UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS DE CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

VITOR ANTONIO APOLINÁRIO

EXPERIMENTAÇÃO DE CARACTERÍSTICAS NO CONTEXTO DA PREDIÇÃO DE VULNERABILIDADES

CHAPECÓ
2022

VITOR ANTONIO APOLINÁRIO

EXPERIMENTACÃAO DE CARACTERÍSTICAS NO CONTEXTO DA PREDIÇÃO DE VULNERABILIDADES

Projeto de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dr. Guilherme Dal Bianco

Aprovado em: 01/ 04/ 2022 .

BANCA AVALIADORA

Prof. Dr. Guilherme Dal Bianco – UFFS

Prof. Dr. Denio Duarte - UFFS

Prof. Dr. Felipe Grando - UFFS

Experimentação de características no contexto da predição de vulnerabilidades

Vitor Antonio Apolinário[1] **, Guilherme Dal Bianco**[1]

[1]Universidade Federal da Fronteira Sul
Rodovia SC 484 Km 02, Bairro Fronteira Sul (Campus Chapecó) - 89.815-899 - Chapecó - SC

vitorapoli@gmail.com, guilherme.dalbianco@uffs.edu.br

Abstract. *The growing use of technology makes the development of secure applications essential. In contrast, the secure software development cycle is a costly task, considering the human effort required to review application code for finding vulnerabilities. In order to minimize this cost (human effort), vulnerability prediction models (VPMs) can be used by software development teams during inspection tasks. In general, VPMs are machine learning-based algorithms capable of indicate potentially vulnerable software components. Among the aspects that make the application of VPM unfeasible its low precision, which indicates the waste of human effort in the review of non-vulnerable components. Therefore, one of the obstacles in the construction of efficient VPMs is modeling meaningful features related to the vulnerabilities found. This work proposed a new feature, extracted through another domain (defect prediction) techniques. We evaluated the feature within an active learning-based VPM through a simulation on real open source projects. Our results indicates that the proposed feature looks promising in cost saving when applied to vulnerability inspection tasks.*

1 Introdução

Softwares de forma direta ou indireta fazem parte do nosso cotidiano. Em vista disso, o ciclo de desenvolvimento de *software* possui etapas minuciosas idealizadas com objetivo de mitigar defeitos e garantir a qualidade do software disponibilizado ao usuário final [1].

As vulnerabilidades, assim como os defeitos, são inconsistências presentes no *software* que quando acionadas podem causar uma falha. Embora ambas sejam semelhantes, as vulnerabilidades são ameaças que possuem uma natureza mais nociva pois podem ser exploradas por usuários mal intencionados (*i.e.*, atacantes). Geralmente a exploração de vulnerabilidades ocorre em busca de privilégios e informações importantes dos sistemas, em sua maioria, valiosas [2].

Qualquer mudança no código da aplicação pode potencialmente inserir vulnerabilidades. As mesmas podem ser identificadas e corrigidas através da revisão dos códigos que compõem o *software*. É uma tarefa custosa visto que projetos de *software* podem conter dezenas de milhares de arquivos, que por sua vez podem conter milhares de linhas de código [1]. Portanto, nos últimos anos, pesquisadores conduziram diversos experimentos utilizando aprendizado de máquina na construção e aplicação de Modelos de Predição de Vulnerabilidades (MPVs) com intuito de reduzir o esforço humano de revisão nas tarefas de inspeção de vulnerabilidade.

Yu et al. [3] identificaram que os MPVs considerados no estado da arte demonstram-se promissores na tarefa de encontrar vulnerabilidades com esforço humano reduzido, mas possuem três limitações principais: (*i*) aprendem com dados que descrevem vulnerabilidades da aplicação, no entanto, projetos em versões iniciais podem não conter esses dados; (*ii*) não permitem que o usuário escolha a taxa de revocação (percentual de vulnerabilidades encontradas) desejado ao fim da inspeção; e (*iii*) não possuem mecanismos de correção de falha humana (revisão incorreta). Nesse contexto, para contornar as limitações encontradas, os autores [3] desenvolveram um método que utiliza aprendizado ativo e características de Mineração de Texto (MT) para o treino do modelo.

Embora resultados de MPVs baseados em MT demonstrem eficiência no quesito revocação [3, 4], a precisão geralmente é baixa. Por exemplo, em [3] 95% das vulnerabilidades foram encontradas com inspeção de 20% dos arquivos, um indicativo que somente 1 em cada 22 arquivos recomendados pelo modelo é realmente vulnerável. A baixa precisão torna os métodos impraticáveis em projetos com poucos recursos de inspeção, afinal o custo é elevado devido ao desperdício de esforço voltado à revisão de arquivos que não possuem vulnerabilidades. Nesse contexto, Yu et al. [3] encontraram que defeitos registrados em tempo de execução da aplicação podem ser utilizados como conhecimento prévio a fim de melhorar a predição do método baseado em aprendizado ativo desenvolvido em seu trabalho. Porém, o registros referentes aos defeitos – obtidos em tempo de execução do programa – nem sempre estão disponíveis e podem demandar um esforço impraticável para coleta e tratamento, o que não garante que a heurística conjecturada seja aplicável em diferentes projetos de *software*. Em relação às características de MT, Li and Shao [5] afirmam que esta abordagem não é capaz de representar a estrutura e os aspectos lógicos do programa que estão fortemente relacionados às vulnerabilidades. Também, os autores salientam que a seleção das características apropriadas para o treino dos modelos é um dos grandes desafios na construção de MPVs eficientes.

O presente trabalho propõe e avalia experimentalmente um novo conjunto de características, extraídas a partir de técnicas de predição de defeitos. Um conjunto de características bem elaborado pode contribuir com a viabilização da aplicação dos MPVs nas etapas de desenvolvimento de *software*, favorecendo o desenvolvimento de aplicações mais seguras para as organizações e seus

usuários.

A heurística proposta por Nam and Kim [6] e estendida para a predição de vulnerabilidades no presente trabalho utiliza o conceito de que arquivos com métricas de *software* (*e.g.*, quantidade de linhas de código, quantidade de atributos, etc.) mais altas tendem à ser mais complexos e, consecutivamente defeituosos. De acordo com Li and Shao [5] existem similaridades entre defeitos e vulnerabilidades. Portanto, o presente trabalho explora a seguinte questão de pesquisa:

QP1. *A utilização de uma heurística desenvolvida para a predição de defeitos é capaz de diminuir o custo humano de inspeção de vulnerabilidades através de um MPV baseado em aprendizado ativo?*

O trabalho está organizado da seguinte maneira: a Seção 2 discute alguns conceitos base e trabalhos relacionados; a Seção 3 explana a metodologia utilizada; a Seção 4 apresenta o estudo de caso elaborado; especificidades dos experimentos e a discussão dos resultados são apresentados na Seção 5; a conclusão é realizada na Seção 6;

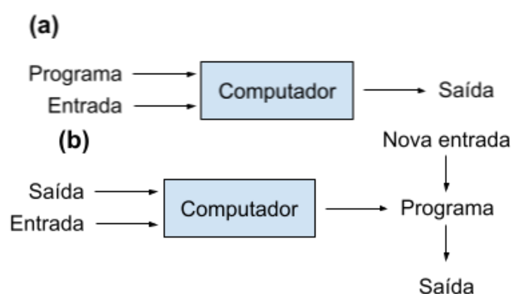
2 Referencial teórico e Trabalhos Relacionados

2.1 Aprendizado de máquina

Segundo Duarte and Ståhl [7], Aprendizado de Máquina (AM) é um sub-campo da ciência da computação que busca permitir que os computadores aprendam. Dados são entradas dos sistemas de AM. Considerando a existência de dados sobre um determinado domínio, algoritmos supervisionados de AM generalizam estes dados – também conhecidos como dados de treino – e constroem um modelo matemático que pode ser utilizados na tentativa de prever novas entradas.

Na Figura 1 é possível visualizar a diferença entre programas tradicionais (a) e aprendizado de máquina (b). Em (a) busca-se encontrar a saída correta em virtude da entrada e do programa fornecidos, enquanto em (b) busca-se um programa, a partir das entradas e saídas fornecidas, onde o programa é capaz propor saídas a partir de novas entradas.

Figura 1: Programação tradicional (a) e aprendizado de máquina (b). Adaptado de [7].

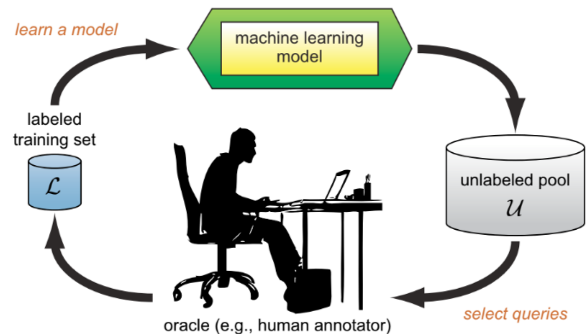


2.2 Aprendizado ativo

Aprendizado ativo é um sub-campo do aprendizado de máquina frequentemente utilizado quando a obtenção dos rótulos dos dados é uma tarefa custosa. Segundo Settles [8], a motivação da utilização de aprendizado ativo consiste na ideia de que o algoritmo pode obter melhores níveis de acurácia, com menos treino, se possuir a capacidade de escolher os dados com os quais aprende.

A Figura 2 representa um ciclo da aprendizado ativo. Inicialmente os dados de treino são compostos por uma pequena quantidade de exemplos rotulados pelo revisor. A partir destes dados, um modelo é treinado e utilizado na escolha de algumas instâncias não rotuladas. As instâncias não rotuladas selecionadas tem seu rótulo aferido pelo oráculo (*e.g.,* humano revisor) e posteriormente são adicionadas ao conjunto de dados rotulados. O ciclo se repete até que um critério de parada específico seja alcançado.

Figura 2: Ciclo de aprendizado ativo baseado em lotes. Extraído de [8].



Existem várias abordagens para a seleção das instâncias que serão rotuladas pelo revisor, como a incerta e a convicta [9]. Na primeira, são selecionadas as instâncias mais próximas da fronteira de decisão, ou seja, as instâncias em que o modelo tem mais ”dúvidas”. Na segunda estratégia são selecionadas as instâncias mais distantes da fronteira de decisão.

2.3 Modelos de predição de vulnerabilidades

Existem duas maneiras tradicionais de realizar detecção de vulnerabilidades: (i) análise estática e (ii) análise dinâmica [10]. Na análise estática o código fonte da aplicação é inspecionado em busca de componentes (arquivos, funções, blocos de código, etc) vulneráveis. A análise dinâmica, por sua vez, requer que o código seja executado e o comportamento do programa seja verificado em tempo de execução. Em ambos os casos, a causa da vulnerabilidade é identificada pelo revisor (eg., programador, analista de sistemas, etc) e corrigida.

O presente trabalho tem como alvo a discussão de técnicas de predição de vulnerabilidade através da análise estática. Nesse contexto, diversos trabalhos vêm sendo desenvolvidos com o objetivo de representar o código

fonte de uma maneira adequada, permitindo a utilização de técnicas de aprendizado de máquina como meios de análise de segurança de software, os chamados modelos de predição de vulnerabilidades (MPVs).

Morrison et al. [1] afirmam que vulnerabilidades são um subconjunto dos defeitos, tendo como diferença mais evidente a proporção: vulnerabilidades são geralmente encontradas em baixas quantidades. Portanto, modelos de predição de vulnerabilidades (MPVs) precisam lidar com problemas causados por conjuntos de dados desbalanceados.

Embora existem diferenças entre defeitos e vulnerabilidades, MPVs utilizam características parecidas com as utilizadas por modelos de predição de defeitos. Li and Shao [5] sumarizaram as principais características utilizadas em MPVs, entre elas estão:

- Métricas de *software* (MS): O vetor de características é composto por métricas que representam o *software* (e.g., contagem de linhas de código, contagem de métodos, complexidade de aninhamento). Estas métricas geralmente são capturadas durante o ciclo de desenvolvimento do *software*;
- Mineração de texto (MT): O vetor de características é criado a partir do pré-processamento do código fonte via técnicas de MT;
- Grafos de atributos de código (GAC): A estrutura do código é representada através de grafos de dependência, que são utilizados pelos modelos para prever os locais vulneráveis do código.

A Figura 3 apresenta o funcionamento genérico de um MPV supervisionado. Inicialmente, é escolhido um repositório que contém informações sobre o estado – vulnerável ou não vulnerável – de cada componente (*e.g.,* arquivo de código) do *software* que será analisado. Na etapa de treino, cada componente do *software* (Si) passa por um processo de extração dos vetores de características (Fi). Cada vetor Fi é associado a um estado (Vi) mapeado no repositório de vulnerabilidades, e um modelo classificador é treinado com as tuplas $< Fi, Vi >$. Na etapa de predição, os componentes de *software* não utilizados no treino (St) também têm suas respectivas características extraídas (Ft). As características Ft são então utilizadas como entrada para o modelo treinado anteriormente, que produz uma predição Vt (vulnerável ou não).

2.4 Predição de vulnerabilidades baseada em aprendizado supervisionado

Walden et al. [4] realizaram um comparativo entre as características de métricas de *software* e mineração de texto, utilizadas na construção de modelos de predição de vulnerabilidade. Para avaliar os resultados dos modelos foram construídos 3 conjuntos de dados que contém informações sobre vulnerabilidades contidas em versões específicas das aplicações web e *open source* Drupal[1], Moodle[2] e

Figura 3: Ilustração de um modelo de predição de vulnerabilidades supervisionado. Adaptado de [2].

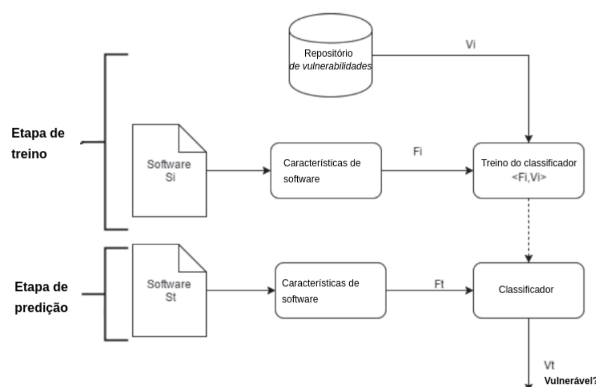


Tabela 1: Conjuntos de dados utilizados em [4].

	Arquivos vulneráveis	Total de arquivos	Arquivos vuln. (%)
Drupal	62	202	30,68
PHPMyAdmin	27	322	8,39
Moodle	24	2942	0,82

PHPMyAdmin[3]. O tamanho dos conjuntos de dados e a distribuição das vulnerabilidades são apresentados na Tabela 1.

As três aplicações selecionadas são construídas na linguagem PHP[4]. Para a obtenção das características de mineração de texto foi desenvolvido um programa que separa os *tokens* de cada arquivo de código através da função interna do PHP *token_get_all* e retorna a representação *bag-of-words* do arquivo pré-processado. O programa ignora comentários e espaços em branco, enquanto *String*'s e literais numéricos são transformados em um *token* fixo (*e.g.,*, T_STRING).

A seguir são apresentados exemplos de métricas de *software* utilizadas:

- Linhas de código: contagem das linhas de código em que são utilizados *tokens* PHP, linhas em branco ou que possuem somente comentários são desconsiderada;
- Número de funções: Contagem de funções e métodos definidos no arquivo;
- Complexidade ciclomática: Mensura a complexidade do código através da quantidade de caminhos que podem ser executados no componente analisado, neste caso, arquivos.

As métricas utilizadas para comparar os modelos foram revocação e custo de inspeção. O custo de inspeção é calculado através da Equação 1 e os acrônimos podem ser conferidos na Tabela 2. Maiores níveis de revocação sugerem que o modelo é capaz de indicar mais arquivos

[1] www.drupal.org/
[2] www.moodle.org/
[3] www.phpmyadmin.net/
[4] www.php.net/

Tabela 2: Matriz de confusão. Adaptado de [7]

		Real	
		Positivo	Negativo
Previsto	Positivo	Verd. pos. (VP)	Falso pos. (FP)
	Negativo	Falso neg. (FN)	Verd. neg. (VN)

Figura 4: Média (μ) e desvio padrão (σ) reportados para os indicadores *Recall* (revocação) e *Inspection* (custo de inspeção) [4].

	Indicator		*SW metrics (%)*	*Text mining (%)*
Drupal	Recall	μ	**76.9**	**80.5**
		σ	2.8	3.3
	Inspection	μ	**45.5**	**43.3**
		σ	2.3	2.3
PHPMyAdmin	Recall	μ	**66.3**	**73.7**
		σ	12.0	9.8
	Inspection	μ	**42.0**	**43.4**
		σ	2.7	4.6
Moodle	Recall	μ	**70.4**	**80.0**
		σ	10.8	6.1
	Inspection	μ	**32.1**	**28.7**
		σ	4.1	3.7

vulneráveis. Menores níveis de "inspeção" indicam um menor custo humano envolvido na revisão dos arquivos "vulneráveis" rotulados pelo modelo.

$$I = \frac{VP + FP}{VP + VN + FP + FN} \quad (1)$$

O método classificador escolhido foi *Random Forest* com uma configuração de 100 árvores. Os resultados são reportados na Figura 4. Segundo a análise dos autores os resultados sugerem que características de mineração de texto demonstram melhores níveis de revocação com praticamente o mesmo custo de inspeção.

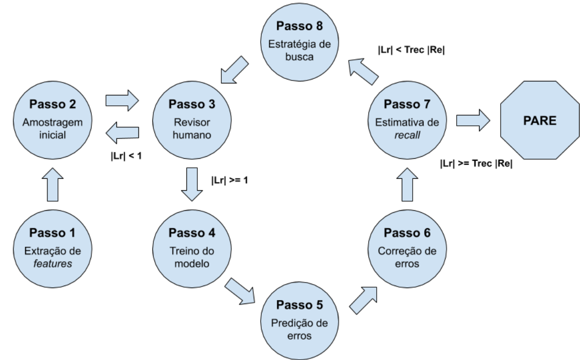
2.5 *Framework* genérico para recuperação da informação

Yu and Menzies [11] construíram um *framework* genérico de aprendizado ativo para recuperação de informação. O *framework* é ilustrado na Figura 5, onde os Passos 4 e 8 buscam melhorar a eficiência da recuperação de informação através da utilização de um modelo de AM para a recomendação. O Passo 7 busca atender o critério de parada, e os Passos 5 e 6 objetivam a correção de erros humanos. Este *framework* permite que o usuário obtenha um percentual T de informações relevantes com um esforço humano reduzido L.

2.6 Predição de vulnerabilidades baseada em aprendizado ativo

Yu et al. [3] desenvolveram uma ferramenta que utiliza aprendizado ativo com objetivo de reduzir o esforço de inspeção de vulnerabilidades de *software*. Os autores identificaram que a tarefa de encontrar mais vulnerabilidades inspecionando menos código pode ser generalizada à um problema de "*total recall*" (ou revocação total). Portanto,

Figura 5: *Framework* baseado em aprendizado ativo para problemas de revocação total



abordagens que utilizam aprendizado ativo são mais promissoras que classificadores, visto que uma vulnerabilidade não identificada pode gerar impactos significativos [3].

A partir da extensão do *framework* (Figura 5) desenvolvido em [11] para a tarefa de inspeção de vulnerabilidades foi obtida a ferramenta *HARMLESS*. Não existem restrições quanto às características utilizadas. Foram experimentadas três diferentes características: (*i*) métricas de software, (*ii*) contagem de *bugs* de código (por arquivo) e (*iii*) mineração de texto.

De acordo com os resultados apresentados em [3], os experimentos que utilizaram mineração de texto se mostraram mais eficientes. Também foi observado que a eficiência foi melhorada através da utilização das contagens de *bugs* de código como critério de escolha dos primeiros arquivos a serem revisados. Esta melhora na eficiência pode ser entendida na amostragem inicial (Passo 2 na Figura 5). Foram utilizadas duas abordagens, (*i*) amostragem aleatória, onde arquivos de código fonte são escolhidos aleatoriamente para serem analisados, e (*ii*) amostragem com conhecimento, considerando que existam características informativas sobre a propensão do arquivo a vulnerabilidades, elas podem ser utilizadas para guiar a amostragem. Theisen et al. [12], citados por Yu et al. [3], definiram que códigos fonte atrelados a maiores quantidades de problemas são mais propensos a vulnerabilidades. Portanto, foi verificado no trabalho que selecionando primeiramente os arquivos com mais problemas (*bugs*), a eficiência final foi melhorada.

Através dos resultados reportados foi possível identificar que no quesito eficiência (não são aplicados os Passos 5, 6 & 7) é possível encontrar um percentual de vulnerabilidades – 60, 70, 80, 90, 95, 99% – com respectivos 6, 8, 10, 16, 20, 34% custos de inspeção. Embora os resultados apresentados se mostrem satisfatórios existem algumas limitações, entre elas a baixa precisão apresentada. Foi estimado que cerca de 1 em cada 22 arquivos indicados possui vulnerabilidades, o que pode tornar a utilização do *HARMLESS* infactível em projetos com poucos recursos de inspeção. Os autores sugerem que a combinação de características de mineração de texto com outras métricas infor-

mativas sobre vulnerabilidades podem melhorar a predição do modelo.

2.7 Predição de defeitos não supervisionada

Nam and Kim [6] proporam soluções para predição de defeitos de *software* em projetos com dados históricos limitados, visto que a coleta e rotulação dos mesmos pode ser uma tarefa complexa e a reutilização de modelos nem sempre apresenta resultados aplicáveis.

As soluções foram nomeadas (i) CLA (*Clustering and Labeling*) e (ii) CLAMI (*Clustering, Labeling, Metric and Instance selection*) e consistem na predição de defeitos em um conjunto de dados não rotulado. Basicamente, as instâncias são agrupadas através de métodos heurísticos e então rotuladas com base na magnitude das métricas de *software* coletadas. O método (ii) possui passos adicionais que consistem na seleção de instâncias e métricas para a construção de um novo conjunto de dados, que pode então ser utilizado para a construção de diferentes classificadores.

Os métodos desenvolvidos tiveram sua performance de predição comparada com métodos supervisionados e não supervisionados. Através de experimentos realizados em sete conjuntos de dados e avaliação das métricas de revocação, precisão, *F-Score* e AUC foi possível identificar que os métodos CLA e CLAMI demonstraram melhores resultados em termos de revocação, porém apresentaram baixa precisão. Apesar da baixa precisão, os autores salientam que modelos com alta revocação são úteis em várias situações reais, e que os métodos propostos, que não demandam esforço humano, demonstram resultados equivalentes em termos de revocação, *F-score* e AUC aos métodos base, que demandam esforço humano.

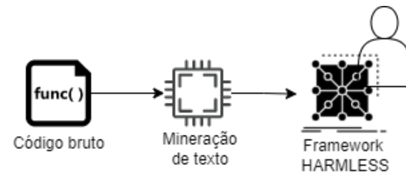
3 Metodologia

Considerando as limitações dos MPVs convencionais – apresentadas na Introdução – o presente trabalho define como método base o *framework* de predição de vulnerabilidades baseado em aprendizado ativo descrito na Seção 2.6 (HARMLESS). No trabalho [3], a combinação de características de mineração de texto com outras métricas informativas referentes a vulnerabilidades apresentou melhora no desempenho do MPV e consequentemente redução no custo de rotulação nas tarefas de predição de vulnerabilidade. Portanto, a ideia do presente trabalho é analisar experimentalmente o desempenho do método base quando utilizado em conjunto com técnicas de outra área (predição de defeitos).

3.1 O problema da "partida fria"

Conforme descrito na Sessão 2.6, o HARMLESS [3] é um método iterativo para predição de vulnerabilidades. A partir do primeiro elemento vulnerável rotulado (Passo 3 - Figura 5), um modelo é treinado (Passo 4) e utilizado para recomendar (Passo 8) prováveis componentes vulneráveis para a próxima rodada de revisão. Porém, enquanto nenhum componente vulnerável é encontrado o algoritmo "recomenda" arquivos randomicamente para o revisor, dada a ausência de conhecimento sobre o padrão das vulnerabilidades presentes no conjunto. No contexto da Recuperação de Informação, esse problema é denominado "partida fria".

Figura 6: Método base, proposto por Yu et al. [3]



Para amenizar tal problema, os autores desenvolveram uma abordagem que consiste em utilizar conhecimento prévio sobre os defeitos[5] presentes na aplicação. Para cada arquivo, são computadas a quantidades de vezes em que ocorreu um registro de falha em tempo de execução. Posteriormente, a contagem de falhas é utilizada como característica adicional. Também, a contagem de defeitos é utilizada durante a amostragem inicial: enquanto o primeiro arquivo vulnerável não é encontrado, os arquivos são recomendados ao revisor em ordem descendente de acordo com a contagem de falhas. Após o primeiro componente vulnerável encontrado, o treino do modelo (Passo 4) é realizado com a combinação das *features* de texto e contagem de falhas (descrita acima) e os componentes a serem revisados na próxima rodada são recomendados pelo modelo treinado.

Tal heurística parte do princípio conjecturado por Theisen et al. [12], que afirma que arquivos com maiores quantidades de defeitos são mais propensos a conter vulnerabilidades. Embora os resultados demonstrem uma melhora da predição nos estágios iniciais – e consequentemente uma redução de custo ao final da execução – a técnica requer a disponibilidade de registros sobre a ocorrência de erros em tempo de execução. Porém, tais indícios podem nem sempre estar disponíveis e demandar esforço (refatoração, engenharia de dados, etc.) para obtenção. Nesse contexto, o presente trabalho explora técnicas de outra área em busca de obter uma característica informativa e capaz de melhorar a predição do modelo nos estágios iniciais, sem a necessidade de dados obtidos durante o tempo de execução da aplicação.

3.2 Buscando características informativas

O método proposto (HARMLESS) não possui restrições quanto as características utilizadas. Porém, em [3] foi constatado através dos experimentos que quando utilizadas características de mineração de texto (MT) menores custos de rotulação são alcançados, indicando o aumento de eficiência. Considerando o problema de partida fria e a premissa de que arquivos defeituosos tendem a possuir vulnerabilidades [4], foi iniciada a busca por métodos capazes de fornecer informações referentes aos defeitos presentes nos arquivos e utilizá-los a fim de guiar o revisor durante o processo de amostragem inicial (Passo 2 - Figura 5).

O trabalho realizado por Nam and Kim [6] propõe

[5] vulnerabilidades pertencem ao subconjunto dos defeitos

Figura 7: Método de clusterização para predição de defeitos. Adaptado de [6].

Dataset não rotulado

Instâncias	X_1	X_2	X_3	X_4	X_5	X_6	X_7	Rótulo	K
A	**3**	1	**3**	0	5	1	**9**	**?**	**3**
B	1	1	2	0	**7**	**3**	8	**?**	**2**
C	**2**	**3**	2	**5**	5	**2**	1	**?**	**4**
D	0	0	**8**	1	0	1	**9**	**?**	**2**
E	1	0	2	**5**	**6**	**10**	8	**?**	**3**
F	1	**4**	1	1	**7**	1	1	**?**	**2**
G	1	0	1	0	0	1	7	**?**	**0**
Mediana	1	1	2	1	5	1	8		

Cluster, K=4	Cluster, K=3	Cluster, K=2	Cluster, K=0
C	A, E	B, D, F	G

componentes defeituosos — componentes não defeituosos

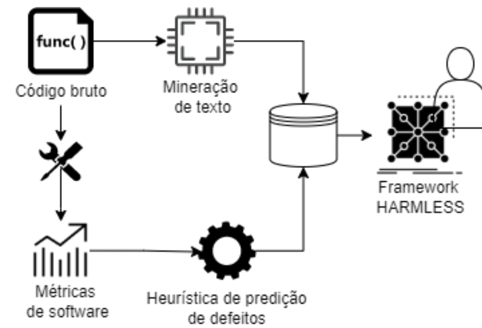
o método denominado *Clustering and Labeling* (CLA), capaz de indicar arquivos defeituosos com base na magnitude das métricas de software. Embora o método demande informações das métricas de software (podem ser coletadas de maneira estática), o mesmo não necessita de rótulos e de esforço humano. A Figura 7 apresenta visualmente o seu funcionamento e será explicada a seguir.

Dado o conjunto de componentes de software não rotulados que possui instâncias A-G e características (métricas de software) X_1-X_7, são identificadas todas as métricas que extrapolam um determinado limite numérico. Neste caso, para cada métrica X_i o limite é a mediana da métrica X_i observado em todas as instâncias (e.g. para X_1 $\{0,1,1,\mathbf{1},1,2,3\}$ o limite é **1**). Posteriormente, para cada instância, é realizada uma contagem de métricas que superam o valor mediano encontrado. Na Figura 7, todas as métricas que superam o limite imposto estão em negrito. Para cada instância I_i, a contagem de métricas maiores que o limite definido é denominada K_i e utilizada para agrupar os componentes. Essa etapa é denominada *Clustering* (agrupamento).

Posteriormente, na etapa de *Labeling* (marcação ou aferição de rótulos), os *clusters* são divididos em duas partes, onde todas as instâncias presentes nos grupos com maiores valores de K são classificados como defeituosos e as instâncias dos grupos com menores K são consideradas "limpas", ou seja, não defeituosas. Na Figura 7 as instâncias presentes nos grupos C (K=4), A e E (K=3) são classificadas como defeituosas, enquanto as instâncias contidas nos grupos B, D e F (K=2) e G (K=0) são classificadas como limpas.

O trabalho [6] propõe ainda uma variação (CLAMI) que possui alguns passos adicionais. Porém, segundo os autores, o primeiro método (CLA) é mais simples e demonstra um desempenho semelhante na tarefa de predição de defeitos.

Figura 8: Método proposto no presente trabalho, combina o *framework* HARMLESS [3] e a heurística de predição de defeitos [6] apresentada na Seção 3.2.



Portanto, considerando que a heurística proposta por Nam and Kim [6] apresentou eficiência na tarefa de predição de defeitos e não demanda esforço humano ou registros de tempo de execução, ela será utilizada como objeto do presente trabalho.

3.3 Proposta

Este trabalho verifica a aplicabilidade de técnicas de outro domínio (predição de defeitos) à tarefa de predição de vulnerabilidades, através do método de aprendizagem ativa HARMLESS. Para isso, a técnica descrita na Seção 3.2 foi utilizada para extrair a "contagem de violações" de cada componente de software contido no projeto alvo. Posteriormente, a contagem foi utilizada como conhecimento prévio durante a amostragem inicial e como *feature* adicional para o treino do modelo, Passos 2 e 4 na Figura 5. Em relação à amostragem inicial proposta no trabalho original [3], duas estratégias foram utilizadas (Passo 2 na Figura 5):

1. **Amostragem aleatória**: os arquivos são selecionados aleatoriamente até que seja encontrado ao menos um arquivo vulnerável;
2. **Amostragem com conhecimento**: informações referentes ao domínio do software são utilizadas com o objetivo de encontrar o primeiro arquivo vulnerável antecipadamente.

Logo, o diferencial do método proposto neste trabalho reside nos seguintes aspectos:

- Durante a **amostragem com conhecimento** a "contagem de violações" é utilizada como conhecimento do domínio, portanto o HARMLESS seleciona os arquivos de acordo com a ordem descendente da contagem de violações;
- A contagem de violações contribui também como uma *feature* adicional durante os estágios de treino e predição modelo de aprendizagem.

4 Estudo de caso

Embora o HARMLESS seja um método desenvolvido para guiar revisores humanos na busca por vulnerabilidades, torna-se inviável experimentar tal tarefa com diferentes parâmetros e conjuntos de dados. Portanto, a simulação proposta em [3] foi estendida no presente trabalho e apresentada nesta Seção.

4.1 Conjuntos de dados

No estudo original [3] o método foi testado no conjunto de dados do Mozilla Firefox, que contém 28.750 arquivos das linguagens C e C++, dos quais 271 possuem vulnerabilidades. Portanto, a mesma base foi utilizada a fim de permitir comparabilidade entre as características *baseline* e as desenvolvidas no presente trabalho.

Com o objetivo de permitir mais observações referentes à eficiência do método proposto, foram selecionados 3 conjuntos de dados construídos em um estudo [4] de predição de vulnerabilidades. Os conjuntos são constituídos de dados das aplicações Drupal, PHPMyAdmin e Moodle. Tais aplicações foram desenvolvidas na linguagem de programação *web* PHP e a densidade de vulnerabilidades são apresentadas na Tabela 1.

4.2 Variáveis dependentes

Para os 4 conjuntos selecionados cada instância representa um arquivo de código, mapeado como vulnerável ou não vulnerável. O conjunto de dados do Mozilla possui informações referente ao tipo da vulnerabilidade presente no arquivo. Porém, tal informação não será utilizada pois o método busca classificar um arquivo como vulnerável ou não, independente do tipo da vulnerabilidade.

4.3 Variáveis independentes

Esta seção descreve brevemente as *features* experimentadas, assim como suas respectivas técnicas de extração.

4.3.1 Métricas de software

Os conjuntos selecionados contam com métricas extraídas no formato adequado para a realização dos experimentos. É importante frisar que no atual trabalho as métricas de software não foram utilizadas diretamente como *features* do modelo. Sua utilização é descrita na Subseção 4.3.2.

4.3.2 Contagem de violações

As contagens de violações foram coletadas através de um programa desenvolvido por Nam and Kim [6] e adaptado para o presente trabalho. Para cada conjunto de dados, o programa consome arquivos em formato específico (ARFF) e foi alterado para contabilizar a contagem de violações por arquivo.

Para cada um dos conjuntos, Drupal, Moodle e PHPMyAdmin, a seguinte sequência de passos é realizada:

1. O programa carrega dois arquivos por *dataset*: um contendo somente o código "bruto" de cada arquivo, enquanto o outro contém as métricas de cada instância;
2. As métricas de cada arquivo são utilizadas para calcular as contagens de violações, através da heurística descrita na Seção 3.2;
3. As violações contabilizadas para cada arquivo são agrupadas com seu respectivo código fonte;
4. Por fim, um arquivo CSV é gerado com o código fonte (não estruturado) e contagem de violações, para cada instância.

Como o *dataset* do Mozilla Firefox foi idealizado em outro estudo, a base original[6] possui um formato diferente e requeriu algumas etapas adicionais.

1. As métricas de software são transformadas do formato CSV para o formato ARFF através de um *script* escrito em Python;
2. O programa (CLA) carrega o arquivo gerado no passo anterior, que contém as métricas de cada instância e calcula as contagens de violações através da heurística descrita na Seção 3.2;
3. Por fim, o programa gera um arquivo contendo somente a contagem de violações para cada instância;
4. Posteriormente, um segundo *script* Python é utilizado para carregar o arquivo gerado no passo anterior, assim como o arquivo que contém o código fonte bruto do Mozilla. O *script* então unifica o código fonte de cada arquivo com sua respectiva contagem de violação e gera um arquivo no formato CSV.

4.3.3 Características de mineração de texto (*baseline*)

Todos os conjuntos de dados gerados conforme descrito na Subseção 4.3.2 passaram pelo mesmo processo de mineração de texto proposto por Yu et al. [3]. Para cada base, o código fonte de cada instância passa pela etapa de separação de *tokens*, sem remoção de *stop-words*. Posteriormente, são selecionados os 4000 *tokens* com maiores pontuações *tf-idf* (obtido através das somas de cada pontuação para cada *token*). Após, uma matriz termo-frequência é construída com base nos 4000 termos selecionados anteriormente. Cada linha da matriz foi normalizada via norma L2.

4.4 Simulação

Yu et al. [3] projetaram um cenário para simular a tarefa de inspeção de vulnerabilidades e verificar a eficiência do método proposto. A mesma simulação foi implementada nesse trabalho.

Embora o MPV *HARMLESS* seja uma extensão do *framework* descrito na Seção 2.5, os experimentos que verificam a eficácia de inspeção desconsideram as etapas de correção de erro (Passos 5 & 6 na Figura 5) e a utilização do *recall* estimado como critério de parada (Passo 7 na Figura 5). A simulação é ilustrada na Figura 9 e utiliza as seguintes configurações:

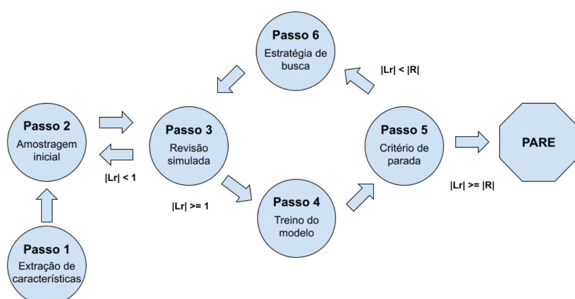
- **Passo 1 (Extração de características):** E é o conjunto dos componentes do software (arquivos, funções, etc), R contém todos os componentes vuneráveis, $L \leftarrow \emptyset$ é o conjunto dos componentes revisados, $L_R \leftarrow \emptyset$ é o conjunto dos componentes vulneráveis já inspecionados. As características são extraídas de cada arquivo em E.
- **Passo 2 (Amostragem inicial):** Os arquivos são escolhidos em lotes e recomendados para o revisor. No trabalho original são revisados

[6] https://github.com/ai-se/Mozilla_Firefox_Vulnerability_Data

100 arquivos por vez. Segundo os autores esse número foi escolhido com o objetivo de acelerar a simulação, porém são recomendados lotes menores para aplicações reais [3]. Para o *Mozilla* o mesmo tamanho de lote será utilizado (100 arquivos). Porém, para os conjuntos Drupal, Moodle e PHPMyAdmin será utilizado um lote de 10 arquivos, a fim de permitir mais rodadas de inspeção. Duas abordagens são utilizadas para a escolha dos arquivos iniciais, que são: amostragem aleatória e amostragem com conhecimento (descritas na Seção 3.3).

- **Passo 3 (Revisão simulada):** Os rótulos **reais** dos arquivo selecionados no passo anterior são verificados no conjunto de dados e aplicados, a fim de simular a inspeção realizada por um humano. Os arquivos vulneráveis identificados são adicionados ao conjunto L_R. Se ao menos um arquivo vulnerável for encontrado ($|L_R| \geq 1$) o algoritmo segue para o **Passo 4**, senão, retorna ao **Passo 2**.
- **Passo 4 (Treino do modelo):** Um modelo *support vector machine* (SVM) é treinado com base nos arquivos rotulados (conjunto L). O classificador escolhido e seus parâmetros são os mesmos utilizados no trabalho original [3].
- **Passo 5 (Critério de parada):** O critério de parada utilizado é a revocação pré-definida (*e.g.* 95%, 90%, 60% etc.). O algoritmo tem visão do total de vulnerabilidades existentes $|R|$, portanto, termina a simulação quando a revocação pré-definida é alcançada $|L_R| \geq |R|$.
- **Passo 6 (Estratégia de busca):** O classificador treinado no **Passo 4** é utilizado para selecionar o próximo lote de arquivos indicados ao revisor. Existem duas abordagens (descritas na Seção 2.2) utilizadas para a escolha: Enquanto menos de 10 arquivos vuneráveis foram encontrados $|L_R| < 10$ é utilizada amostragem incerta, caso contrário $|L_R| \geq 10$, o algoritmo utiliza amostragem convicta.

Figura 9: Cenário da simulação



5 Experimentos e Resultados

Esta seção descreve os experimentos realizados a fim de avaliar a eficiência do método proposto para a tarefa de predição de vulnerabilidades. As simulações ocorreram de acordo com o estudo de caso apresentado na Seção 4 e os resultados foram utilizados para responder questão de pesquisa (QP1).

5.1 Métricas

Um dos objetivos do *HARMLESS* é minimizar o custo de inspeção de vulnerabilidades no atingimento de altos níveis de revocação [3]. Nesse contexto, a revocação, ou *recall* (Equação 2), informa sobre o percentual de arquivos vulneráveis encontrados ao final do processo, enquanto o custo (Equação 3) informa o percentual de componentes revisados ao fim da inspeção.

$$recall = \frac{\textit{\# de componentes vuln. encontrados}}{\textit{\# de componentes vulneráveis de existentes}} \tag{2}$$

$$custo = \frac{\textit{\# de componentes inspecionados}}{\textit{\# de componentes existentes}} \tag{3}$$

Portanto, o método que apresentar menor custo para alcançar um determinado nível de revocação será considerado o mais eficiente.

5.2 Desenho da simulação

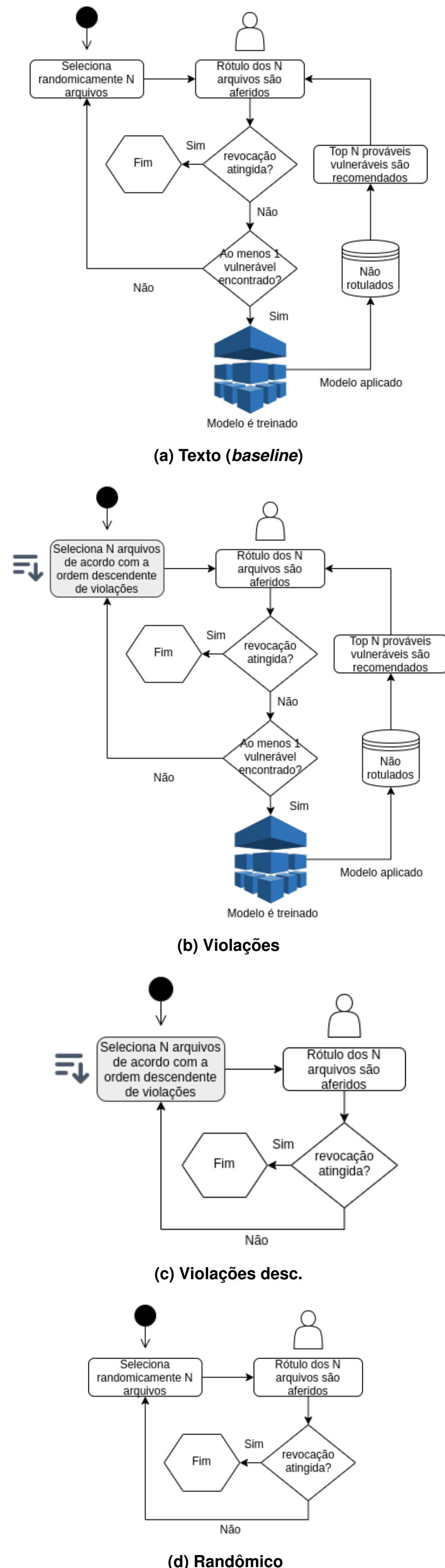
Assim como no estudo original [3], o custo para atingir diferentes níveis de revocação será utilizado para comparar a eficácia de inspeção de vulnerabilidades de cada método. Inicialmente foram testadas as seguintes características em conjunto com um classificador supervisionado:

- **Texto (*baseline*):** modelo classificador treinado na granularidade[7] de arquivo. O código fonte é tratado como texto não estruturado e passa pela extração de características conforme descrito na Seção 4.3.3. Ilustrado na Figura 10a;
- **Violações:** modelo classificador treinado na granularidade de arquivo. Utiliza a combinação de características de mineração de texto e contagem de violações (descrita na Seção 4.3.2). Durante a amostragem inicial – bloco cinza na Figura 10b – a contagem de violações é utilizada como **conhecimento prévio**, onde arquivos com mais violações serão selecionados primeiro.

A abordagem proposta por Yu et al. [3] foi estendida a fim de simular cenários onde a inspeção de vulnerabilidades é realizada sem a utilização de algoritmos de aprendizado de máquina. Logo, as seguintes heurísticas foram testadas:

- **Violações descendente**: os arquivos são selecionados para inspeção de acordo com a ordem descendente de violações. É importante salientar que nesse método nenhum modelo é treinado para indicação de arquivos vulneráveis;
- **Randômico**: os arquivos são selecionados de maneira aleatória para inspeção. Este cenário simula a inspeção de vulnerabilidades sem a utilização de ferramentas de apoio.

Figura 10: Detalhes da simulação executada para cada método.



(a) Texto (*baseline*)

(b) Violações

(c) Violações desc.

(d) Randômico

Método	Preditor	Meios de aprendizado: Carac. de min. de texto (MT)	Carac. cont. de violações (CLA)	Humano revisor
Texto (baseline)	SVM	X		X
Violações	SVM	X	X	X
Violações desc.	n/a		X	
Randômico	n/a			

Tabela 3: Sumário dos métodos de predição de vulnerabilidades experimentados

O sumário dos métodos experimentados pode ser conferido na Tabela 3. Os métodos que "aprendem com humano revisor" (*i.e.*, possuem a última coluna da tabela assinalada) utilizam o *framework* HARMLESS [3]. A Figura 10 ilustra o a simulação realizada para cada método.

5.3 Resultado experimental

A Tabela 4 apresenta o custo obtido através de 30 execuções de simulação para cada método. Diferentes níveis de revocação alvo foram testados. Assim como no trabalho original [3], os resultados são reportados em termos de mediana e a dispersão através da distância interquartil. Um método é considerado melhor que o outro se apresentar menor custo para atingir o mesmo nível de revocação [3]. Os melhores métodos para cada configuração (projeto e revocação alvo) estão em negrito. É possível observar que:

- Os métodos **Violações** e **Texto** não apresentam redução expressiva de custo nos conjuntos Drupal, Moodle e PHPMyAdmin, mas ainda sim se mostram mais eficientes do que o método **Randômico** na maioria das configurações;
- O método **Violações desc.** apresenta efetividade nos estágios iniciais de predição, porém, não é capaz de encontrar níveis mais altos de vulnerabilidades nos conjuntos Drupal e PHPMyAdmin;

Considerando os métodos supervisionados, é possível observar:

- O método **Violações** apresenta melhor eficiência em 17 dos 24 cenários (*i.e.*, combinações projeto-revocação);
- Para o conjunto Drupal o método **Violações** desempenha melhor que **Texto** para os níveis de revocação 60%, 80% e 85%, porém o mesmo custo é obtido nos níveis 70%, 90% e 95%;
- O método **Violações** supera o **Texto** no conjunto Moodle nos estágios iniciais (revocação $< 85\%$) e na revocação $= 95\%$;
- No conjunto PHPMyAdmin o método **Violações** supera o método **Texto** para todos os níveis de revocação $> 60\%$;
- Para o projeto Mozilla o método **Violações** supera o **Texto** em todos os níveis de revocação $< 95\%$;

A Figura 11 reporta a curva do custo de rotulação necessário para obtenção de 95% dos arquivos vulneráveis (*i.e.*, revocação $= 95\%$). Curvas mais acentuadas indicam que o método é capaz de atingir o mesmo nível de

[7] unidade inspecionada (*e.g.*, função, linha de código, etc.)

		Revocação alvo					
		0.6	0.7	0.8	0.85	0.9	0.95
Projeto	Método	Custo para atingir a revocação alvo					
Drupal	Violações	**34 (0)***	**44 (4)**	**49 (0)***	**59 (0)***	79 (0)	**84 (0)**
	Texto (baseline)	39 (4)	**44 (4)**	54 (9)	64 (8)	79 (4)	**84 (4)**
	Violações desc.	**34 (0)**	**44 (0)**	**49 (0)**	**59 (0)**	**64 (0)**	n/a
	Randômico	61 (4)	69 (4)	79 (4)	84 (4)	89 (9)	94 (0)
Moodle	Violações	23 (0)*	**24 (0)***	**27 (0)***	72 (0)	79 (0)	89 (0)*
	Texto (baseline)	31 (10)	36 (8)	48 (8)	50 (20)*	56 (18)*	94 (4)
	Violações desc.	**16 (0)**	35 (0)	38 (0)	**38 (0)**	**48 (0)**	**52 (0)**
	Randômico	56 (8)	63 (9)	75 (11)	79 (8)	85 (9)	88 (7)
PHPMyAdmin	Violações	37 (3)	52 (9)*	55 (0)*	**59 (0)***	**62 (0)***	**65 (0)***
	Texto (baseline)	37 (11)	60 (18)	71 (8)	76 (8)	86 (6)	96 (6)
	Violações desc.	**31 (0)**	**34 (0)**	**43 (0)**	n/a	n/a	n/a
	Randômico	55 (14)	65 (9)	77 (6)	80 (5)	86 (8)	91 (3)
Mozilla	Violações	**5 (0)***	**7 (0)***	**10 (0)***	**12 (0)***	**16 (0)***	24 (0)
	Texto (baseline)	6 (0)	8 (0)	11 (0)	14 (0)	18 (0)	**23 (1)***
	Violações desc.	35 (0)	41 (0)	46 (0)	47 (0)	49 (0)	54 (0)
	Randômico	60 (2)	69 (3)	79 (2)	85 (2)	89 (1)	94 (1)

Tabela 4: A tabela acima mostra o custo (*i.e.*, percentual de arquivos revisados) necessário para atingir diferentes níveis de revocação pré-definidos. Os valores reportam a mediana de 30 simulações e a dispersão (valor entre parênteses) foi obtida através da Distância Interquartil ($DI = Q3 - Q1$). Para o método "Violações desc." algumas células possuem o valor "n/a" indicando que o método não foi capaz de alcançar a revocação pré-definida. As células que contém um "*" indicam qual método supervisionado (Violações ou Texto) apresenta custo inferior – em termos de mediana – para determinada configuração (*i.e.*, projeto e revocação alvo).

revocação com um custo inferior. O custo é reportado em termos de mediana. As áreas sombreadas representam a dispersão do respectivo método e os limites inferiores e superiores são obtidos a partir do primeiro e terceiro quartis, respectivamente. É possível observar que:

- O método heurístico **Violações desc.** desempenha bem nos estágios iniciais e não possui dispersão porém não é capaz de alcançar 95% de revocação em projetos específicos (Drupal, PHPMyAdmin);
- O método **Texto** apresenta maior dispersão em comparação ao método **Violações**;
- O método **Violações** apresenta uma curva mais acentuada que o método **Texto** nos conjuntos Drupal, Moodle e PHPMyAdmin;

Existem alguns aspectos capazes de explicar o desempenho heterogêneo dos métodos para cada projeto. Os conjuntos Drupal e PHPMyAdmin possuem tamanhos parecidos (202 e 322 arquivos, respectivamente), porém, a diferença na densidade de vulnerabilidades é significativa 30% no Drupal e 8,4% para o PHPMyAdmin. Os conjuntos Moodle e Mozilla possuem densidade de vulnerabilidade semelhantes (0,82% e 0,94%, respectivamente), porém, a quantidade total de arquivos é desproporcional (28750 arquivos no Mozilla *versus* 2942 no Moodle). Também, é necessário salientar que existem diferença nas tecnologias empregadas em cada projeto. Os conjuntos Drupal, Moodle, PHPMyAdmin são *softwares* desenvolvidos na linguagem *PHP* e executados em servidores *web*. Por outro lado, o projeto Mozilla é uma aplicação *desktop* construída na linguagem *C++*. A diferença tecnológica implica que distintas classes de vulnerabilidades podem estar presentes em cada projeto, uma potencial justificativa para a heterogeneidade observada nos resultados, afinal nem todos os tipos de vulnerabilidades são detectáveis através de técnicas de aprendizado de máquina [5].

Outro fator a ser discutido é a dispersão (valores entre parênteses na Tabela 4 e áreas sombreadas nos gráficos da Figura 11) das 30 simulações para cada configuração (projeto, método e revocação alvo). Sobre os métodos supervisionados, é possível observar que o método **Violações** apresenta maior estabilidade, dado que a amostragem inicial é guiada pela contagem descendente de violações, ou seja, um processo constante. Já o método base (**Texto**) utiliza amostragem aleatória até o momento em que 1 arquivo vulnerável é encontrado, o que corrobora com a maior dispersão observada nos custos de inspeção. Discutindo os métodos heurísticos, o método **Violações descendente** não apresenta variação, afinal trata-se de um método constante que escolhe os arquivos de acordo com a ordem descendente de violações. O método **Randômico** seleciona os arquivos aleatoriamente para inspeção, o que justifica a dispersão observada.

É possível observar que o método heurístico **Violações descendente** possui células preenchidas com "n/a", indicando a impossibilidade de alcançar o nível de revocação especificado. A justificativa parte da abordagem proposta por Yu et al. [3] e estendida para a contagem de violações: a inspeção simulada é finalizada quando os arquivos não rotulados restantes não possuem violações (*i.e.*, zero violações). Por consequência, arquivos vulneráveis que não violam nenhuma métrica de *software*

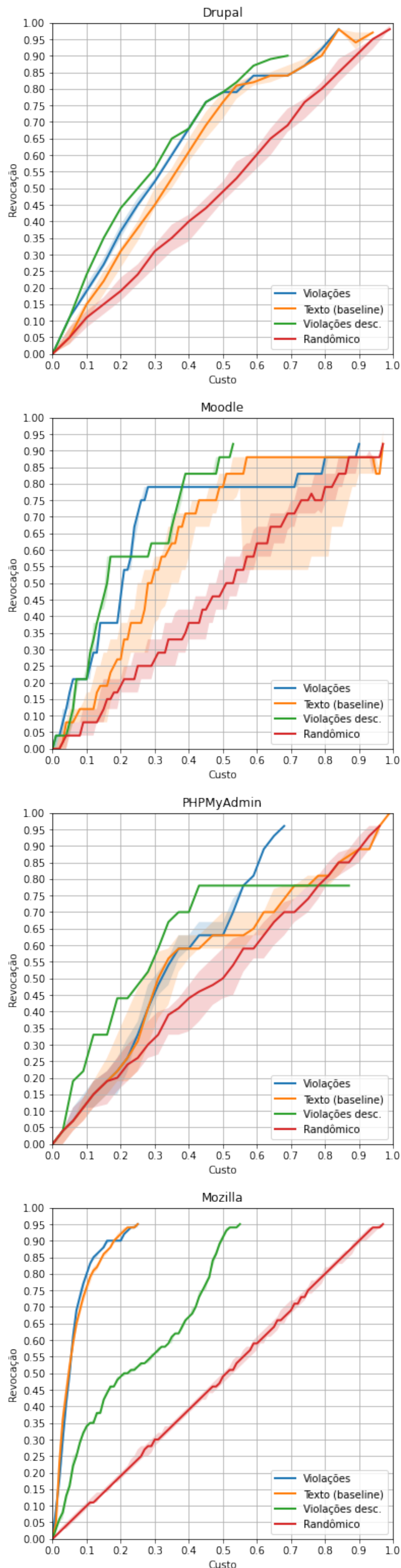


Figura 11: Revocação (eixo y) e custo de rotulação (eixo x) comparando o desempenho de diferentes métodos no alcance de 95% de revocação. As áreas sombreadas representam a variação capturada para 30 repetições.

não são encontrados, impedindo que a revocação alvo seja alcançada. Embora a **amostragem com conhecimento** seja uma heurística simples que não utiliza técnicas de aprendizado de máquina, os experimentos demonstram que o método é promissor para a predição de vulnerabilidades quando aliado às contagens de violações. Por exemplo, na Figura 11 o método **Violações desc.** apresenta curvas acentuadas nos estágios iniciais (*i.e.*, custo $< 50\%$) de predição. Também, é possível identificar na Tabela 4 que o método **Violações desc.** mostra-se a melhor abordagem em diferentes cenários, mesmo comparada com métodos que utilizam aprendizado ativo (**Texto** e **Violações**).

Com base nos resultados experimentais e observações realizadas, buscamos responder a QP1:

QP1. *A utilização de uma heurística desenvolvida para a predição de defeitos é capaz de diminuir o custo humano de inspeção de vulnerabilidades através de um MPV baseado em aprendizado ativo?*

Sim. Através da simulação realizada para diferentes projetos de *software* (Drupal, Moodle, PHPMyAdmin e Mozilla) e múltiplos níveis de revocação alvo (60, 70, 80, 90, 95%) o método proposto (**Violações**) apresenta menores custos de inspeção em relação ao método base (**Texto**) em diversas configurações. Considerando a revocação alvo de 95% e os conjuntos Moodle e PHPMyAdmin, o método **Violações** apresentou redução de 5% ($2942 * 0.05 \approx 147$ arquivos) e 31% ($322 * 0.31 \approx 99$ arquivos) nos custos de inspeção em relação ao método base.

6 Conclusão e Trabalhos futuros

A redução de custo nas tarefas de inspeção de vulnerabilidades é crucial para garantir aplicações mais seguras. Os Modelos de Predição de Vulnerabilidades (MPVs) são métodos que utilizam aprendizado de máquina em busca de reduzir o custo humano nas tarefas de mitigação de vulnerabilidades em projetos de *software*. O presente trabalho propôs a utilização de uma característica de outro domínio para a predição de vulnerabilidades. A contagem de violações foi extraída a partir de heurísticas de predição de defeitos e apresentou-se promissora na tarefa de predição de vulnerabilidades.

A abordagem foi testada em diferentes conjuntos de dados através de experimentos elaborados com intuíto de simular a tarefa de inspeção de vulnerabilidades e capturar a eficiência do MPV para cada conjunto de características. Foi possível observar que a característica proposta é capaz de diminuir os custos de inspeção em diferentes cenários. Os conjuntos de dados, o código utilizado para extração de características e execução da simulação estão disponível no Github[8].

Trabalhos futuros podem incluir uma análise estatística acerca da informatividade das características propostas (contagem de violações) em relação às vulnerabilidades. Também, futuros estudos podem analisar experimentalmente a eficiência das contagens de violações

[8]https://github.com/vulnerability-research

quando utilizadas como características em MPVs convencionais (sem a utilização de aprendizado ativo).

Referências

[1] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pages 1–9, 2015.

[2] PK Shamal, K Rahamathulla, and Ali Akbar. A study on software vulnerability prediction model. In *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pages 703–706. IEEE, 2017.

[3] Zhe Yu, Christopher Theisen, Laurie Williams, and Tim Menzies. Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering*, 2019.

[4] James Walden, Jeff Stuckman, and Riccardo Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *2014 IEEE 25th international symposium on software reliability engineering*, pages 23–33. IEEE, 2014.

[5] ZhanJun Li and Yan Shao. A survey of feature selection for vulnerability prediction using feature-based machine learning. In *Proceedings of the 2019 11th International Conference on Machine Learning and Computing*, pages 36–42, 2019.

[6] Jaechang Nam and Sunghun Kim. Clami: Defect prediction on unlabeled datasets (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 452–463. IEEE, 2015.

[7] Denio Duarte and Niclas Ståhl. Machine learning: a concise overview. *Data Science in Practice*, pages 27–58, 2019.

[8] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[9] Zhe Yu, Nicholas A Kraft, and Tim Menzies. Finding better active learners for faster literature reviews. *Empirical Software Engineering*, 23(6):3161–3186, 2018.

[10] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. Vulnerability prediction from source code using machine learning. *IEEE Access*, 8:150672–150684, 2020.

[11] Zhe Yu and Tim Menzies. Fast2: An intelligent assistant for finding relevant papers. *Expert Systems with Applications*, 120:57–71, 2019.

[12] Christopher Theisen, Kim Herzig, Patrick Morrison, Brendan Murphy, and Laurie Williams. Approximating attack surfaces with stack traces. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 199–208. IEEE, 2015.