na infraestrutura de *Continuous Integration* (CI). Em ambientes corporativos, especialmente aqueles com grandes bases de código e múltiplas esteiras em paralelo, análises frequentes podem gerar um volume significativo de processamento, com reflexos tanto em custos de infraestrutura quanto em consumo de energia [13]. Nesse contexto, surge uma linha complementar de pesquisa em *engenharia de software sustentável*, que investiga como decisões de arquitetura, configuração de pipelines e escolha de ferramentas influenciam o consumo energético de atividades de desenvolvimento, testes e verificação automática [24]. Além de acurácia e desempenho, esta dissertação inclui o consumo de energia e as emissões estimadas de CO_2 associadas à execução de ferramentas SAST sobre conjuntos de projetos Java reais.

Tabela 2.1: Ferramentas SAST consideradas nesta dissertação (visão comparativa).

Ferramenta	**Foco principal**	**Escopo / linguagem**	**Suporte a APIs de criptografia**	**Avaliação / evidências na literatura**
CogniCrypt	Uso correto de APIs de criptografia	Java (código-fonte, integração com Eclipse)	Regras específicas para JCA/JCE, geração de código seguro	Avaliado em estudos empíricos com projetos Java reais [9]
CryptoGuard	Uso inseguro de criptografia e SSL/TLS	Java (projetos de grande porte, Apache, Android)	16 categorias de falhas de cripto e configuração insegura	Avaliação em 46 projetos Apache e milhares de apps Android [20]
Horusec	Orquestração de scanners SAST/SCA em CI/CD	Múltiplas linguagens (inclui Java)	Detectores para padrões de segurança, incluindo alguns de cripto	Documentação e estudos de uso em pipelines reais [7, 8, 12]
SecBugs	Vulnerabilidades gerais de segurança em Java/Android	Java e bytecode (plugin sobre SpotBugs)	Detectores específicos para uso inseguro de APIs de criptografia	Forte uso na prática; pouca avaliação acadêmica formal [2, 25]

2.3.2 Cognicrypt

O CogniCrypt foi desenvolvido por um grupo de pesquisadores da *Paderborn University, University of Alberta* e *Technische Universität Darmstadt* [9]. O projeto surgiu da necessidade de auxiliar desenvolvedores na utilização correta de APIs de criptografia, uma vez que estudos indicam que a maioria dos sistemas que utilizam essas APIs contém vulnerabilidades. Pesquisas anteriores mostraram que aproximadamente 90% das aplicações que empregam criptografia apresentam pelo menos um erro de uso de API, o que pode comprometer a segurança dos dados [9]. Dessa forma, a proposta da ferramenta é minimizar esses erros e fornecer suporte aos programadores durante o desenvolvimento de aplicações.

A ferramenta CogniCrypt atua por meio de duas abordagens principais: a geração automática de código seguro e a análise estática do código-fonte [9]. A primeira funcionalidade permite que os desenvolvedores selecionem tarefas de criptografia comuns

e obtenham implementações seguras automaticamente. Já a análise estática identifica falhas de segurança, emitindo alertas quando práticas inseguras são detectadas. Entre as vulnerabilidades encontradas estão o uso inadequado de algoritmos de criptografia, a escolha de cifras inseguras e a reutilização de vetores de inicialização fixos, problemas frequentemente associados a implementações deficientes de criptografia [9].

Para validar sua eficácia, o CogniCrypt foi avaliado por meio de sua integração ao ambiente Eclipse, permitindo que desenvolvedores utilizassem a ferramenta em projetos reais e recebessem retorno automático sobre erros de criptografia [9]. Os principais benefícios incluem facilidade de uso, suporte contínuo e prevenção de vulnerabilidades, tornando a ferramenta útil para programadores com pouco conhecimento em segurança. No entanto, há algumas limitações, como o suporte exclusivo para Java, a dependência do Eclipse e a cobertura restrita a um conjunto específico de tarefas de criptografia [9].

2.3.3 Cryptoguard

Pesquisadores da *Virginia Tech University* e *University of Texas at Dallas* desenvolveram o CryptoGuard para aprimorar a detecção de vulnerabilidades de criptografia em projetos Java de grande escala. A motivação para essa proposta surgiu da necessidade de reduzir falsos positivos em ferramentas de análise estática, que frequentemente geram alertas irrelevantes, dificultando a adoção por desenvolvedores e empresas. O objetivo do grupo foi criar uma solução mais precisa e escalável, permitindo que falhas no uso de APIs de criptografia fossem identificadas de maneira confiável e aplicável em grandes bases de código [20].

O CryptoGuard opera como uma ferramenta de análise estática especializada na detecção de vulnerabilidades em APIs de criptografia. Utiliza *slicing* de programas, técnica que rastreia a origem e o impacto de variáveis sensíveis, como chaves e senhas. Além disso, incorpora um mecanismo de refinamento de alertas, reduzindo falsos positivos ao eliminar elementos irrelevantes no fluxo de análise. A ferramenta é capaz de identificar 16 categorias de vulnerabilidades, incluindo uso de chaves previsíveis, criptografia fraca, configurações inseguras de SSL/TLS e senhas *hardcoded*, problemas comuns que podem comprometer a segurança de sistemas e aplicativos [20].

A avaliação empírica do CryptoGuard foi conduzida em 46 projetos Apache e 6.181 aplicativos Android, demonstrando uma precisão de 98,61% e uma redução de falsos positivos entre 76% e 80%. Esses resultados destacam seus principais benefícios, como alta escalabilidade, precisão aprimorada e ampla cobertura de vulnerabilidades de criptografia. No entanto, algumas limitações foram identificadas, incluindo falsos negativos ocasionais, a falta de análise baseada no estado dos objetos e o suporte restrito a projetos Java [20].

2.3.4 Horusec

A ferramenta de código aberto Horusec foi desenvolvida pela empresa Zup IT, com o objetivo de identificar vulnerabilidades de segurança em código-fonte por meio da análise estática SAST [7]. A ferramenta auxilia desenvolvedores na detecção precoce de falhas, permitindo a integração com pipelines de Continuous Integration / Continuous Deployment (Integração Contínua / Entrega Contínua) (CI/CD) para automatizar o processo de verificação de segurança e pode ser utilizada via linha de comando Command Line Interface (Interface de Linha de Comando) (CLI) oferecendo suporte a diversas linguagens de programação [8].

O Horusec atua como um orquestrador de outras ferramentas de segurança e identifica vulnerabilidades ou falhas de segurança em projetos e armazena os resultados em um banco de dados para análise e geração de métricas. Entre os benefícios da ferramenta estão a automação da segurança e a compatibilidade com múltiplas linguagens, enquanto suas limitações incluem a possibilidade de falsos positivos e a necessidade de configuração adequada para a melhor precisão dos resultados [8]. Embora o Horusec também possa identificar falhas em APIs de criptografia, sua utilidade vai além disso, sendo uma ferramenta eficiente para detectar diferentes tipos de vulnerabilidades de segurança no código.

2.3.5 SecBugs

O SecBugs é uma ferramenta de análise estática para código Java, derivada do FindBugs, criada na Universidade de Maryland, e atualmente mantida por uma comunidade *open source* [25]. Foi proposta com o objetivo de identificar automaticamente padrões de código que possam indicar falhas antes da execução dos programas. Com a interrupção do projeto original, o SecBugs surgiu para manter sua continuidade e adaptá-lo a versões mais recentes da linguagem Java.

O SecBugs, mantido por Philippe Arteau, é um *plugin* complementar voltado à detecção de vulnerabilidades de segurança no código [2]. Embora inclua detectores específicos para o uso inseguro de APIs de criptografia, sua cobertura é maior, englobando diferentes classes de vulnerabilidades em aplicações Java e Android. Apesar do uso recorrente em práticas de desenvolvimento seguro, não foram identificadas publicações científicas que apresentem avaliações empíricas formais sobre a eficácia dessas ferramentas. Entre seus benefícios estão a gratuidade, a integração com pipelines de CI/CD e a personalização de regras; já entre as limitações, destacam-se os falsos positivos e a escassez de validação acadêmica [2, 25].

2.4 Trabalhos Relacionados

Estudos anteriores têm avaliado a eficácia de ferramentas de análise estática na detecção de vulnerabilidades em código Java [4, 12, 17]. A precisão dessas ferramentas é uma preocupação central, com alguns trabalhos utilizando *benchmarks* sintéticos, como o Juliet Test Suite [17] e o Open Web Application Security Project (Projeto Aberto para Segurança em Aplicações Web) (OWASP) *Benchmark* [12], para quantificar a capacidade das ferramentas em identificar vulnerabilidades conhecidas. No entanto, estudos anteriores apontam que a avaliação em *datasets* sintéticos pode levar a uma percepção enviesada da eficácia real das ferramentas, dada a simplicidade das vulnerabilidades em comparação com cenários do mundo real [13].

Em relação ao consumo de recursos computacionais, a análise do tempo de execução e do uso de memória é crucial, especialmente em projetos de grande escala [13]. Estudos demonstraram que o tempo de análise pode variar significativamente entre as ferramentas, apresentando um aumento considerável no tempo de execução em projetos maiores [13], enquanto outras exibem menor sensibilidade ao tamanho do projeto devido a técnicas de processamento paralelo [13]. Adicionalmente, a identificação das principais classes de erros e vulnerabilidades detectadas pelas ferramentas é essencial para compreender suas limitações e áreas de foco [17]. A análise manual dos resultados e a comparação com classificações de Common Weakness Enumeration (Enumeração Comum de Fraquezas) (CWE) (Common Weakness Enumeration) podem revelar quais tipos de vulnerabilidades são mais facilmente detectados e quais são frequentemente negligenciados [13].

Embora muitos estudos foquem em tempo de execução e uso de memória como indicadores de custo computacional [13], há um interesse crescente em métricas de *consumo de energia* e emissões de CO_2 associadas ao processo de desenvolvimento e verificação de software. Ferramentas de análise estática, compilação e testes automatizados são executadas repetidamente em pipelines de CI, o que pode representar uma parcela significativa do consumo energético de equipes de desenvolvimento de grande porte [24]. Assim, a avaliação de ferramentas SAST sob a perspectiva de energia não é só uma questão de eficiência técnica, mas também de sustentabilidade e de planejamento de capacidade de infraestrutura.

Para viabilizar esse tipo de avaliação, surgiram bibliotecas específicas para estimar energia e emissões a partir de medições de tempo, utilização de CPU/RAM e perfis de hardware. Um exemplo é o *CodeCarbon*, um *framework* leve que monitora o consumo de recursos e estima o gasto energético total (kWh) e as emissões associadas de CO_2e, a partir de fatores de emissão específicos por região geográfica [10]. Embora o CodeCarbon tenha sido inicialmente adotado em cenários de aprendizado de máquina, sua abordagem é genérica e pode ser integrada a experimentos com ferramentas de análise e pipelines de CI, permitindo comparar diferentes configurações sob a ótica de energia e emissões. Nesta

dissertação, essa biblioteca é utilizada para instrumentar as execuções das ferramentas CogniCrypt, CryptoGuard, Horusec e SecBugs, de modo a quantificar o custo energético médio por projeto e investigar a associação entre tempo de execução, porte do sistema e consumo de energia.

Diferentemente dos *benchmarks* existentes, como CryptoAPI-Bench [20], Braga Benchmark [5] e Ghera Benchmark [18], este estudo emprega dois *datasets* distintos. O primeiro é o CamBench, desenvolvido pela comunidade com foco em transparência e extensibilidade, utilizado nesta pesquisa para avaliar a acurácia das ferramentas de análise estática voltadas à detecção de uso indevido de APIs de criptografia em Java [23].

O segundo é um *benchmark* empírico, construído no contexto desta dissertação, a partir de 211 artefatos Java (`.jar`) utilizados em sistemas reais de uma instituição financeira. A seleção desses artefatos considerou critérios como volume estimado de código, criticidade e relevância institucional. Este *benchmark* foi concebido com o objetivo de avaliar a escalabilidade das ferramentas em ambientes reais de desenvolvimento.

As ferramentas de análise selecionadas para este estudo — CogniCrypt, CryptoGuard, SecBugs e Horusec — representam abordagens complementares na detecção de vulnerabilidades em APIs de criptografia em Java, sendo relevantes no contexto de instituições financeiras. O CogniCrypt auxilia desenvolvedores na utilização segura dessas APIs por meio de análises estáticas contínuas [9], enquanto o CryptoGuard emprega técnicas de *program slicing* para identificar falhas relacionadas ao uso incorreto de APIs de criptografia e ao uso de SSL/TLS com alta precisão, mesmo em projetos de grande escala [20]. O SecBugs, por sua vez, é voltado à auditoria de segurança de aplicações web Java [3], e o Horusec integra outras ferramentas, como GitLeaks e OWASP Dependency-Check, ampliando sua cobertura para detectar vulnerabilidades e exposição de segredos no código-fonte [12].

A escolha dessas ferramentas visa cobrir diferentes dimensões da segurança de software, desde o suporte à escrita de código seguro até a identificação de vulnerabilidades e dados sensíveis em ambientes complexos. Ao avaliá-las com o *benchmark* CamBench, este estudo busca evidenciar suas capacidades e limitações na proteção de aplicações contra o uso incorreto de APIs de criptografia.

Em síntese, a literatura aponta que (i) o uso incorreto de APIs de criptografia é recorrente e difícil de evitar na prática, (ii) ferramentas SAST especializadas, como CogniCrypt e CryptoGuard, oferecem suporte relevante, mas ainda com limitações em termos de cobertura, falsos positivos e escalabilidade, e (iii) há um interesse crescente em avaliar não apenas a acurácia, mas também o custo computacional e energético da adoção dessas ferramentas em ambientes reais.

Lacunas na literatura: (i) validade externa limitada, pois parte relevante das avaliações concentra-se em datasets sintéticos, com pouca evidência em cenários industriais [12, 17];

(ii) foco restrito à acurácia, já que o custo operacional (tempo/memória) aparece de modo fragmentado e raramente é comparado entre ferramentas [13]; (iii) energia e emissões, com escassez de estudos que quantifiquem consumo energético e CO_2e de SAST em fluxos de CI/CD [24]; (iv) harmonização de categorias, pela falta de padronização entre detectores e classes de misuse de criptografia, o que dificulta comparabilidade; e (v) contexto financeiro, com poucos resultados reportados em ambientes bancários e artefatos reais.

Posicionamento deste estudo: esta dissertação endereça (i) e (v) ao combinar o CamBench (para acurácia) com um benchmark institucional de 211 artefatos Java (para validade externa); responde a (ii) ao medir, de forma padronizada, tempo e memória por ferramenta; contribui para (iii) ao instrumentar as execuções com o CodeCarbon para estimar energia (Wh/kWh) e CO_2e; e mitiga (iv) ao normalizar categorias de misuse entre as ferramentas para permitir comparação consistente.

Diante dessas lacunas, o próximo capítulo detalha o desenho experimental que integra (a) avaliação de acurácia no CamBench e (b) análise de escalabilidade e energia em 211 artefatos reais, incluindo métricas e procedimentos de harmonização adotados para comparar CogniCrypt, CryptoGuard, Horusec e SecBugs.

Capítulo 3

Caracterização da Pesquisa

Apesar da literatura consolidada [4, 12, 17], persistem lacunas sobre a escalabilidade de ferramentas de análise estática em sistemas financeiros de grande porte. Avaliar, em conjunto, desempenho, consumo de energia e acurácia na detecção de uso indevido de APIs de criptografia é fundamental para orientar a adoção industrial. Este capítulo fixa o desenho do estudo: o CamBench é usado para aferir acurácia (RQ1), enquanto o conjunto de JARs reais sustenta a avaliação de desempenho (RQ2), classes de vulnerabilidade (RQ3) e da associação entre tempo e energia/CO_2 (RQ4), além de estabelecer reprodutibilidade dos procedimentos.

Em linha com a Seção 1.2, este estudo investiga o uso indevido de APIs de criptografia e a escalabilidade das ferramentas na detecção dessas vulnerabilidades. Adotamos uma abordagem empírica, de caráter quantitativo, avaliando projetos de diferentes tamanhos. As métricas de desempenho incluem tempo de execução, consumo de memória e uso de CPU; as métricas energéticas contemplam energia total (kWh), emissões de CO_2 (kg), duração da medição e a desagregação por CPU e RAM (respectivamente correspondentes às variáveis `energia_kwh_total`, `emissoes_kgco2`, `cc_duration_s`, `energia_kwh_cpu` e `energia_kwh_ram`). Com base nessa caracterização, investigamos as seguintes questões de pesquisa:

(RQ1): Qual é a acurácia das ferramentas CogniCrypt, CryptoGuard e SecBugs ao serem avaliadas pelo *benchmark* CamBench? As métricas (*precision*, *recall*, F1) comparam as saídas com o gabarito do CamBench. Como o Horusec não opera sobre `.jar`/*bytecode*, ele será executado sobre o código-fonte do subconjunto *Cap*; as demais ferramentas serão avaliadas sobre o `CamBench_Cap.jar`.

(RQ2): Qual é o custo computacional das ferramentas ao serem executadas em projetos Java reais de diferentes tamanhos e complexidades? Serão medidos tempo de execução, pico de memória e uso médio de CPU, controlando porte do artefato e ocorrência de *timeouts*.

(RQ3): Quais são as principais classes de vulnerabilidades identificadas pelas ferramentas em projetos reais? A categorização utiliza uma taxonomia canônica definida neste estudo (ver Seção 4.3.1); as contagens da RQ3 consideram os **211 artefatos reais**.
(RQ4): O tempo de execução prediz o consumo de energia e as emissões de CO_2 para cada ferramenta? A análise investigará a associação entre tempo de execução e consumo de energia/emissões por ferramenta (correlações e modelos com controle de escala).

3.1 Conjunto de Dados

Este estudo utiliza dois conjuntos complementares. O primeiro é o *benchmark* CamBench (subconjunto *Cap*), empregado para aferir acurácia (RQ1) na detecção de uso incorreto de APIs de criptografia Java. A medição energética no CamBench é opcional e não compõe as análises principais.

O segundo conjunto é formado por projetos reais da instituição financeira, distribuídos como artefatos `.jar`, e sustenta as análises de desempenho (RQ2), classes de vulnerabilidade (RQ3) e a relação tempo↔energia/CO_2 (RQ4). Como o código-fonte não está disponível para a maior parte dos projetos, a escala é caracterizada por *proxies* derivadas do `.jar`: tamanho (MB), número de classes, número de pacotes e bytes agregados dos `.class`, registradas em `reports/jars_proxies.csv`. A Tabela 3.1 resume os conjuntos e indicadores de escala.

Tabela 3.1: Conjuntos, uso no estudo e indicador de escala

Conjunto	**Uso no estudo**	**#Projetos**	**Escala**	**Energia medida**
CamBench (Cap)	Acurácia (RQ1)	1	N/A	Opcional
JARs reais	Desempenho (RQ2); Classes (RQ3); Tempo↔Energia (RQ4)	211	*jar_mb*, *num_classes*, *num_packages*	Sim

Estratificação por escala (clusters). Para análises comparativas, estratificamos os 211 JARs em três faixas de escala (*pequeno*, *médio*, *grande*) por tercis da distribuição de `jar_mb` (quantis 33% e 66%). Os rótulos resultantes são versionados em `reports/jars_proxies.csv` e no artefato auxiliar `clusters_jars.csv`. Essa estratificação é reutilizada nas análises de desempenho e energia do Cap. 4.

3.2 Coleta dos Dados

As execuções foram orquestradas via `executarScriptsCsv.py`, com entrada em `jars/` (nível superior) e saída em `reports/<execucao>/`, onde cada execução gera relatórios por ferramenta (CSV/JSON/HTML) e, quando habilitado, o arquivo de energia do CodeCarbon (`emissions.csv`). O ambiente de execução é controlado por *containers* Docker e versões do OpenJDK (8/11/17), com variáveis de ambiente documentadas nos *scripts*. O fluxo completo é mostrado na Figura 3.1.

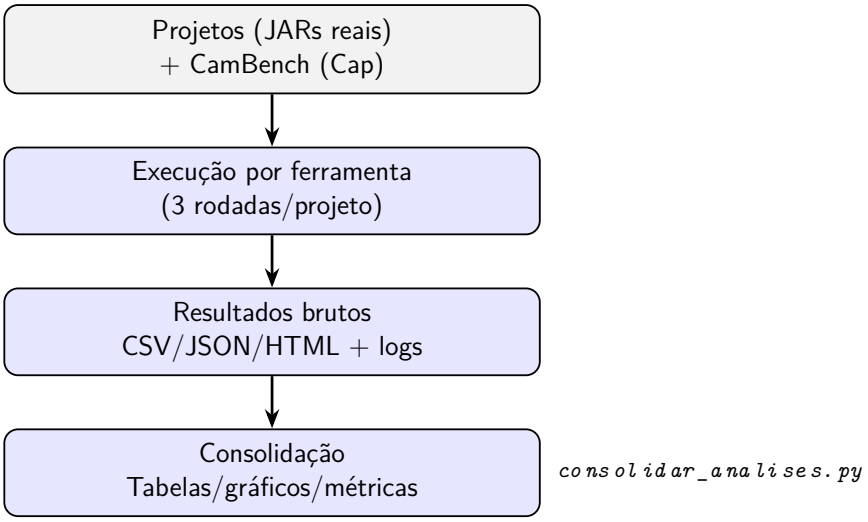

Figura 3.1: Fluxo de coleta e consolidação. Detalhes de harmonização são apresentados posteriormente (Seção 4.3.1).

Unidade de análise e deduplicação. A unidade primária de análise em sistemas reais é o **projeto**. Para evitar inflar contagens por verbosidade de relatório, deduplicamos achados por *projeto–categoria–ferramenta* (múltiplas ocorrências iguais no mesmo projeto contam como um). Essa convenção é aplicada após a etapa de preparação dos dados (ver Seção 4.3.1) e antes das agregações e comparações entre ferramentas.

Realizamos três rodadas por projeto e ferramenta, totalizando $211 \times 3 = 633$ execuções por ferramenta. Para avaliação, adotamos duas visões complementares: (i) *por execução*, voltada à robustez operacional; e (ii) *por projeto*, que agrega as três rodadas priorizando **ok** em relação a **erro**. Execuções marcadas como **nao_aplicavel** são **excluídas** das análises quantitativas.

Após a coleta, os resultados são integrados por `consolidar_analises.py` e `consolidar_cry pto_summaries.py`, produzindo `reports/consolidado/consolidado_analise_report_*.csv` e `reports/consolidado/resumo_tempo_medio_por_ferramenta_*.csv`. Em seguida, uma **etapa posterior de preparação dos dados** unifica esquemas e rótulos entre ferramentas; os

detalhes de harmonização (vocabulário canônico e mapeamento para CWE) são apresentados na Seção 4.3.1.

3.3 Metodologia de Avaliação

A arquitetura experimental está na Figura 3.2.

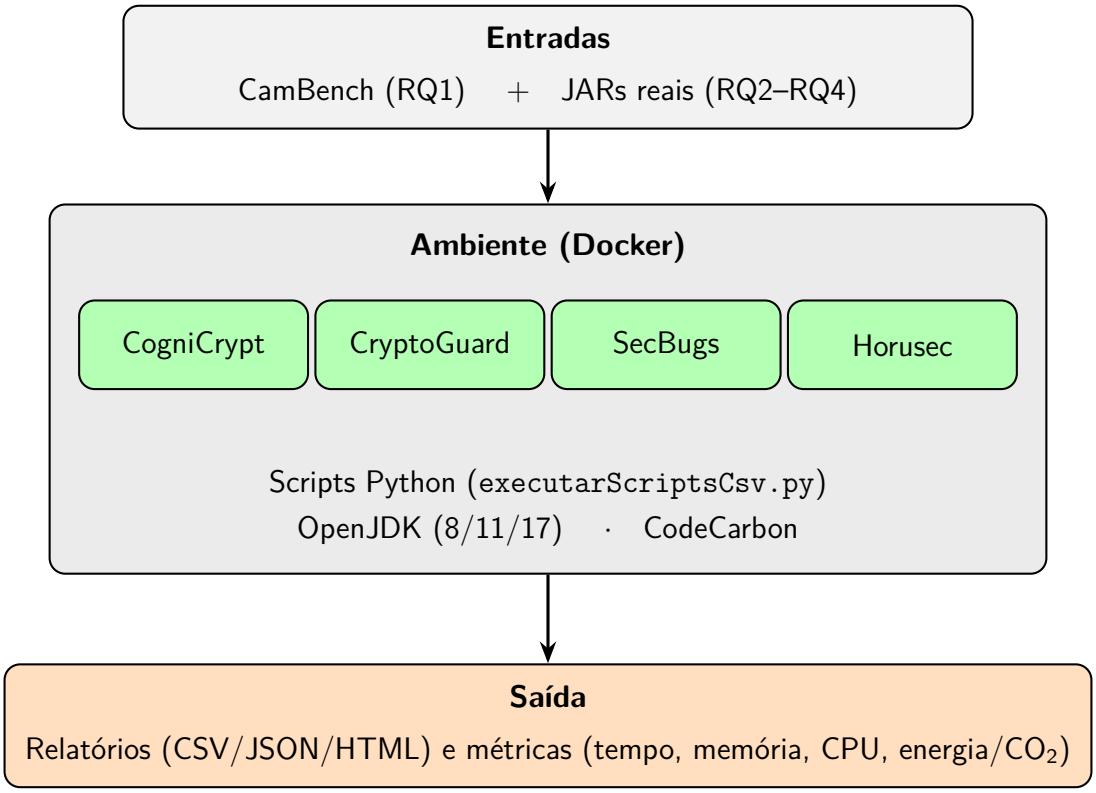

Figura 3.2: Arquitetura: entradas (CamBench para acurácia; JARs reais para desempenho/energia), ambiente (Docker, ferramentas, JDKs, scripts) e saídas (relatórios e métricas).

Em termos de GQM (Goal–Questions–Metrics), o objetivo é avaliar eficácia (acurácia) e custo operacional (desempenho/energia). As métricas adotadas seguem as definições operacionais deste capítulo e são obtidas dos artefatos consolidados e padronizados descritos nas Seções 3.1 e 3.2, com registros em `reports/consolidado/`.

3.3.1 Benchmarks

O CamBench (subconjunto *Cap*) fornece casos com gabarito para aferir acurácia (RQ1). Consideramos *falso positivo* quando a ferramenta reporta um caso inexistente no gabarito e *falso negativo* quando não sinaliza um caso presente. As fórmulas de *precision*, *recall* e

F1 estão na Seção 3.4. Observação: Horusec não opera sobre artefatos `.jar`/bytecode, sendo então utilizado o código-fonte para aferir o CamBench_Cap.

Com o objetivo de avaliar escalabilidade e custo operacional em cenários reais, utilizamos um conjunto de **211 artefatos Java** no formato `.jar`, oriundos de sistemas de uma instituição financeira. A seleção considerou criticidade, volume estimado de código e relevância institucional; os artefatos foram obtidos de repositórios internos, mantendo o formato de distribuição. Este benchmark sustenta as análises de desempenho (tempo, memória, CPU) e de energia/emissões (kWh, CO_2) nas RQ2–RQ4.

3.3.2 Avaliação de Desempenho

Serão considerados tempo total, pico de memória e uso médio de CPU por *execução* e *por projeto*, com agregação de três rodadas e controle de escala por proxies do JAR (*jar_mb*, *num_classes*, *num_packages*). As contagens de status (OK/Timeout/Erro) e os resumos descritivos aparecem no Cap. 4.

3.4 Análise e Síntese dos Dados

No contexto desta pesquisa, um *falso positivo* ocorre quando a ferramenta sinaliza uma vulnerabilidade que não está presente no gabarito do CamBench; um *falso negativo* corresponde à ausência de sinalização para uma vulnerabilidade existente. Esses conceitos fundamentam as métricas *precision*, *recall* e F1.

As fórmulas empregadas são:

- $Precision = \frac{TP}{TP+FP}$
- $Recall = \frac{TP}{TP+FN}$
- $F1\text{-}score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$

Para desempenho (RQ2) e energia (RQ4), sintetizamos tempo total de execução, pico de memória, uso médio de CPU e métricas energéticas (kWh, CO_2), relacionando-as com a escala expressa por proxies de JAR (*jar_mb*, *num_classes*, *num_packages*). Resultados consolidados são derivados de `reports/consolidado/consolidado_analise_report_*.csv` e dos artefatos em `reports/consolidado/padronizacao/`.

Capítulo 4

Experimentos, Resultados e Avaliação da Pesquisa

Este capítulo apresenta a **avaliação empírica** das quatro ferramentas SAST, organizada por questão de pesquisa em uma sequência *método sucinto* → *resultado* → *síntese*. As definições de conjuntos de dados, métricas, proxies de escala e ambiente estão no Cap. 3; os **detalhes de harmonização de rótulos** são apresentados aqui na Seção 4.3.1.

A escala dos artefatos é caracterizada por *proxies* derivadas do bytecode (*jar_mb*, *num_classes*, *num_packages*, bytes agregados de `.class`), conforme definido no Cap. 3 (`reports/jars_proxies.csv`). Esses proxies são usados como variáveis de controle nas análises.

Tabela 4.1: Resumo do ambiente de execução

Componente	Especificação
Containers (Docker)	Repositório/Tag: `projeto-analise:latest`; Image ID: `sha256:74aa7a0680e2c9cd6d09c33464b4615dd3659a7d908274d27674c5d923c2a2a3`; RepoDigest: (imagem local; sem manifesto de registro).
Hardware	CPU: Intel Core i7-8565U (4c/8t, base 1,8 GHz, turbo 4,6 GHz); RAM: 24 GB; Discos: NVMe 953,9 GB (raiz) + HDD 1,8 TB; SO: Ubuntu 24.04.3 LTS (noble); Kernel: 6.14.0-35-generic.
Java (OpenJDK)	Versões disponíveis: 8 (`1.8.0_462`), 11 (`11.0.28`), 17 (`17.0.16`) e 21 (`21.0.8`); avaliação principal com 8/11/17.
Timeouts por ferramenta	COGNICRYPT_TIMEOUT=1800 s; CRYPTOGUARD_TIMEOUT=1800 s; HORUSEC_TIMEOUT=1800 s; SECBUGS_TIMEOUT=1800 s; ORQ_TIMEOUT_PADRAO=1800 s.
CodeCarbon	Versão no host: 3.0.8; versão no container: 3.0.7; modo offline; CC_MEASURE_SECS=5; país = BR.
Timezone	`America/Sao_Paulo`.

4.1 Efetividade das Ferramentas no CamBench (RQ1)

A primeira questão de pesquisa investiga a acurácia das ferramentas CogniCrypt, CryptoGuard, Horusec e SecBugs ao serem avaliadas com o *benchmark* CamBench (subconjunto `Cap`). A avaliação considera as métricas clássicas de detecção—verdadeiros positivos (TP), falsos negativos (FN) e falsos positivos (FP), bem como as métricas derivadas de precisão (*precision*), (*recall*) e F1, analisadas tanto de forma global quanto por categoria de vulnerabilidade.

Escopo. O CamBench (subconjunto `Cap`) é utilizado para avaliar acurácia (RQ1). Métricas de tempo, throughput e energia são analisadas somente em sistemas reais (RQ2, RQ3 e RQ4). No caso específico do CamBench_Cap, CogniCrypt, CryptoGuard e SecBugs foram avaliadas sobre o artefato `CamBench_Cap.jar`, enquanto o Horusec foi executado sobre o código-fonte do pacote `org/cambench/cap`, uma vez que não opera diretamente sobre `.jar`/*bytecode*.

O subconjunto `Cap` do CamBench contém **576 casos** de teste rotulados, extraídos do pacote `org/cambench/cap`. Cada caso é identificado por um `case_id` e associado a uma `categoria` de vulnerabilidade de criptografia e a um rótulo de referência:

- `TP_ref`: casos em que há, no código, um uso indevido de API (ground truth positivo);
- `FP_ref`: casos em que o uso é seguro (ground truth negativo).

A Tabela 4.2 resume o gabarito por categoria. No total, são **375 casos positivos** (`TP_ref`) e **201 casos negativos** (`FP_ref`).

Tabela 4.2: RQ1 — Gabarito do CamBench_Cap por categoria.

Categoria	**# TP_ref**	**# FP_ref**	**Total**
`brokencrypto`	63	34	97
`brokenhash`	63	34	97
`ecbmode`	44	22	66
`insecurerandom`	44	22	66
`pbeparameters`	35	21	56
`smallkeysize`	63	34	97
`staticiv`	63	34	97
Total	375	201	576

Método de Avaliação. Geramos o *ground truth* diretamente do código fonte do CamBench_Cap (`org/cambench/cap`) por varredura de arquivos `.java`, extraindo `(case_id, categoria, rótulo)` com rótulo `TP_ref` ou `FP_ref`. Em seguida, normalizamos as saídas das quatro ferramentas para o formato `(tool, case_id, categoria)`, unindo com o *ground truth* por igualdade exata de `(case_id, categoria)`. A partir dos contadores agregados, derivamos *precision*, *recall* e F1 por ferramenta e por categoria.

O gabarito do CamBench_Cap (lista de casos positivos/negativos por categoria) foi obtido a partir do repositório oficial do *benchmark* e da sua documentação associada [23].
Nota interpretativa (TN/FP_ref). *TN* é contado quando, em casos `FP_ref`, a ferramenta *não marca* positivo. Se um motor não reporta TN explicitamente, isso decorre do desenho (negativo ≡ "não marcou"), não de erro. Mantemos assimetria TN/FP quando aplicável.

Resultados globais por ferramenta

A Tabela 4.3 apresenta os contadores agregados por ferramenta (TP/FN/FP/TN) e as métricas derivadas no CamBench_Cap.

Tabela 4.3: RQ1 — Acurácia no CamBench_Cap (contadores e métricas em %).

Ferramenta	TP	FN	FP	TN	Total	Precision (%)	Recall (%)	F1 (%)
CogniCrypt	375	0	201	0	576	65,10	100,00	78,86
CryptoGuard	175	200	87	114	576	66,79	46,67	54,95
Horusec	313	62	168	33	576	65,07	83,47	73,13
SecBugs	73	302	33	168	576	68,87	19,47	30,35

Nota: Para o *CogniCrypt*, FN = 0 e TN = 0 resultam do desenho do CamBench_Cap e do comportamento da ferramenta: o relatório indicou todos os casos rotulados como `truepositive` (Recall = 100%) e também todos os casos `falsepositive` (o que reduz a *Precision* para ≈ 65%). Assim, o total (375+201 = 576) coincide exatamente com o conjunto de referência do benchmark.

Em termos de F1 global, *CogniCrypt* apresenta o melhor equilíbrio entre *precision* e *recall* ($F1 \approx 0{,}79$), seguida de *Horusec* ($F1 \approx 0{,}73$). *CryptoGuard* obtém F1 intermediário ($\approx 0{,}55$), com bom *precision* mas *recall* mais baixo, enquanto *SecBugs* apresenta o menor F1 ($\approx 0{,}30$), sobretudo devido ao grande número de falsos negativos.

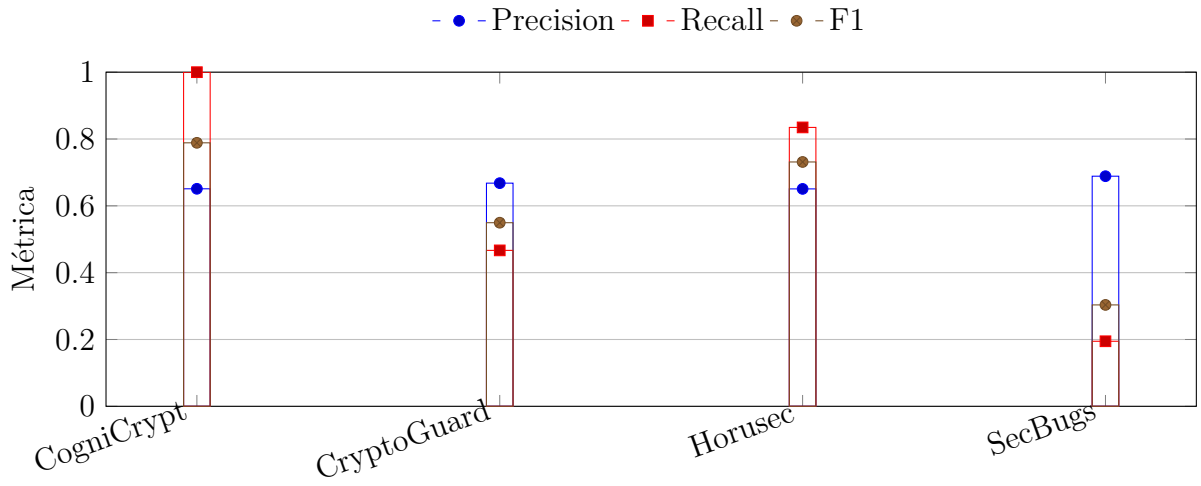

Figura 4.1: RQ1 — Precision, Recall e F1 por ferramenta no CamBench_Cap.

Resultados por categoria de vulnerabilidade

Para entender melhor em quais tipos de falha cada ferramenta se destaca ou falha, calculamos métricas por categoria de vulnerabilidade de criptografia. A Tabela 4.4 resume o F1 por categoria e por ferramenta no CamBench_Cap.

Tabela 4.4: RQ1 — F1 por categoria de vulnerabilidade no CamBench_Cap.

Categoria	**CogniCrypt**	**CryptoGuard**	**Horusec**	**SecBugs**
`brokencrypto`	0,788	0,626	0,788	0,219
`brokenhash`	0,788	0,614	0,031	0,118
`ecbmode`	0,800	0,623	0,800	0,125
`insecurerandom`	0,800	0,581	0,800	0,778
`pbeparameters`	0,769	0,500	0,769	0,000
`smallkeysize`	0,788	0,194	0,788	0,000
`staticiv`	0,788	0,579	0,788	0,376

A Tabela 4.4 mostra que *CogniCrypt* e *Horusec* mantêm F1 elevado e relativamente estável na maioria das categorias, com exceção de `brokenhash` para *Horusec*. *CryptoGuard* apresenta desempenho razoável em `brokencrypto`, `brokenhash` e `staticiv`, mas cai em `smallkeysize`. Já *SecBugs* tem desempenho bastante variável: vai de F1 próximo de zero em `pbeparameters` e `smallkeysize` até valores comparáveis às demais ferramentas em `insecurerandom`. Em conjunto, esses resultados sugerem que a escolha da ferramenta impacta diretamente quais padrões de falha de criptografia são mais (ou menos) cobertos na prática.

4.2 Performance das Ferramentas em Sistemas Reais (RQ2)

A segunda questão de pesquisa investiga o *custo computacional* das ferramentas SAST quando executadas em projetos Java reais com diferentes níveis de complexidade (tamanho). Esse custo é estimado em termos de tempo de execução, pico de uso de memória e utilização de CPU.

Coleta e Análise de Dados. Para cada projeto e para cada ferramenta, realizamos **três execuções independentes**, totalizando $211 \times 3 = 633$ execuções por ferramenta de análise estática. As análises foram conduzidas a partir de duas perspectivas complementares: (i) *por execução*, permitindo avaliar a robustez operacional e identificar falhas pontuais; e (ii) *por projeto*, agregando os resultados das três rodadas com base no melhor status observado (**OK** > **Timeout** > **Erro**). Aplicamos modelos de regressão para confirmar ou refutar os achados.

Estatísticas descritivas de desempenho

Antes de analisar a distribuição de tempo por projeto e estimar o modelo de regressão, apresentamos um resumo descritivo do comportamento de cada ferramenta no conjunto de 211 projetos. A Tabela 4.5 sintetiza, para cada ferramenta, o número de projetos com análise bem-sucedida (status final OK em até 1800 segundos), o tempo típico por projeto e a variação observada, bem como a proporção de projetos que terminaram com *timeout* ou erro.

Tabela 4.5: RQ2 — Resumo de tempo por projeto, por ferramenta (projetos com status OK e tempo $\leq$ 1800s)

Ferramenta	$\mathbf{N_{OK}}$	**Mediana (s)**	**Q1 (s)**	**Q3 (s)**	% não ok
CogniCrypt	177	6.15	5.14	8.16	16,1%
CryptoGuard	178	26.33	10.11	56.14	15,6%
Horusec	209	25.01	13.00	87.97	0,9%
SecBugs	157	201.74	20.72	692.31	25,6%

Complementando o tempo de execução, a Tabela 4.6 apresenta, para os mesmos projetos com `Status=ok` e tempo $\leq$ 1800s, o pico mediano de memória (RAM) e o uso mediano de CPU por ferramenta.

Tabela 4.6: RQ2 — Tempo de execução, pico de memória e uso de CPU por ferramenta (projetos com status OK e tempo $\leq$ 1800s)

Ferramenta	**Tempo mediano (s)**	**Pico de RAM mediano (MB)**	**CPU mediana (%)**
CogniCrypt	6.15	708	264.9
CryptoGuard	26.33	398	97.9
Horusec	25.01	228	401.5
SecBugs	201.74	2690	490.3

Nota. A métrica "CPU Processo (%)" representa a soma por núcleo (100% por núcleo). Assim, valores acima de 100% indicam paralelismo em múltiplos núcleos.

As duas tabelas, em conjunto, resumem o *custo computacional* (RQ2): tempo de execução, memória e CPU. Em linha com a caracterização de escala da Tabela 4.7, observa-se que, à medida que avançamos dos projetos menores para os maiores, o custo absoluto tende a crescer para todas as ferramentas, mas de forma desigual: *CogniCrypt* mantém tempo e consumo de recursos relativamente baixos; *CryptoGuard* e *Horusec* ocupam uma faixa intermediária; e *SecBugs* concentra os maiores tempos, maior pico de RAM e maior

uso típico de CPU. Essas diferenças de perfil serão retomadas na análise por clusters de tamanho e na análise de regressão do tempo.

Características dos projetos analisados

Os **211 projetos** sustentam RQ2, RQ3 e RQ4. A escala foi estratificada em três *clusters* (*pequeno*, *médio*, *grande*) por tercis de `jar_mb` (definição no Cap. 3). Na Tabela 4.7, apresentamos o rótulo `jar_mb` para facilitar a leitura.

Tabela 4.7: RQ2 — Caracterização dos projetos por tamanho de JAR (coluna `jar_mb`, em MB)

Cluster	**N**	% projetos	**Mediana JAR (MB)**	**[Q1, Q3] JAR (MB)**
Pequeno	70	33,2%	0,29	[0,07, 14,55]
Médio	71	33,6%	36,14	[24,41, 45,11]
Grande	70	33,2%	98,44	[92,83, 122,60]
Total	211	100%	36,14	[14,56, 92,12]

Essa caracterização mostra que o conjunto inclui projetos de portes distintos, desde JARs relativamente pequenos, até artefatos significativamente maiores. A divisão em três clusters de escala será reutilizada na discussão de tempo, permitindo avaliar se o impacto de cada ferramenta é semelhante em projetos menores e em projetos mais volumosos.

Visão geral de tempo por projeto

A Figura 4.2 resume o tempo de execução por projeto para cada ferramenta, considerando projetos com status final **OK** e tempo de execução menor ou igual a 1800 segundos. A escala do eixo y é logarítmica em base 2, o que evita compressão dos valores e permite comparar, na mesma figura, projetos muito rápidos e projetos mais custosos.

De forma geral, observa-se que o *CogniCrypt* apresenta tempos medianos de poucos segundos por projeto, com baixa dispersão. *CryptoGuard* e *Horusec* ocupam uma faixa intermediária, com medianas na ordem de dezenas de segundos e caudas mais longas, indicando que uma parte dos projetos exige execuções mais demoradas. O *SecBugs*, por sua vez, concentra as maiores medianas, na casa de centenas de segundos, além de maior variação entre projetos.

Esse contraste sugere que, mesmo sob o mesmo conjunto de 211 projetos reais, as ferramentas diferem de forma consistente em termos de tempo de análise: há uma ferramenta mais rápida (*CogniCrypt*), duas intermediárias (*CryptoGuard* e *Horusec*) e uma mais custosa em tempo (*SecBugs*). A próxima subseção investiga se parte dessa

diferença pode ser explicada pela *escala* dos artefatos (tamanho do JAR) em conjunto com a ferramenta.

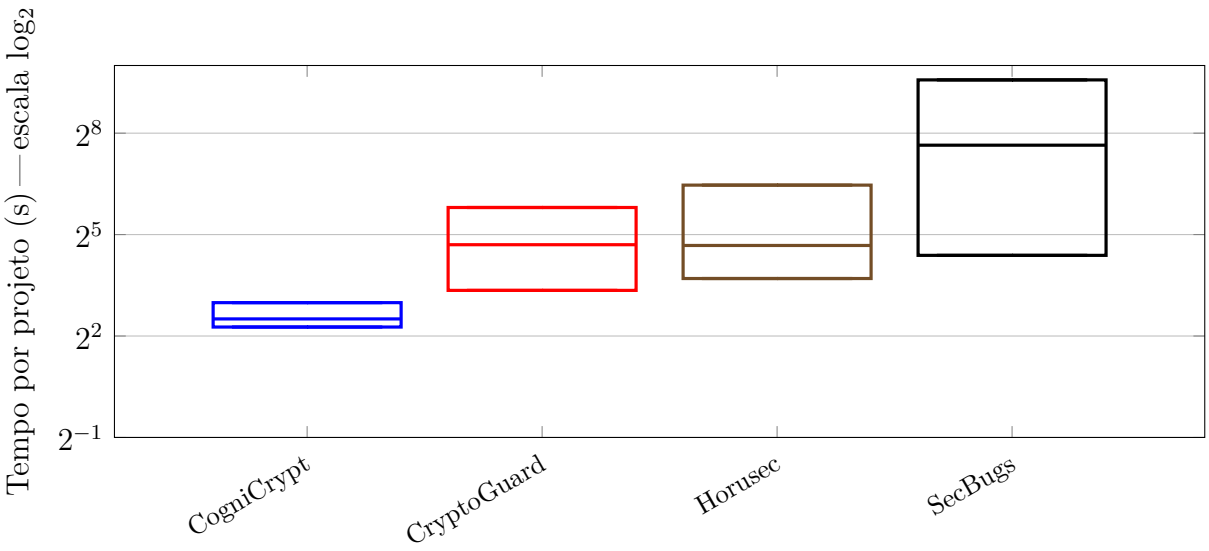

Figura 4.2: RQ2 — Boxplots de tempo por projeto (escala $\log_2$).

Efeito da escala e da ferramenta no tempo (regressão)

Para quantificar o efeito do tamanho do artefato sobre o tempo de execução, em conjunto com a ferramenta utilizada, estimamos o seguinte modelo linear:

$$\text{Tempo (s)} \sim \beta_0 \; + \; \beta_1 \cdot \log(\texttt{jar\_mb}) \; + \sum_{f \in \{\text{CG,HZ,SB}\}} \gamma_f \cdot \mathbf{1}\{\text{Ferramenta} = f\} \; + \; \varepsilon.$$

Nesse modelo, a variável resposta é o tempo de execução por projeto (em segundos). A variável $\log(\texttt{jar\_mb})$ representa a escala do artefato (tamanho do JAR, transformado em logaritmo natural), e a ferramenta é codificada como variável categórica, tendo *CogniCrypt* como categoria de referência. Cada linha do modelo corresponde a um par *(projeto, ferramenta)*, usando o tempo agregado por projeto (mediana das execuções bem-sucedidas para aquela ferramenta naquele JAR).

O modelo foi ajustado por mínimos quadrados ordinários, com erros-padrão robustos do tipo HC3. Essa correção busca tornar as inferências mais estáveis quando a variabilidade dos resíduos não é uniforme entre os projetos (heterocedasticidade), o que é comum em dados de desempenho.

A Tabela 4.8 apresenta o modelo ajustado, com coeficientes, intervalos de confiança de 95% e respectivos p-valores, além do R^2 e do número de observações (N) utilizados na regressão.

Tabela 4.8: RQ2 — Regressão de Tempo (s) por projeto: efeito de escala e ferramenta (coeficientes em segundos)

Variável	**Coef. (s)**	**IC95% (s)**	**p-valor**
Intercepto (ref: CogniCrypt)	-1.6	[-2.0, -1.2]	< 0.001
Ferramenta = Horusec	+19.6	[+13.0, +26.0]	$< 10^{-6}$
Ferramenta = CryptoGuard	+20.2	[+14.0, +27.0]	$< 10^{-6}$
Ferramenta = SecBugs	+194.0	[+176.0, +212.0]	$< 10^{-16}$
log(`jar_mb`)	+10.5	[+8.5, +12.5]	$< 10^{-10}$

Interpretação. O coeficiente de log(`jar_mb`) resume o aumento esperado de tempo quando o tamanho do JAR cresce: valores positivos e estatisticamente significativos indicam que projetos maiores tendem a demandar mais tempo de análise, mesmo após aplicar o controle pela ferramenta. Os coeficientes associados a *Ferramenta* = *X* quantificam o deslocamento médio em relação ao *CogniCrypt*: coeficientes positivos sugerem que a ferramenta correspondente é, em média, mais lenta; coeficientes negativos indicariam uma ferramenta mais rápida, para projetos de tamanho semelhante.

O ajuste global do modelo, sintetizado pelo R^2 e pelo número de observações N na Tabela 4.8, indica em que medida a combinação entre a escala do artefato e a escolha da ferramenta explica a variação observada no tempo por projeto. Em conjunto com os boxplots da Figura 4.2, o modelo evidencia que as diferenças entre ferramentas não se devem a casos pontuais: observa-se um padrão consistente em que *SecBugs* concentra os maiores tempos, *CryptoGuard* e *Horusec* situam-se em uma faixa intermediária, e *CogniCrypt* apresenta os menores tempos médios para projetos de tamanho comparável.

Síntese da Análise de Desempenho

Nos 211 projetos Java analisados da instituição, as quatro ferramentas exibem perfis distintos de desempenho. Em termos de *tempo por projeto*, *CogniCrypt* é a ferramenta mais rápida, *CryptoGuard* e *Horusec* ocupam posição intermediária e *SecBugs* apresenta os maiores tempos medianos e a maior taxa de *timeout*/erro. Quando incorporamos *pico de memória* e *uso de CPU*, o padrão se mantém: *CogniCrypt* consome menos RAM e CPU típicas, *CryptoGuard* e *Horusec* têm um perfil intermediário, e *SecBugs* combina maior tempo, maior uso de memória e maior ocupação de CPU.

O modelo de regressão confirma que o tamanho do artefato (medido por log(`jar_mb`)) tem efeito significativo sobre o tempo e que a escolha da ferramenta introduz diferenças sistemáticas adicionais. Assim, mesmo sob o mesmo conjunto de sistemas reais e controlando pela escala, o custo computacional para incorporar verificação de usos de criptografia na

esteira varia de forma consistente entre as ferramentas avaliadas, tanto em tempo quanto em memória e CPU.

4.3 Classes de Vulnerabilidade Detectadas em Sistemas Reais (RQ3)

Esta questão de pesquisa explora quais são as classes de vulnerabilidades identificadas pelas ferramentas em projetos Java reais da instituição e como essas classes se distribuem entre as ferramentas SAST avaliadas. A categorização segue a harmonização definida nesta seção (ver §4.3.1) e emprega deduplicação por *projeto–categoria–ferramenta*.

4.3.1 Harmonização de Rótulos e Mapeamento para CWE

Ferramentas distintas empregam taxonomias diferentes para classificar mau uso de criptografia. Para permitir comparação direta, adotamos um dicionário de harmonização em duas etapas: (i) mapeamos as *tags* do CamBench e os rótulos nativos das ferramentas para um vocabulário canônico de categorias usado neste trabalho; (ii) quando aplicável, associamos cada categoria canônica aos *Common Weakness Enumeration* (CWE) correspondentes. Os mapas servem para padronizar nomenclatura; as contagens de classes em sistemas reais não misturam casos do CamBench.

Tabela 4.9: Harmonização de rótulos do CamBench para categorias canônicas

CamBench tag	**Categoria canônica**
StaticIV	staticiv
ECBMode	ecbmode
WeakMessageDigest	brokenhash
WeakPRNG	insecurerandom
HardCodedKey	hardcodedkey
WeakCipher	brokencrypto
InsufficientKeySize	smallkeysize
PBEParameters	pbeparameters

Tabela 4.10: Categorias canônicas e CWEs correspondentes (quando aplicável)

Categoria canônica	**CWE(s)**
hardcodedkey	CWE-321
brokenhash	**CWE-328**
brokencrypto	CWE-327; (parcialmente) CWE-326
ecbmode	CWE-327
staticiv	CWE-329
insecurerandom	CWE-338
smallkeysize	CWE-326
pbeparameters	—

Validação. Automatizamos a verificação do dicionário com `validar_mapeamentos.py`, checando colunas esperadas, duplicatas e conflitos; os logs e versões estão em `reports/consolidado/padronizacao/`. Essa auditoria evita colisões semânticas e garante reprodutibilidade do processo de padronização.

Para comparar motores com taxonomias distintas, aplicamos o dicionário de harmonização do Cap. 3: rótulos nativos e tags foram mapeados para um vocabulário canônico (*insecurerandom*, *brokenhash*, *brokencrypto*, *ecbmode*, *staticiv*, *pbeparameters*, *smallkeysize*) e, quando aplicável, associados às CWEs correspondentes. A harmonização afeta apenas *nomes*; as contagens e percentuais desta seção provêm dos 211 sistemas reais, após deduplicação por *projeto–categoria–ferramenta*.

Nesta RQ3 adotamos a unidade "**projeto**" e deduplicamos por *projeto–categoria–ferramenta* (múltiplos achados iguais contam como 1) por três motivos: (a) **disponibilidade de dados** — os artefatos são JARs, sem árvore de fontes confiável que permita consolidar por classe ou método; (b) **comparabilidade entre ferramentas** — os motores emitem granularidades distintas (arquivo, classe, regra), e o nível projeto evita inflar contagens por verbosidade; (c) **uso operacional** — a priorização na esteira ocorre tipicamente por *projeto* afetado.

As ferramentas adotam taxonomias distintas para falhas de implementação de criptografia. Para permitir comparação direta, harmonizamos os rótulos em um núcleo comum de categorias: *insecurerandom*, *brokenhash*, *brokencrypto*, *ecbmode*, *staticiv*, *pbeparameters* e *smallkeysize*. Essa harmonização segue o mapeamento consolidado no Cap. 3 entre categorias do CamBench, rótulos internos das ferramentas e CWEs correspondentes, de modo a apoiar a análise de recorrência, cobertura e lacunas de detecção. Para a RQ3, focamos nas categorias em que há de fato achados nos sistemas reais, resultando no subconjunto *insecurerandom*, *brokenhash*, *brokencrypto*, *ecbmode* e *staticiv*. A categoria

genérica `cryptoapi` é analisada separadamente (por capturar o *uso* de APIs de criptografia, e não necessariamente mau uso).

Auditoria de normalização. Validamos os mapas de harmonização com um verificador automatizado (`validar_mapeamentos.py`), checando colunas, duplicatas e conflitos. Dois pontos merecem registro: (i) `pbeparameters` não aparece nos achados reais desta amostra (somente benchmark), e (ii) CWEs amplas (p.ex., **CWE-326** e **CWE-327**) cobrem múltiplas categorias canônicas; mantivemos essa propriedade, como no Cap. 3, para evitar sobre-ajuste de rótulos.[1]

Tabela 4.11: RQ3 — Projetos com achados por categoria e ferramenta (deduplicado por projeto).

Categoria	**CogniCrypt**	**CryptoGuard**	**Horusec**	**SecBugs**
brokencrypto	1	1	0	84
insecurerandom	0	0	0	84
brokenhash	0	0	0	24
ecbmode	0	0	0	10
staticiv	0	0	0	2
Total (linhas)	1	1	0	204

(i) Deduplicação por *projeto–categoria–ferramenta*; (ii) **Total** = união por projeto entre ferramentas; (iii) categorias harmonizadas via mapeamento CamBench/CWE (Cap. 3), usado para padronizar nomes.

Esclarecimento. As contagens refletem a aplicação consistente do mapeamento de categorias (Cap. 3) às saídas normalizadas de cada ferramenta. A assimetria entre motores decorre tanto de diferenças internas de regras/heurísticas quanto do nível de granularidade de emissão; por isso, mantivemos a deduplicação por *projeto–categoria–ferramenta* para assegurar comparabilidade.

A Tabela 4.11 resume os achados deduplicados por *projeto–categoria–ferramenta*. Na Figura 4.3 observa-se que, nas cinco categorias principais (*brokencrypto*, *insecurerandom*, *brokenhash*, *ecbmode* e *staticiv*), somente um projeto é marcado por *CogniCrypt* e um por *CryptoGuard*, enquanto o *SecBugs* concentra a maior parte dos achados (204 ocorrências nessa amostra). Essa assimetria entre ferramentas reforça a necessidade de triagem manual e discussão de possíveis falsos positivos associada à RQ3, retomada no capítulo de discussão.

Consenso entre ferramentas. Além das contagens por ferramenta, construímos um indicador de consenso por categoria, marcando projetos nos quais *duas ou mais ferramentas*

[1] Logs de verificação e mapas versionados em `reports/consolidado/padronizacao/`.

reportam a mesma categoria de falha. Esse consenso é importante porque combina evidências independentes: quando duas ferramentas diferentes sinalizam a mesma categoria em um projeto, a probabilidade de o achado ser relevante (e não apenas um falso positivo específico de uma ferramenta) tende a ser maior.

Distribuição de categorias por ferramenta

Para cada categoria de criptografia, a tabela 4.11 resume quantos projetos tiveram pelo menos um achado por ferramenta, bem como o total de projetos afetados em cada categoria (coluna **Total**). Já a Figura 4.3 apresenta a mesma informação em termos proporcionais, por meio de um gráfico de barras 100% empilhadas.

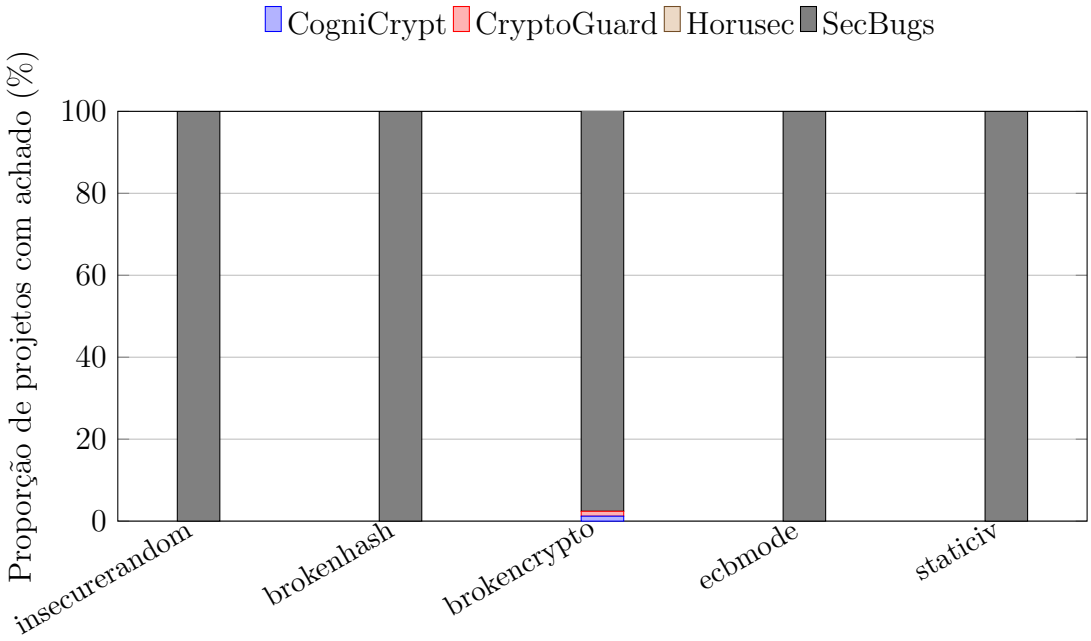

Figura 4.3: RQ3 — Distribuição proporcional de projetos com achados por categoria e ferramenta (100% empilhado, deduplicado por projeto).

Em termos absolutos, o subconjunto analisado concentra-se em poucos tipos de mau uso: *insecurerandom* e *brokenhash* estão presentes em dezenas de projetos, enquanto *brokencrypto*, *ecbmode* e *staticiv* aparecem com menor frequência.

Consenso e assimetria entre motores

Os arquivos consolidados de consenso indicam que, no recorte de categorias de criptografia, apenas um projeto apresenta consenso (duas ou mais ferramentas reportando a mesma categoria) e esse consenso ocorre em *brokencrypto*. Para as demais categorias (*insecurerandom*, *brokenhash*, *ecbmode*, *staticiv*), todos os achados são exclusivos do SecBugs.

Essa assimetria é coerente com o perfil da RQ1: SecBugs tende a reportar um número elevado de achados, tanto no CamBench quanto nos sistemas reais, enquanto CogniCrypt e CryptoGuard são mais conservadores, priorizando precisão em detrimento de cobertura. No contexto da RQ3, isso se manifesta em dois efeitos complementares:

- SecBugs domina a detecção das categorias de criptografia em sistemas reais, cobrindo praticamente todos os projetos com achados nessas categorias.
- O consenso entre ferramentas é raro, o que reforça a necessidade de triagem manual cuidadosa, sobretudo para categorias observadas apenas por um motor.

Leitura simples. Do ponto de vista de priorização na esteira, achados com consenso (duas ou mais ferramentas) oferecem um sinal mais forte para intervenção — especialmente em categorias como *brokencrypto*, que já possuem impacto conhecido em termos de confidencialidade e integridade. Já as categorias reportadas por uma única ferramenta, em especial quando concentradas em um motor mais "verboso" como o SecBugs, merecem atenção, mas demandam triagem manual para mitigar o risco de sobrecarga da equipe com falsos positivos.

4.4 Consumo de Energia e Emissão de CO_2 (RQ4)

Esta questão de pesquisa investiga se o tempo de execução prediz o consumo de energia e as emissões de CO_2 para cada ferramenta SAST, controlando pela escala dos projetos.

Coleta e Análise de Dados. Utilizamos as informações que tanto o *CodeCarbon* quanto o monitor de recursos coletam. Isso inclui:

- `energia_kwh_total`, `emissoes_kgco2`,
- `cc_duration_s`, `energia_kwh_cpu`, `energia_kwh_ram`,
- `cc_tracking_mode`,
- `cc_measure_power_secs`, `cc_online`,
- `cc_country_iso`, `cc_output_dir`,
- `escopo_analise`, `motivo_fim`.

Selecionamos apenas as execuções com `Status=ok` e, para cada par *(Analise, Projeto)*, agregamos as três rodadas pela mediana. Em seguida, calculamos médias por ferramenta, restringindo-nos a projetos com análise bem-sucedida. De forma consistente com o Cap. 3,

a escala do projeto é controlada por *proxies* derivadas do JAR (*jar_mb*, *num_classes*, *num_packages*), que entram na análise por meio da variável log(`jar_mb`) nos modelos de regressão. De forma geral, computamos as métricas Energia total (kWh) convertida para Wh (1 kWh = 1000 Wh), emissões (gCO_2e) a partir de $kgCO_2e$ (1 kg = 1000 g) e duração (s). A decomposição CPU/RAM é expressa em Wh por projeto. *GPU não se aplica ao ambiente de coleta e, por isso, não é utilizada nas figuras e tabelas.*

Energia, emissões e pico de RAM

A Tabela 4.12 apresenta o consumo médio de energia por projeto, as emissões médias de CO_2 equivalente e o pico médio de RAM para cada ferramenta, considerando apenas os projetos com `Status=ok`. Essas métricas permitem caracterizar o *custo energético médio por projeto* de cada ferramenta, complementando a análise de tempo realizada na RQ2.

Tabela 4.12: RQ4 — Energia, emissões e recursos (médias por projeto OK)

Ferramenta	**Wh/proj.**	**gCO_2e/proj.**	**Pico RAM (MB)**	**Obs.**
CogniCrypt	0,055	0,005	825	Médias por projeto com `Status=ok` (mediana das execuções).
CryptoGuard	0,220	0,022	649	Conversões: Wh = kWh×1000; gCO_2e = kg×1000.
Horusec	0,292	0,029	281	Valores médios por ferramenta; variações internas capturadas pelos boxplots de tempo.
SecBugs	1,134	0,109	2.439	Consumos maiores refletem tempos mais longos e uso mais intenso de RAM.

A Tabela 4.12 mostra que o perfil observado em tempo (RQ2) se reflete também em energia: *CogniCrypt* apresenta o menor consumo médio de energia e emissões por projeto, seguido por *CryptoGuard* e *Horusec*, enquanto *SecBugs* concentra os maiores consumos e o maior pico médio de RAM. Assim, além de ser mais custosa em tempo, *SecBugs* também impõe o maior custo energético sobre a infraestrutura de CI.

Decomposição CPU/RAM

Nota metodológica.

A decomposição em CPU e RAM é obtida diretamente do CodeCarbon (colunas `energia_kwh_cpu` e `energia_kwh_ram` no consolidado). Para entender melhor de onde vem o consumo de energia, decompomos a energia elétrica média por projeto em componentes atribuídos à CPU e à RAM. A Figura 4.4 apresenta essa decomposição, em Wh por projeto, para cada ferramenta.

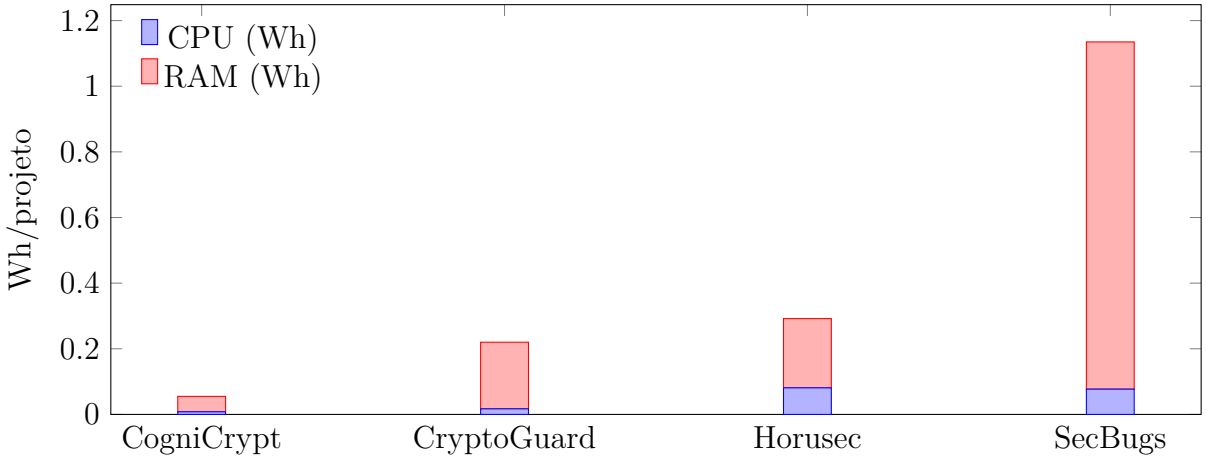

Figura 4.4: RQ4 — Decomposição média da energia por componente (Wh/projeto) — sem GPU.

Observa-se que, para *CogniCrypt*, o consumo é baixo em ambos os componentes (CPU e RAM), coerente com tempos curtos. Para *CryptoGuard* e *Horusec*, a maior parte da energia é atribuída à RAM, com uma contribuição de CPU ainda modesta. Em *SecBugs*, além de maior consumo de CPU, a parcela atribuída à RAM cresce de forma pronunciada, refletindo execuções longas e uso mais intensivo de memória. Esses padrões sugerem que a escolha da ferramenta impacta não apenas o tempo de análise, mas também a forma como os recursos são utilizados na esteira. Para quantificar a associação entre tempo de execução e consumo de energia, controlando por escala do projeto e ferramenta, estimamos o seguinte modelo linear:

$$\text{Energia (Wh)} \sim \alpha_0 + \alpha_1 \cdot \text{Duração (s)} + \sum_{f \in \{\text{CG,HZ,SB}\}} \delta_f \cdot \mathbb{I}\{\text{Ferramenta} = f\} + \alpha_2 \cdot \log(\texttt{jar\_mb}) + \varepsilon.$$

A variável resposta é a energia média por projeto (Wh). A duração é medida em segundos (`cc_duration_s`); a ferramenta é codificada como variável categórica, tendo *CogniCrypt* como categoria de referência; e $\log(\texttt{jar\_mb})$ representa a escala do artefato, obtida a partir do tamanho do JAR em MB. Cada observação corresponde a um par *(projeto, ferramenta)*, usando a mediana da energia e da duração nas execuções bem-sucedidas daquele projeto para aquela ferramenta.

O modelo foi ajustado por mínimos quadrados ordinários, com erros-padrão robustos do tipo HC3, de forma análoga à RQ2. Essa correção torna as inferências mais estáveis em cenários em que a variabilidade dos resíduos não é uniforme entre os projetos (heterocedasticidade), o que é esperado em dados de desempenho e energia.

A Tabela 4.13 apresenta um resumo dos coeficientes estimados, intervalos de confiança de 95% e p-valores, bem como o R^2 e o número de observações (N).

Tabela 4.13: RQ4 — Regressão de Energia (Wh) por projeto: efeito da duração, ferramenta e escala (resumo)

Variável	**Coef.**	**IC95%**	**p-valor**
Intercepto (ref: CogniCrypt)	-0.0217	[-0.0352, -0.00828]	0.00154
Duração (s)	0.00267	[0.00262, 0.00272]	$< 10^{-16}$
Ferramenta = CryptoGuard	0.0808	[0.0594, 0.102]	1.44×10^{-13}
Ferramenta = Horusec	0.1470	[0.127, 0.168]	5.71×10^{-46}
Ferramenta = SecBugs	-0.0224	[-0.0475, 0.00262]	0.0793
log(`jar_mb`)	0.0212	[0.0179, 0.0245]	1.96×10^{-36}
R^2		0.978	
N (projetos OK)		721	

Síntese. A duração é o principal preditor de energia (inclinação positiva e altamente significativa), mas a *ferramenta* e o *porte do JAR* modulam esse custo: mesmo para a duração, *Horusec* e *CryptoGuard* exibem consumo médio adicional em relação ao *CogniCrypt*, enquanto *SecBugs* concentra os maiores valores absolutos por combinar execuções mais longas e maior pico de RAM. Em outras palavras, *tempo explica quase tudo*, porém *quem* executa e *em que escala* desloca o nível de consumo.

Modelamos a energia (Wh) como função da duração (s), da ferramenta e de log(`jar_mb`), com erros robustos (HC3), para isolar o efeito marginal do tempo mantendo escala e motor fixos.

Tabela 4.14: RQ4 — Regressão de Energia (Wh) por projeto: efeito da duração e escala

Variável	**Coef.**	**IC95%**	**p-valor**
Intercepto (ref: CogniCrypt)	-0.0217	[-0.0352, -0.00828]	0.00154
Ferramenta = CryptoGuard	0.0808	[0.0594, 0.102]	1.44×10^{-13}
Ferramenta = Horusec	0.1470	[0.127 , 0.168]	5.71×10^{-46}
Ferramenta = SecBugs	-0.0224	[-0.0475, 0.00262]	0.0793
Duração (s)	0.00267	[0.00262, 0.00272]	$< 10^{-16}$
log(`jar_mb`)	0.0212	[0.0179, 0.0245]	1.96×10^{-36}

Esses achados são consistentes com a decomposição CPU/RAM (Fig. 4.4): a parcela atribuída à RAM cresce com o tempo e com a escolha do motor, e explica parte do diferencial energético entre ferramentas.

Interpretação. O coeficiente em *Duração (s)* aproxima o acréscimo médio de Wh por segundo adicional de execução, mantendo fixa a escala do projeto e a ferramenta. Os termos de *Ferramenta = X* capturam diferenças médias entre motores em relação ao *CogniCrypt*, para projetos de tamanho semelhante: coeficientes positivos indicam consumo médio adicional de energia, e coeficientes negativos indicariam consumo inferior. O termo em log(`jar_mb`) controla o efeito da escala do artefato, indicando que projetos maiores, em média, tendem a exigir mais energia mesmo para a mesma duração observada.

O ajuste global do modelo é elevado ($R^2 \approx 0{,}98$), com $N = 721$ observações (pares *projeto–ferramenta*), o que indica que a combinação de duração, ferramenta e escala explica a maior parte da variação observada na energia por projeto. Em conjunto com a Tabela 4.12 e a Figura 4.4, a regressão confirma que o custo energético segue de perto o tempo de execução, mas é modulado pela escolha da ferramenta e pelo porte do sistema analisado.

Além da regressão da Tabela 4.13, a Figura 4.5 resume a relação tempo × energia com uma amostra de projetos, destacando agrupamentos em torno da reta de regressão, e a Tabela 4.15 mostra os principais outliers em termos de resíduos.

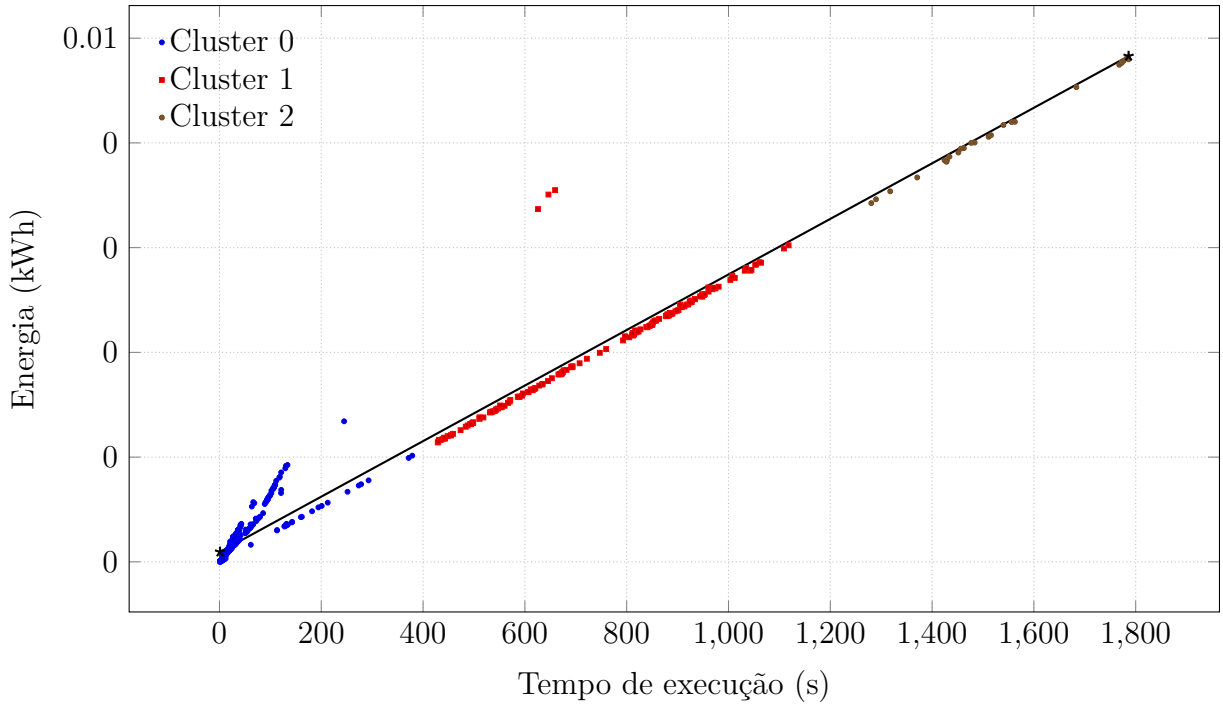

Figura 4.5: RQ4 — Tempo vs Energia com clusterização (k=3) e dados embutidos.

Tabela 4.15: RQ4 — Maiores resíduos (energia observada vs. prevista)

Projeto	**Ferramenta**	**Tempo (s)**	**Energia (kWh)**	**Resíduo**
cbo-cfe-core-2.4.1	CryptoGuard	659.39	0.003549	+0.001709
cbo-cfe-core-2.4.1	CryptoGuard	646.32	0.003507	+0.001701
cbo-cfe-core-2.4.1	CryptoGuard	625.82	0.003368	+0.001617
apf-aapf-10.16.1	Horusec	197.65	0.001470	+0.000854
apf-aapf-10.16.1	Horusec	188.46	0.001399	+0.000807
apf-aapf-10.16.1	Horusec	185.55	0.001382	+0.000798
apj-aapj-5.74.1	Horusec	177.06	0.001304	+0.000742
apj-aapj-5.74.1	Horusec	173.35	0.001277	+0.000726

Tabela 4.16: RQ4 — Parâmetros do CodeCarbon (consolidados)

Ferramenta	**Track**	**Amostragem(s)**	**Online?**	**País(ISO)**	**Observação**
CogniCrypt	offline	5	não	BR	Fator de emissão fixo; mesmas flags em todas as execuções.
CryptoGuard	offline	5	não	BR	Mesmo perfil de coleta (CC_MEASURE_SECS=5)
Horusec	offline	5	não	BR	Sem GPU nas execuções coletadas.
SecBugs	offline	5	não	BR	Status OK são tratados como casos bem-sucedidos nas consolidações.

4.5 Análise Qualitativa

Além da análise quantitativa das RQ1, RQ2, RQ3 e RQ4, observamos alguns padrões qualitativos recorrentes nos relatórios das ferramentas. Primeiro, o *SecBugs* tende a produzir conjuntos mais volumosos de achados, tanto no CamBench quanto nos sistemas reais, incluindo casos em que a inspeção manual sugere cenários de possível falso positivo (por exemplo, inicializações de bibliotecas ou caminhos de código pouco exercitados). Em contraste, *CogniCrypt* e *CryptoGuard* mostram-se mais conservadores, emitindo menos alertas, porém com maior alinhamento ao *ground truth* no benchmark sintético.

Segundo, parte dos achados em sistemas reais concentra-se em contextos de empacotamento mais complexos (por exemplo, JARs *shaded* ou artefatos que agregam múltiplos módulos). Nesses casos, diferenças na forma como cada ferramenta percorre a árvore de classes podem explicar por que certas categorias aparecem como mono-ferramenta na RQ3.

Esses elementos qualitativos serão retomados na discussão final da dissertação, ao relacionar o perfil das ferramentas com implicações práticas para a esteira de desenvolvimento da instituição.

4.6 Ameaças à Validade

Interna. Há risco de viés na coleta e na rotulagem, seja por falhas de *parsing* dos relatórios das ferramentas, seja por erros na harmonização de categorias de criptografia. Mitigamos esse risco com *scripts* reproduzíveis, consolidação padronizada dos CSVs e deduplicação sistemática por `(tool, case_id, categoria)`.

Externa. Os resultados refletem uma combinação específica de benchmark sintético (CamBench_Cap) e 211 projetos Java de uma única instituição financeira. Embora o uso de sistemas reais aumente a validade externa, a generalização para outros domínios, linguagens ou perfis de projeto deve ser feita com cautela.

Construto. Utilizamos F1 e contagens TP/FP/FN para capturar eficácia no benchmark, e tempo, energia e classes de vulnerabilidade para capturar custo e cobertura em sistemas reais. Ainda assim, esses indicadores não esgotam a noção de "qualidade" de uma ferramenta, que também envolve aspectos como usabilidade, integração à esteira e esforço de triagem de falsos positivos.

RQ1 / Parsing CamBench. A extração de `case_id` a partir de relatórios heterogêneos pode introduzir viés (por exemplo, nomes qualificados sem caminho, ou menções apenas ao `.java`). Mitigamos esse problema com regras de *fallback* conservadoras e deduplicação por `(tool, case_id, categoria)`, mas reconhecemos que diferenças sutis de formatação ainda podem afetar a contagem de TP/FP/TN/FN em casos limítrofes.

4.7 Respostas às RQs

RQ1. No CamBench_Cap, *CogniCrypt* e *Horusec* apresentam *recall* elevado, com F1 global em torno de 0,79 e 0,73, respectivamente. *CryptoGuard* obtém F1 intermediário ($\approx$ 0,55), equilibrando boa precisão com *recall* mais baixo, enquanto *SecBugs* exibe o menor F1 ($\approx$ 0,30) devido a muitos falsos negativos. As métricas comparam as saídas com o gabarito do CamBench_Cap, em que casos `TP_ref` e `FP_ref` definem, respectivamente, os exemplos positivos e negativos; a assimetria TN/FP decorre do desenho em que TN corresponde à ausência de marcação nos casos `FP_ref`.

RQ2. Em sistemas reais, o **custo computacional** das ferramentas varia de forma consistente entre motores e cresce com o tamanho do JAR. Os boxplots indicam que *CogniCrypt* é a ferramenta mais rápida, *CryptoGuard* e *Horusec* ocupam faixa intermediária e *SecBugs* concentra as maiores medianas de tempo e a maior taxa de *timeout*/erro. O resumo de recursos mostra ainda que o *SecBugs* tende a consumir mais RAM e apresentar maiores usos de CPU do que as demais ferramentas. O modelo de regressão Tempo $\sim$ $\log(\texttt{jar\_mb})$ + Ferramenta quantifica esse efeito conjunto de escala e motor de análise, alinhado à formulação da RQ2 no Cap. 3.

RQ3. Nos 211 projetos da instituição, as principais classes de vulnerabilidades de criptografia analisadas (*brokencrypto*, *insecurerandom*, *brokenhash*, *ecbmode*, *staticiv*) aparecem em um subconjunto de sistemas, mas a detecção é fortemente assimétrica entre as ferramentas: praticamente todos os projetos com achados são reportados apenas pelo *SecBugs*, com poucos casos pontuais em que *CogniCrypt* e *CryptoGuard* também sinalizam *brokencrypto*. O consenso entre motores (duas ou mais ferramentas indicando a mesma categoria no mesmo projeto) é raro, o que reforça a necessidade de triagem manual cuidadosa, sobretudo para achados mono-ferramenta.

RQ4. A energia média por projeto segue de perto a duração da execução: ferramentas mais rápidas tendem a consumir menos Wh/projeto, enquanto execuções mais longas, especialmente do *SecBugs*, apresentam maior custo energético e maior pico de RAM. A decomposição CPU/RAM indica que, em vários cenários, a RAM é o componente predominante na energia total. O modelo Energia $\sim$ Duração + Ferramenta + $\log(\texttt{jar\_mb})$ quantifica a inclinação tempo$\rightarrow$energia e mostra que a escolha da ferramenta e o porte do sistema modulam esse custo, em linha com a RQ4 do Cap. 3, que pergunta se o tempo de execução prediz o consumo de energia e as emissões por ferramenta.

Capítulo 5

Discussão, Conclusão e Trabalhos Futuros

5.1 Discussão dos Resultados

Os achados das questões de pesquisa descrevem um quadro consistente de *trade-offs* entre efetividade de detecção, custo computacional e custo energético, com implicações diretas para uma esteira *DevSecOps Verde* alinhada a metas ESG. No CamBench_Cap (RQ1), observam-se perfis distintos: motores mais seletivos tendem a equilibrar *precision* e *recall*, enquanto motores mais verbosos ampliam cobertura às custas de maior risco de falsos positivos. Ao migrar para os sistemas reais (RQ2 e RQ3), esse contraste permanece: tempos, picos de RAM e uso de CPU se separam de forma sistemática por ferramenta, e o consenso entre motores é raro, o que reforça a necessidade de triagem criteriosa e de políticas de priorização para evitar sobrecarga operacional. Em energia (RQ4), a duração explica a maior parte do consumo, mas a ferramenta e o porte do artefato deslocam o nível de Wh/gCO_2e por projeto; na prática, isso significa que decisões aparentemente "técnicas" (qual motor ativar, com quais *timeouts* e quando executar) são também decisões de orçamento energético e de sustentabilidade.

Do ponto de vista operacional, os resultados sugerem um arranjo em duas camadas para a esteira: uma camada contínua, voltada a *feedback* rápido com custo energético baixo, e uma camada periódica, executada em janelas programadas, dedicada a ampliar cobertura com triagem estruturada. O uso do CodeCarbon na CI/CD permite transformar essa diretriz em governança: Wh/projeto e gCO_2e por *job* tornam-se indicadores de orçamento, enquanto o "consenso por projeto–categoria" atua como sinal de prioridade de correção. A combinação de *precheck*, *skiplist* e *timeouts* adaptativos ao porte (controlado por proxies do JAR) tende a reduzir reexecuções desnecessárias e evitar desperdício de energia, sem degradar a segurança. Por outro lado, limitações inerentes ao cenário, ausência da árvore

de fontes nos sistemas reais, necessidade de harmonização de rótulos entre ferramentas e foco em uma única instituição, pedem cautela na generalização e, ao mesmo tempo, orientam a aplicação prática: tratar achados com consenso como candidatos preferenciais, manter triagem para categorias marcadas por um único motor e operar a esteira sob um orçamento energético explícito.

Em síntese, a leitura integrada de RQ1, RQ2, RQ3 e RQ4 indica que não há uma ferramenta universalmente melhor: há perfis complementares que devem ser orquestrados segundo metas de qualidade do produto (cobertura e precisão), metas de fluxo (tempo de resposta ao time de desenvolvimento) e metas de sustentabilidade (energia e emissões). Esse resultado reforça a visão de que segurança e sustentabilidade deixam de ser eixos separados e passam a compor, conjuntamente, os critérios de engenharia da esteira.

5.2 Conclusão e Trabalhos Futuros

Esta dissertação demonstrou, em um cenário real de instituição financeira, a viabilidade de avaliar conjuntamente acurácia, desempenho e custo energético de ferramentas SAST para usos indevidos de APIs de criptografia, combinando um *benchmark* com *ground truth* (CamBench_Cap) e 211 artefatos JAR reais. O desenho experimental integrou medição energética com CodeCarbon à CI/CD, padronizou taxonomias de categorias de criptografia entre motores e entregou *scripts* reprodutíveis de coleta e consolidação. A partir dessa base, evidenciaram-se perfis distintos por ferramenta e quantificou-se a relação tempo $\rightarrow$ energia sob controle de escala, fornecendo critérios práticos para escolha, agendamento e parametrização de varreduras na esteira.

As contribuições centrais residem (i) na avaliação integrada de eficácia e custos operacionais–energéticos em sistemas reais, (ii) no dicionário de harmonização que viabiliza comparações consistentes por projeto–categoria–ferramenta, e (iii) na tradução dos resultados em diretrizes de *DevSecOps Verde*: usar uma camada contínua e leve para feedback rápido, reservar varreduras mais pesadas para janelas periódicas com triagem e operar com orçamento explícito de Wh/gCO_2e por sprint. Como perspectivas, três frentes se mostram promissoras: aprofundar a triagem amostral estratificada dos achados para estimar falsos positivos e produzir *playbooks* de correção; sintonizar operacionalmente a esteira com *timeouts* adaptativos, incrementalidade e *caching*, reduzindo custo marginal por verificação; e, quando disponível o código-fonte, reexecutar a análise em unidades mais finas (classe/método) para reavaliar consenso entre motores e refinar a priorização. Em conjunto, estes resultados e propostas apontam para uma esteira de SAST mais efetiva, eficiente e energeticamente consciente, em consonância com as metas ESG da instituição.

Referências

[1] Sven Amann, Hoan AnhNguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transaction Son Software Engineering,Vol.45,No.12,December2019*, 2019. doi: 10.1109/TSE.2018.2827384. 1

[2] Philippe Arteau. Find security bugs - spotbugs plugin for security audits of java web applications and android applications, 2025. URL `https://find-sec-bugs.github.io/`. Acesso em: 14 abr. 2025. 11, 13

[3] Sindre Beba, Magnus Melseth Karlsen, Jingyue Li, and Bing Zhang. Critical understanding of security vulnerability detection plugin evaluation reports. *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, 2021. doi: DOI10.1109/APSEC53868.2021.00035. 15

[4] Gareth Bennett, Tracy Hall, Emily Winter, and Steve Counsell. Semgrep: Improving the limited performance of static application security testing (sast) tools. *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*, June 2024. doi: https://doi.org/10.1145/3661167.3661262. 14, 17

[5] A. Braga, R. Dahab, N. Antunes, N. Laranjeiro, and M. Vieira. Practical evaluation of static analysis tools for cryptography: Benchmarking method and case study. *IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), Toulouse, France, 2017*, 2017. doi: 10.1109/ISSRE.2017.27. 8, 15

[6] Jürgen Cito and Harald C. Gall. Crysl: Validating correct usage of cryptographic apis. *2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion*, 2016. doi: http://dx.doi.org/10.1145/2889160.2891057. 7, 8

[7] Horusec. Horusec, 2025. URL `https://github.com/ZupIT/horusec`. 11, 13

[8] Horusec. Horusec, 2025. URL `https://docs.horusec.io/docs/pt-br/overview/`. 11, 13

[9] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: Supporting developers in using cryptography. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 931, 2017. doi: 10.1109/ASE.2017.8115707. 9, 11, 12, 15

[10] Baptiste L. Lacoste, Alexandra Luccioni, Victor Schmidt, and Pierre Dandres. Codecarbon: Estimate and track carbon emissions from machine learning computing. `https://github.com/mlco2/codecarbon`, 2021. Acessado em: 18 nov. 2025. 14

[11] Jinfeng Li. Vulnerabilities mapping based on owasp-sans: A survey for static application security testing (sast). *Annals of Emerging Technologies in Computing (AETiC)*, 4(3):1–8, jul 2020. doi: 10.33166/AETiC.2020.03.001. 10

[12] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. Comparison and evaluation on static application security testing (sast) tools for java. *ESEC/FSE '23, December 3–9, San Francisco, CA, USA*, pages 4–5, 2023. doi: https://doi.org/10.1145/3611643.3616262. 9, 11, 14, 15, 17

[13] Han Liu, Sen Chen, Ruitao Feng, Chengwei Liu, Kaixuan Li, Zhengzi Xu, Liming Nie, Yang Liu, and Yixiang Chen. A comprehensive study on quality assurance tools for java. *In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States*, July 2023. doi: https://doi.org/10.1145/3597926.3598056. 10, 14, 16

[14] Gary Mcgraw. *Software Security: Building Security In*. Addison-Wesley, 2006. ISBN 0-321-35670-5. 7

[15] Carol C. Mead, Nancy R.; Woody. *Cyber Security Engineering: A Practical Approach for Systems and Software Assurance*. Addison-Wesley Professional, 2016. ISBN 978-0-134-18980-2. 6, 7

[16] Na Meng, Stefan Nagy, Danfeng (Daphne) Yao, Wenjie Zhuang, and Gustavo Arango Argoty. Secure coding practices in java: Challenges and vulnerabilities. *2018 ACM/IEEE 40th International Conference on Software Engineering*, 2018. doi: https://doi.org/10.1145/3180155.3180201. 8

[17] Tamas Kozsik Midya Alqaradaghi. Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in java code. *IEEE Access*, April 2024. doi: 10.1109/ACCESS.2024.3389955. 14, 15, 17

[18] Joydeep Mitra and Venkatesh-Prasad Ranganath. Ghera: A repository of android app vulnerability benchmarks. *PROMISE, November 8, 2017, Toronto, Canada*, 2017. doi: 10.1145/3127005.3127010. 15

[19] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. "jumping through hoops": Why do java developers struggle with cryptography apis? *2016 IEEE/ACM 38th IEEE International Conference on Software Engineering*, 2016. doi: 10.1145/2884781.2884790. 1, 8

[20] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2455–2457, 2019. doi: 10.1145/3319535.3363248. 8, 9, 11, 12, 15

[21] Anmol Ransome, James; Misra. *Core Software Security: Security at the Source*. CRC Press, 2014. ISBN 978-1-4665-6095-6. 7

[22] Irum Rauf, Marian Petre, Thein Tun, Tamara Lopez, Paul Lunn, Dirk Van Der Linden, John Towse, Helen Sharp, Mark Levine, Awais Rashid, and Bashar Nuseibeh. The case for adaptive security interventions. *ACM Transactions on Software Engineering and Methodology*, 31(1), September 2021. doi: 10.1145/3450245. 8, 9

[23] Michael Schlichtig, Anna-Katharina Wickert, Stefan Krüger, Eric Bodden, and Mira Mezini. Cambench - cryptographic api misuse detection tool benchmark suite. *arXiv preprint arXiv:2204.06447*, 2022. URL `https://arxiv.org/abs/2204.06447`. 8, 15, 24

[24] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green ai. *Communications of the ACM*, 63(12):54–63, 2020. doi: 10.1145/3381831. 10, 14, 16

[25] SpotBugs Project. Spotbugs documentation, 2025. URL `https://spotbugs.readthedocs.io/en/stable/`. Acesso em: 14 fev. 2025. 11, 13

[26] Zachary Wadhams, Clemente Izurieta, and Ann Marie Reinhold. Barriers to using static application security testing (sast) tools: A literature review. *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '24)*, pages 161–166, 2024. doi: 10.1145/3691621.3694947. 9