

# Predicting Vulnerable Software Components via Text Mining

Luís Felipe Ramos Ferreira  
Igor Lacerda Faria da Silva

## **Conceitos principais**

# Mineração de dados/texto

- ▶ Técnica de aprendizado de máquina
- ▶ Extração de informação e características
- ▶ Descoberta de padrões e subgrupos
- ▶ Modelos de classificação

# Vulnerabilidade de componentes de Software

- ▶ Fraqueza no sistema
  - ▶ Suscetibilidade a ser explorada por ameaças ou problemas aleatórios
  - ▶ Sistema vulnerável: existe uma oportunidade para uma ameaça quebrar sua segurança
- ▶ Causas
  - ▶ Erros
  - ▶ Falhas no projeto de software
- ▶ Exemplos
  - ▶ Códigos com vazamento de memória

**Projeto de pesquisa realizado**

# Introdução

- ▶ Utilização de técnicas de mineração de textos para classificação de vulnerabilidades em componentes de Software
- ▶ Grande relevância no mundo contemporâneo
  - ▶ Internet das Coisas
  - ▶ Privacidade e segurança
- ▶ Hipótese principal: conseguir classificar códigos que contenham vulnerabilidades
- ▶ Aplicações e impactos
  - ▶ Demonstra força da área de ciência de dados aplicada à cibersegurança
  - ▶ Redução de custos, mais segurança, etc

# Trabalhos relacionados

- ▶ Aprendizado de máquina já foi aplicado na área da cibersegurança
  - ▶ Diferentes *features* de classificação foram utilizadas, em diversas bases de dados
  - ▶ Diferentes objetivos: localizar vulnerabilidades, identificar correlações entre variáveis, etc
  - ▶ Muitas visavam prever defeitos e não necessariamente vulnerabilidades
- ▶ Análise estática de código
  - ▶ *Fortify Source Code Analyzer* (SCA) - ferramenta de análise estática de código para identificar potenciais vulnerabilidades no código
  - ▶ Utilizado para criação das *labels* do conjunto de dados

# Metodologia de pesquisa

- ▶ Objetivo: construir um classificador binário para prever se um software é provável de ser vulnerável
- ▶ Dados utilizados: arquivos de texto com códigos escritos em Java
- ▶ Métrica para construção das *labels*: um código é dito vulnerável se a ferramenta SCA aponta um ou mais avisos em sua análise e não vulnerável caso contrário
- ▶ Predições analisadas com base em uma matriz de confusão
  - ▶ Verdadeiros positivos (TP)
  - ▶ Falsos positivos (FP)
  - ▶ Falsos negativos (FN)
  - ▶ Verdadeiros negativos (TN)



# Indicadores de performance

- ▶ Precisão ( $P$ ) e Revocação ( $R$ )

$$P = \frac{TP}{TP + FP} \quad R = \frac{TP}{TP + FN}$$

- ▶ F-Score ( $F$ )

$$F_{\beta} = \frac{(1 + \beta^2)PR}{\beta^2 P + R}$$

- ▶ Fall-out ( $O$ )

$$O = \frac{FP}{FP + TN}$$

- ▶ *Benchmark*: 80% ou mais para ambas precisão e revocação

# Conjuntos de dados

- ▶ Aplicações *mobile* da plataforma *Android*
- ▶ Considerações nas decisões:
  - ▶ Linguagem de Programação (Java)
  - ▶ Tamanho (1000 linhas de código no mínimo)
  - ▶ Número de versões (ao menos 5)
- ▶ 10 aplicações escolhidas ao final
- ▶ 10 aplicações pré-instaladas do *Android* foram adicionadas

# Conjunto de variáveis relevantes

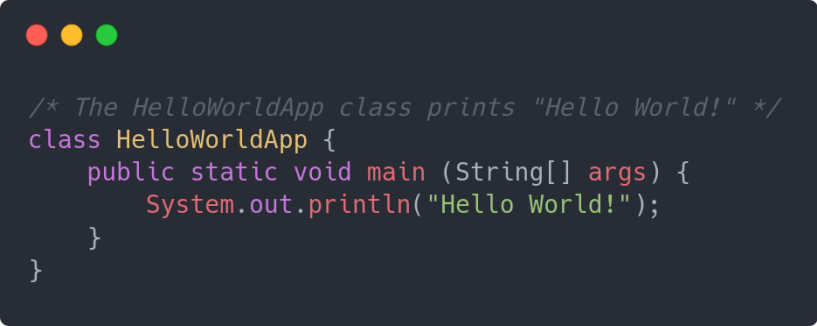
## Variáveis dependentes

- ▶ Ferramentas de análise estática de código são custosas
- ▶ SCA: escaneia o código e monta um relatório de possíveis falhas
- ▶ Reporta tipo de vulnerabilidade juntamente com sua escala de ameaça (trabalhos futuros)

## Variáveis independentes

- ▶ Cada arquivos de código é analisado conforme um *pipeline*
- ▶ Tokenização, contagem de caracteres, etc

# Exemplos



```
/* The HelloWorldApp class prints "Hello World!" */  
class HelloWorldApp {  
    public static void main (String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

**Figura 1:** Arquivo HelloWorldApp.java

► Feature Vector para o arquivo HelloWorldApp.java:

args: 1, class: 2, main: 2, HelloWorldApp: 2, The: 1,  
out: 1, println: 1, prints: 1, public: 1, static: 1,  
String: 1, System: 1, Hello: 1, void: 1, World: 2

# Técnicas de aprendizado de máquina utilizadas

- ▶ *Labels* e *features* já definidas
- ▶ *No free lunch theorem*
- ▶ Inicialmente, cinco algoritmos famosos foram considerados
- ▶ Dois apresentaram melhores resultados iniciais: *Naive Bayes* (NB) e *Random Forest* (RF)
- ▶ Ferramenta utilizada: Weka
- ▶ Discretização de *features*: limpeza do conjunto de dados

# Principais perguntas a serem respondidas

- ▶ Um modelo preditivo pode ser criado?
- ▶ Predições sobre vulnerabilidades futuras podem ser feitas?
- ▶ Predições entre diferentes projetos podem ser feitas?

# Um modelo preditivo pode ser criado?

- ▶ Versão Inicial  $v_0$
- ▶ *Cross Validation 10-fold*
  - ▶ Evitar *overfitting*

# Um modelo preditivo pode ser criado?

Application	Naïve Bayes			Random Forest		
	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{O}$
AnkiDroid	0.92	0.92	0.07	0.85	0.92	0.13
BoardGameGeek	1.00	0.86	0.00	1.00	0.86	0.00
ConnectBot	1.00	0.81	0.00	1.00	0.86	0.00
CoolReader	1.00	0.91	0.00	1.00	0.91	0.00
Crosswords	0.87	0.87	0.10	0.87	0.87	0.10
FBReader	0.67	0.66	0.15	0.87	0.58	0.04
K9Mail	0.95	0.75	0.05	0.91	0.81	0.09
KeePassAndroid	0.85	0.77	0.09	0.88	0.80	0.07
MileageTracker	0.75	1.00	0.14	0.75	1.00	0.14
Mustard	0.97	0.86	0.03	0.93	0.86	0.09
Browser	0.92	0.92	0.05	0.92	0.92	0.05
Calendar	1.00	0.82	0.00	1.00	0.82	0.00
Camera	0.92	0.77	0.07	0.92	0.77	0.07
Contacts	0.91	0.77	0.07	0.91	0.77	0.07
DeskClock	0.75	0.75	0.17	0.71	0.63	0.17
Dialer	0.75	0.71	0.10	0.91	0.59	0.03
Email	0.94	0.72	0.06	0.93	0.81	0.08
Gallery2	0.69	0.62	0.13	0.88	0.55	0.04
Mms	0.86	0.70	0.07	0.94	0.59	0.02
QuickSearchBox	0.62	0.84	0.14	0.83	0.48	0.03

**Figura 2:** Performance média entre os 10 *folds*



# Um modelo pode ser criado

- ▶ 11 dos 20 aplicativos foram “excelentes” com pelo menos um dos algoritmos
- ▶ “Bom o suficiente”: 75% de recall: 15 (NB) e 14 (RF)
- ▶ Restam algumas dúvidas
  - ▶ Influência do tamanho do arquivo
  - ▶ Influência dos falso positivos
  - ▶ Experimentando com dados “reais”

# Influência do tamanho dos arquivos

- ▶ Códigos maiores naturalmente são mais propensos a *bugs* e falhas de segurança
- ▶ Na prática: apenas olhar o tamanho é muito mais ineficiente do que o método proposto
  - ▶ Queda de 25% na precisão
  - ▶ Queda de 37% no *recall*

# Influência dos falsos positivos

- ▶ Ferramentas de análise estática de código podem cometer erros
- ▶ Para mitigar isso, dois aplicativos foram inspecionados “manualmente”: AnkiDroid e Mustard
- ▶ “Custo”: 2 dias de trabalho
- ▶ Resultados
  - ▶ AnkiDroid: 10 de 12 FP
  - ▶ Mustard: 14 de 43 FP

# Influência dos falsos positivos

App		Naïve Bayes		Random Forest	
		$\mathcal{P}$	$\mathcal{R}$	$\mathcal{P}$	$\mathcal{R}$
AnkiDroid	Fortify SCA	0.92	0.92	0.85	0.92
	<b>Validated</b>	<b>1.00</b>	<b>1.00</b>	<b>1.00</b>	<b>1.00</b>
Mustard	Fortify SCA	0.97	0.86	0.93	0.86
	<b>Validated</b>	<b>1.00</b>	<b>0.86</b>	<b>1.00</b>	<b>0.79</b>

*The numbers are in line with the previous results.*

**Figura 3:** Repetição com validação manual

# Influência dos falsos positivos

- ▶ Melhora dos algoritmos, de forma geral
  - ▶ Resultados anteriores seriam um “limite inferior”
- ▶ Mas realmente são?

# Experimentando com dados “reais”

- ▶ Drupal
  - ▶ PHP
  - ▶ Falhas de segurança bem documentadas
  - ▶ Métricas semelhantes a alguns aplicativos: número de arquivos, número de termos e taxa de positivos
- ▶ Resultados
  - ▶ NB: *recall* 73% e precisão 55%
  - ▶ RF: *recall* 82% e precisão 59%
- ▶ Na prática: redução em 45% de arquivos para checar

# Predições sobre vulnerabilidades futuras

- ▶ Modelo da versão inicial testado nas versões subsequentes
- ▶ Não somente se é possível, mas por quanto tempo é possível
  - ▶ Limite arbitrário: deterioração de 10% da performance do  $F_2$

# Predições sobre vulnerabilidades futuras

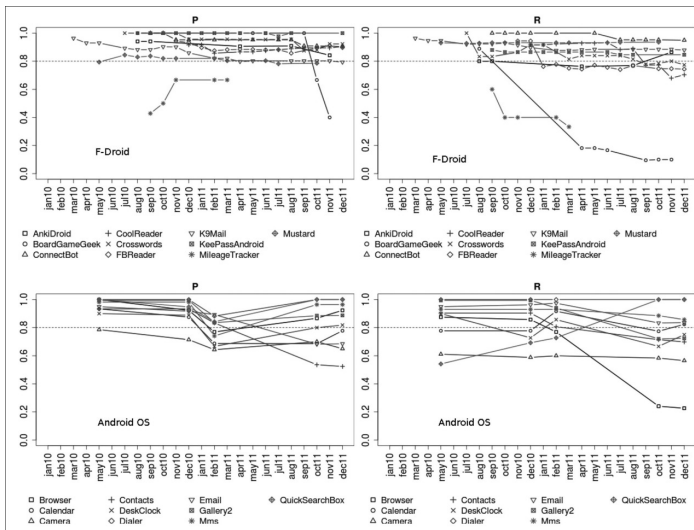


Figura 4: Precisão e *recall* em versões futuras com o RF



# Predições sobre vulnerabilidades futuras

Application	Naïve Bayes			Random Forest		
	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{O}$	$\mathcal{P}$	$\mathcal{R}$	$\mathcal{O}$
AnkiDroid	0.85	0.82	0.23	0.91	0.80	0.13
BoardGameGeek	0.51	0.32	0.08	0.87	0.24	0.01
ConnectBot	0.99	0.92	0.01	0.95	0.98	0.04
CoolReader	1.00	0.77	0.00	0.89	0.85	0.10
Crosswords	0.90	0.76	0.07	0.95	0.84	0.04
FBReader	0.66	0.74	0.16	0.89	0.78	0.04
K9Mail	0.87	0.73	0.10	0.85	0.91	0.15
KeePassAndroid	0.85	0.69	0.08	1.00	0.86	0.00
MileageTracker	0.59	0.43	0.18	0.59	0.43	0.18
Mustard	0.83	0.86	0.15	0.81	0.93	0.19
Browser	0.88	0.57	0.04	0.90	0.59	0.03
Calendar	0.77	0.75	0.17	0.79	0.81	0.16
Camera	0.72	0.49	0.07	0.70	0.59	0.09
Contacts	0.76	0.70	0.11	0.75	0.81	0.13
DeskClock	0.79	0.68	0.16	0.81	0.78	0.16
Dialer	0.72	0.70	0.11	0.98	1.00	0.01
Email	0.83	0.71	0.16	0.83	0.91	0.21
Gallery2	0.70	0.64	0.12	0.92	0.87	0.03
Mms	0.80	0.73	0.11	0.93	0.91	0.04
QuickSearchBox	0.62	0.80	0.14	0.96	0.79	0.01

*The performance of Naïve Bayes is inferior.*

**Figura 5:** No geral, a previsão de versões futuras é melhor com o RF

# Predições sobre vulnerabilidades futuras

- ▶ Desempenho satisfatório, especialmente no *recall*, que é mais importante
- ▶ Apenas 4 aplicações não “se aproximaram” de ter um desempenho favorável
  - ▶ MileageTracker: poucos arquivos
  - ▶ BoardGameGeek: refatoração na  $v_2$
  - ▶ Browser: aumento no número de vulnerabilidades
  - ▶ Camera: autores não comentam
- ▶ Estariam os dados apenas sendo “decorados”?
  - ▶ Arquivos em comum com a versão inicial
  - ▶ Por outro lado, o tamanho dos aplicativos varia bastante
  - ▶ E vulnerabilidades presentes em um arquivo são removidas
  - ▶ Ou ausentes são adicionadas (com o passar do tempo)

# Predições sobre vulnerabilidades futuras

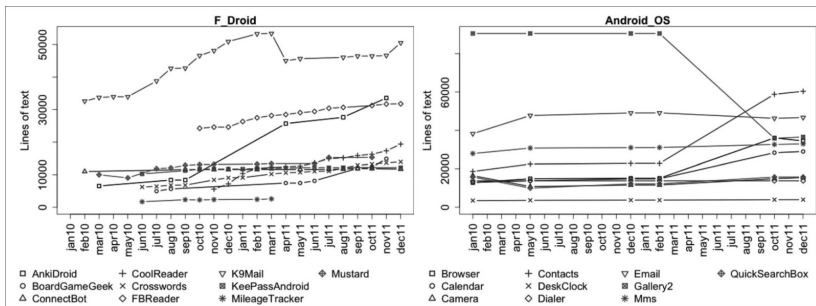


Figura 6: Evolução do Tamanho dos aplicativos

## Predições sobre vulnerabilidades futuras

	Retrain after (months)				
	NB	RF		NB	RF
AnkiDroid	21 *	21 *	Browser	13	13
BoardGameGeek	9	9	Calendar	23 *	23 *
ConnectBot	22 *	22 *	Camera	21	23 *
CoolReader	10	10	Contacts	21	13
Crosswords	2	2	DeskClock	23	23
FBReader	14 *	3	Dialer	23 *	23 *
K9Mail	22 *	22 *	Email	21	21
KeePassAndroid	18 *	18 *	Gallery2	23 *	21
MileageTracker	4	4	Mms	23 *	23 *
Mustard	21 *	21 *	QuickSearchBox	23 *	23 *
<b>Average</b>	<b>14</b>	<b>13</b>		<b>21</b>	<b>21</b>

*In most cases, retrain is never needed over the two years period.*

**Figura 7:** Tempo, em meses, para treinar o modelo novamente

# Predições sobre vulnerabilidades futuras

- ▶ Performance naturalmente decai com o tempo
- ▶ Mas se mantém estável para 11 casos
- ▶ Tempo médio para retreino:
  - ▶ F-Droid: 13 meses
  - ▶ Sistema Operacional: 21 meses

# Predição entre projetos

- ▶ Objetivo ambicioso
- ▶ Até então não testado na área de predição de vulnerabilidades
- ▶ 20 modelos usando as versões iniciais de cada aplicativo
- ▶ Testando nos outros 19 aplicativos, para cada aplicativo
- ▶ Objetivo: atingir o *benchmark*
  - ▶ Falta de análise mais precisa (somente resposta binária)

## Predição entre projetos

Model is applicable to (no. of other apps)					
	NB	RF		NB	RF
<i>AnkiDroid</i>	1	4	<i>Browser</i>	0	0
<i>BoardGameGeek</i>	0	0	<i>Calendar</i>	1	1
<i>ConnectBot</i>	0	0	<i>Camera</i>	0	1
<i>CoolReader</i>	0	1	<i>Contacts</i>	1	0
<i>Crosswords</i>	0	0	<i>DeskClock</i>	0	0
<i>FBReader</i>	5	0	<i>Dialer</i>	0	0
<i>K9Mail</i>	3	2	<i>Email</i>	0	0
<i>KeePassAndroid</i>	0	0	<i>Gallery2</i>	2	0
<i>MileageTracker</i>	0	0	<i>Mms</i>	0	0
<i>Mustard</i>	4	1	<i>QuickSearchBox</i>	0	0
<i>Some models have a more general applicability.</i>					

**Figura 8:** Aplicabilidade entre projetos

# Predição entre projetos

- ▶ Maioria (13) dos aplicativos não são generalizáveis
- ▶ Os que são, no geral, são generalizáveis em poucos casos
- ▶ Aplicativos com mais vulnerabilidades (como o K9Mail) possuem um desempenho melhor
  - ▶ Com o RF,  $F_2$  em 78%
  - ▶ No geral, um *recall* considerável
  - ▶ Quantidade de arquivos não parece ser um fator relevante
- ▶ Não é esperado que aplicativos que não tiveram uma boa avaliação de versões futuras (como o BoardGameGeek) possuam um bom desempenho em outros projetos



# Predição entre projetos

- ▶ É necessária uma análise mais profunda para averiguar os critérios que determinam um bom resultado entre projetos
  - ▶ Hipótese: a “natureza” do texto pode ser um fator de importância
- ▶ Combinação de aplicativos
  - ▶ Análise “futura”
  - ▶ Modelo construído usando 2 ou mais aplicativos como treino
  - ▶ Aplicativos diversificados

# Ameaças à validade

- ▶ Não foi feito uso de um banco de dados de vulnerabilidades
  - ▶ Ao invés disso, foi usada uma ferramenta de análise estática de código
  - ▶ Existe uma correlação entre essas alternativas, mas há uma tendência produzir muitos falsos positivos
  - ▶ Validação manual para mitigar esse problema
- ▶ Apenas arquivos Java foram analisados
  - ▶ Arquivos XML contém, por exemplo, permissões usadas pelos aplicativos
- ▶ Vulnerabilidades que se mantiveram através das versões podem ter inflado os resultados
- ▶ É necessário um estudo em escala maior
  - ▶ Os resultados podem ser específicos dos aplicativos selecionados
  - ▶ Variação dos tipos de programas

# Ameaças à validade

- ▶ Validação manual não foi extensiva
- ▶ Análise muito específica
  - ▶ Limitação da época
    - ▶ Ambiente Android mais diverso atualmente

# Conclusões

- ▶ Variação do nível dos componentes (classes, métodos)
- ▶ Análise poderia ser complementar a métodos já existentes