UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA




# Bugspotting 2.0 - Criação de datasets de vulnerabilidades com recurso a inteligência artificial

**David Miguel Dias Gonçalves**

**Mestrado em Segurança Informática**

Dissertação orientada por:
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves
Profª. Doutora Iberia Vitória de Sousa Medeiros

2025

# Agradecimentos

Na construção desta dissertação, quero expressar a minha profunda gratidão a todos aqueles que, de alguma forma, contribuíram para a sua realização. No entanto, gostaria de destacar algumas pessoas que foram cruciais e imprescindíveis para a sua conclusão.

Em primeiro lugar, um sincero agradecimento aos meus orientadores, Professor Doutor Nuno Neves e Professora Doutora Ibéria Medeiros, cujo conhecimento, disponibilidade e orientação foram essenciais para a concretização desta dissertação. O seu acompanhamento incansável, conselhos e paciência tornaram possível ultrapassar os desafios que encontrei ao longo deste percurso.

Agradeço também de forma muito especial à minha namorada, Ana Rita Ferreira, pelo seu apoio incondicional em todos os momentos, pela paciência, compreensão e por estar sempre presente, nos bons e nos maus momentos, proporcionando-me força e motivação para continuar.

Um agradecimento especial aos meus pais, Victor e Maria da Luz, pelo apoio constante e pelo incentivo, mesmo à distância, garantindo sempre que nunca me faltasse nada.

Agradeço igualmente à minha irmã, Ana, e ao meu afilhado, Bernardo, pelo carinho e companhia, que tornaram este caminho mais fácil.

Agradeço ainda aos meus vizinhos Carlos e Albertina, que na falta dos meus pais, pois moram longe, ajudaram em tudo o que foi possível.

Não posso deixar de agradecer aos meus avós, que diariamente se preocupam e perguntam pelo meu trabalho e percurso académico. Infelizmente, um dos meus avós partiu recentemente, e é com grande tristeza que não pode estar aqui para ver este momento. No entanto, a sua memória será sempre uma inspiração para mim.

A todos vós, o meu mais sincero obrigado.

# Resumo

A evolução tecnológica e o processamento cada vez mais rápido da informação colocam desafios ao funcionamento e segurança eficaz dos programas (*softwares*) e das aplicações *web*. Neste sentido, os alvos mais escolhidos pelos atacantes são aplicações *web* de modo a explorar possíveis vulnerabilidades que elas contêm. Tornou-se um objetivo principal e essencial prevenir, detetar e mitigar de forma rápida e eficaz estas mesmas vulnerabilidades.

As ferramentas de análise estática de código (*Static Analysis Tools, SASTs*) têm vindo a ser amplamente usadas na deteção de vulnerabilidades pela inspeção do código fonte das aplicações. Estas ferramentas realizam uma análise de execução rápida e de baixo custo de manutenção, pois é mais fácil de manter e modificar o código, mas como contrapartida geram muitos falsos positivos e negativos, tornando-se assim necessária a implementação de outros métodos em conjunto com a sua análise. Apesar das SAST serem usadas por profissionais com muito conhecimento, estas não são perfeitas, e são propícias a erros, como referido anteriormente.

Estudos recentes demonstram que o uso de técnicas de Aprendizagem Automática (*Machine Learning*, ML) pode ser empregue para o auxílio e aprimoramento na construção destas ferramentas, melhorando as mesmas [8].

Neste sentido, para que possamos aplicar estas técnicas é necessário a construção de um *dataset* preciso e confiável para treinar modelos de ML. Tendo por base o *website* BugSpotting 1.0, que permite a obtenção de várias classificações de (*slices*, pedaços de código PHP) tanto por SASTs, como por *crowdsourcing*. Nesta dissertação propomos uma nova forma de construção de *datasets*, bem como a melhoria dos algoritmos de classificação de *slices* que permitirão a composição de *datasets* confiáveis e no uso de modelos de ML para a descoberta de vulnerabilidades em aplicações *web*, escritas em PHP. Estas novas implementações resultam numa nova versão do BugSpotting, a 2.0.

**Palavras-chave:** Vulnerabilidades de aplicações *web* em PHP, Ferramentas de Análise Estática (SAST), Classificação de instâncias para uso de aprendizagem automática, Construção de *datasets* para ML, *Crowdsourcing*

# Abstract

Technological advancements and increasingly faster information processing challenges the effective functioning and security of software programs and web applications. In this context, web applications are the primary targets chosen by attackers, aiming to exploit potential vulnerabilities they may contain.

It has become a key and essential objective to prevent, detect, and mitigate these vulnerabilities quickly and effectively.

Static code analysis tools (Static Analysis Tools, SASTs) have been widely used for vulnerability detection through the inspection of application source code. These tools perform a fast execution analysis with low maintenance costs, as maintaining and modifying the code is easier. However, they generate a high number of false positives and negatives, making it necessary to implement additional methods alongside their analysis. Although SASTs are used by highly knowledgeable professionals, they are not perfect and are prone to errors, as previously mentioned.

Recent studies have shown that the use of Machine Learning (ML) techniques can assist and enhance the development of these tools, improving their effectiveness [8].

In this regard, to apply these techniques, it is necessary to build a precise and reliable dataset to train ML models. This work is based on the BugSpotting 1.0 website, which allows for the classification of slices (pieces of PHP code) both by SASTs and through crowdsourcing. In this dissertation, we propose a new approach to building datasets, as well as improving the classification algorithms of slices, which will enable the creation of reliable datasets and the use of ML models for discovering vulnerabilities in web applications written in PHP. These new implementations result in a new version of BugSpotting, version 2.0.

**Keywords:** Vulnerabilities on *web* applications in PHP, Static Analysis Tools (SAST), Instance classification for use in ML, Construction of *datasets* used from , *Crowdsourcing*

# Índice

**Lista de Figuras** **ix**

**Lista de Tabelas** **xi**

**Lista de Algoritmos** **xiii**

**Lista de Acrónimos** **xvi**

**1 Introdução** **1**

1.1 Motivação . . . . . 1

1.2 Contribuições . . . . . 2

1.3 Estrutura do documento . . . . . 3

**2 Conceitos e Trabalho Relacionado** **5**

2.1 Vulnerabilidades . . . . . 6

2.2 Deteção de vulnerabilidades . . . . . 6

2.3 Classes de Vulnerabilidades . . . . . 6

2.3.1 *Cross-Site Scripting* (XSS) . . . . . 6

2.3.2 *SQLinjection* (SQLi) . . . . . 7

2.3.3 *Remote File Inclusion* (RFI) . . . . . 7

2.3.4 *Local File Inclusion* (LFI) . . . . . 7

2.3.5 Directory Traversal . . . . . 7

2.3.6 Source Code Disclosure . . . . . 7

2.3.7 OS Command Injection . . . . . 8

2.3.8 PHP Code Injection . . . . . 8

2.4 Crowdsourcing . . . . . 8

2.5 Ferramentas análise estática . . . . . 9

2.6 Inteligência artificial . . . . . 10

2.6.1 Machine Learning . . . . . 10

2.7 BugSpotting 1.0 . . . . . 11

2.7.1 Análise ao BugSpotting 1.0 . . . . . 11

**3 BugSpotting 2.0** **13**
3.1 Visão Geral . . . 13
3.2 Arquitetura . . . 14
3.2.1 Arquitetura de três camadas . . . 15
3.2.2 Padrões MVC e MVVM . . . 15
3.2.3 API BugSpotting . . . 16
3.3 Algoritmos do processo para atribuição de etiquetas . . . 17
3.3.1 Conceitos e variáveis . . . 17
3.3.2 Processo configuração BugSpotting . . . 21
3.3.3 Execução da ferramenta SAST . . . 22
3.3.4 Algoritmos . . . 22

**4 Desenvolvimento BugSpotting 2.0** **45**
4.1 Tecnologias . . . 45
4.1.1 Componentes utilizados . . . 45
4.1.2 Base Dados . . . 48
4.2 BugSpotting 2.0 Interface . . . 54

**5 Avaliação** **57**
5.1 Simulações de alguns cenários . . . 57
5.1.1 Classificação de *Slices* . . . 58
5.1.2 Tendência . . . 61
5.1.3 Indecisão . . . 64

**6 Conclusão** **69**
6.1 Trabalho Futuro . . . 69

**Bibliografia** **73**

**Índice** **74**

# Lista de Figuras


2.1 Vertentes do Crowdsourcing . . . . . 8
2.2 Camadas da inteligência artificial . . . . . 10

3.1 Padrão MVC . . . . . 16
3.2 Diagrama da configuração do sistema . . . . . 21
3.3 Diagrama obter slice . . . . . 23
3.4 Diagrama obter reputação . . . . . 28
3.5 Diagrama obter reputação dinâmica . . . . . 39
3.6 Diagrama obter grau de indecisão . . . . . 43

4.1 Componentes utilizadas no BugSpotting 2.0 . . . . . 45
4.2 HTML, CSS e JavaScript . . . . . 47
4.3 Modelo ER . . . . . 49
4.4 Interface BugSpotting 2.0 . . . . . 54
4.5 Classificação de Slice de reputação BugSpotting 2.0 . . . . . 55
4.6 Classificação de Slices BugSpotting 2.0 . . . . . 55

5.1 Gráfico VC1-S1 de Tendência e criação de etiqueta final em t=8 . . . . . 62
5.2 Gráfico VC1-S2 de Tendência e criação de etiqueta final em t=7 . . . . . 62
5.3 Gráfico VC1-S3 de Tendência e criação de etiqueta final em t=8 . . . . . 63
5.4 Gráfico VC1-S4 de Tendência e criação de etiqueta final em t=8 . . . . . 63
5.5 Gráfico VC2-S1 de Tendência e criação de etiqueta final em t=8 . . . . . 64
5.6 Gráfico VC1-S1 referente tendência e grau de indecisão em t=2 . . . . . 64
5.7 Gráfico VC1-S1 referente tendência e grau de indecisão em t=3 . . . . . 65
5.8 Gráfico VC1-S1 referente tendência e grau de indecisão em t=8 n=56 e N=120 . 65
5.9 Gráfico de barras VC1-S1 referente tendência e grau de indecisão em t=8 n=56 e N=120 . . . . . 66
5.10 Gráfico VC1-S2 referente tendência e grau de indecisão em t=7 n=21 e N=30 . . 66
5.11 Gráfico VC1-S3 referente tendência e grau de indecisão em t=8 n=57 e N=120 . 67
5.12 Gráfico VC1-S4 referente tendência e grau de indecisão em t=8 n=33 e N=120 . 67
5.13 Gráfico VC2-S1 referente tendência e grau de indecisão em t=8 n=24 e N=120 . 68

# Lista de Tabelas


3.1 As oito classes de vulnerabilidades contempladas pelo BugSpotting 2.0 . . . . . 18
3.2 Utilizador 1 classifica a *slice* 1 . . . . . 31
3.3 Utilizador 2 classifica a *slice* 1 . . . . . 31
3.4 Cálculo tendência tempo=1 . . . . . 31
3.5 Cálculo tendência tempo=2 . . . . . 31
3.6 Cálculo tendência tempo=3 . . . . . 32
3.7 Cálculo tendência tempo=8 . . . . . 32
3.8 Exemplo com dois utilizadores e duas classificações . . . . . 33
3.9 Cálculo da avaliação inicial de três utilizadores com 80% de confiança . . . . . 34
3.10 Exemplo com dois utilizadores e duas classificações . . . . . 35
3.11 Reputação dinâmica tempo=1 . . . . . 35
3.12 Reputação dinâmica tempo=2 . . . . . 35
3.13 Reputação dinâmica tempo=3 . . . . . 35
3.14 Reputação dinâmica tempo=8 x-1-1 . . . . . 36
3.15 Classificação tendencial e evolução da reputação dinâmica do utilizador 1 nas VC1-S1 e S4 no t=3 . . . . . 39
3.16 Classificação tendencial e evolução da reputação dinâmica do utilizador 1 nas VC1-S1 e S4 no t=4 . . . . . 40
3.17 Classificação tendencial e evolução da reputação dinâmica do utilizador 2 no tempo 8 nas VC1-S1, S3 e S4 . . . . . 40
3.18 Grau de indecisão referente às 5 slices . . . . . 44

5.1 Exemplo com dois utilizadores e duas classificações . . . . . 59
5.2 Cálculo da avaliação inicial de três utilizadores com 80% de confiança . . . . . 59
5.3 Exemplo com dois utilizadores e duas classificações . . . . . 60
5.4 Reputação dinâmica tempo=1 . . . . . 60
5.5 Reputação dinâmica tempo=2 . . . . . 60
5.6 Reputação dinâmica tempo=3 . . . . . 61
5.7 Reputação dinâmica tempo=8 x-1-1 . . . . . 61

# Lista de Algoritmos


1 Obter Slice . . . . . 25
2 Fase de reputação inicial . . . . . 27
3 CalculateScore . . . . . 27
4 Obter tendência . . . . . 30
5 Obter reputação dinâmica . . . . . 41
6 Resolver grau de indecisão . . . . . 42

# Lista de Acrónimos

**API** *Application Programming Interface*.

**CLR** *Common Language Runtime*.

**CSS** *Cascading Style Sheets*.

**DL** *Deep Learning*.

**ER** Entidade Relacional.

**HTML** *HyperText Markup Language*.

**IOC** *Inversion of Control*.

**JS** *JavaScript*.

**JSON** *JavaScript Object Notation*.

**LFI** *Local File Inclusion*.

**MVC** *Model-View-Controller*.

**MVVM** *Model-View-ViewModel*.

**NIST** *National Institute of Standards and Technology*.

**OS** *Operating System*.

**PHP** *Hypertext Preprocessor*.

**REST** *Representational State Transfer*.

**RFI** *Remote File Inclusion*.

**SARD** *Software Assurance Reference Dataset*.

**SAST** *Static Application Security Testing*.

**SGBDR** Sistema de Gestão de Base Dados Relacional.

**SQLi** *SQLinjection*.

**UI** *Use Interface*.

**UX** *User Experience*.

**VC** Classe de Vulnerabilidade.

**XSS** *Cross-Site Scripting*.

# Capítulo 1

# Introdução

Com o passar dos anos, a sociedade ficou cada vez mais dependente do mundo digital e das tecnologias. O desenvolvimento das tecnologias implicou, por sua vez, um aumento proporcional do número e sofisticação dos ciberataques, daí resulta a importância na breve deteção e prevenção dessas atividades maliciosas.

O fraco desenvolvimento de *software* é, desde logo, um dos principais motivos que contribui para o aumento do número de ameaças, assim como os testes que deveriam ser efetuados e muitas das vezes não o são. Estas lacunas resultam na existência de vulnerabilidades que podem ser exploradas pelos atacantes.

Todas estas falhas podem permitir ao atacante agir maliciosamente, trazendo problemas graves para a vítima alvo, que se for uma grande empresa, comprometendo a sua reputação, credibilidade e confiança no mercado, ou até, perdas financeiras.

Dada a crescente complexidade do ecossistema digital, torna-se imperativa a realização de uma análise minuciosa do código-fonte de todas as aplicações, independentemente da sua finalidade, sejam elas de teste, qualidade ou desenvolvimento, antes de entrarem em produção. O código pode ser analisado usando para isso ferramentas de análise estática (SAST, *Static Analysis Tools*) que é uma forma rápida de verificação do código, com baixo custo e ótimo benefício.

Uma das estratégias adotadas para resolver as situações anteriormente descritas, pressupõe o uso de modelos de machine learning (ML), para deteção de vulnerabilidades no código dos programas. O uso de ML, requer um *dataset* confiável e fidedigno, contendo *slices* de código (i.e., excertos de código) usados para treino. Quanto maior for a qualidade deste *dataset*, mais eficaz será o modelo de ML na deteção e mitigação de vulnerabilidades, contribuindo assim para um ecossistema digital mais seguro.

## 1.1 Motivação

A deteção de vulnerabilidades no código é um fator essencial para garantir a segurança das aplicações *web*, especialmente em linguagens amplamente utilizadas, como é o caso do PHP. Como resposta a essa necessidade, foi desenvolvido o BugSpotting, uma plataforma *web* que permite a criação de *datasets* devidamente etiquetados, para serem posteriormente utilizados no treino

de algoritmos de ML. Esses *datasets* são compostos por *slices* de código classificados como vulneráveis ou não, focando principalmente em algumas das vulnerabilidades mais conhecidas em PHP, como é o exemplo de *SQL Injection (SQLi) e Cross-Site Scripting (XSS)*.

A escolha do PHP para este trabalho deve-se à sua ampla utilização no desenvolvimento e programação de aplicações *web*. No entanto, a linguagem apresenta diversas fragilidades estruturais que a tornam vulnerável a ataques que podem comprometer a segurança dos sistemas e seus utilizadores. Grande parte destes problemas decorre da inconsistência e falta de rigor no desenvolvimento destas aplicações, originando vulnerabilidades exploráveis.

Uma solução ou abordagem utilizada para esse fim é o recurso a ferramentas SAST, que permitem identificar potenciais falhas de segurança sem necessidade de executar o programa. Estas ferramentas ajudam a detetar vulnerabilidades logo na fase de desenvolvimento, evitando custos elevados com correções posteriores e reduzindo o risco de falhas na segurança em produção. Além disso, podem ser utilizadas para reforçar boas práticas de programação, garantindo padrões de qualidade mais elevados no desenvolvimento do *software*. No entanto, apesar das vantagens, estas ferramentas apresentam limitações significativas, pois, os resultados gerados necessitam de análise humana, o que pode tornar o processo moroso e causar impacto no desenvolvimento. Além disso, como a análise é feita sem execução do código, erros em tempo de execução podem passar despercebidos. Outro problema comum é a ocorrência de falsos positivos e negativos, pois as SASTs podem assinalar erroneamente código seguro como vulnerável ou falhar na deteção de vulnerabilidades reais, comprometendo assim a eficácia da análise.

Para garantir a qualidade dos *datasets*, é essencial que todas as etiquetas dadas às *slices* de código sejam precisas e fiáveis, mas devido à elevada taxa de falsos positivos e negativos das SASTs podem gerar, confiar exclusivamente nelas para classificação das *slices* pode comprometer a fiabilidade dos modelos de ML. Torna-se assim necessário recorrer ao auxílio de outra abordagem, o *crowdsourcing*, que permite a vários utilizadores classificarem as *slices* com vista a criar um *dataset* de forma rápida e correta.

## 1.2 Contribuições

Tal como o BugSpotting 1.0, a versão 2.0 mantém os mesmos princípios e objetivos, criação de um *dataset* confiável, deste modo, podem ser enumeradas três contribuições principais:

Em primeiro lugar, foi disponibilizada uma aplicação *web* que permite aos utilizadores a classificação de *slices* de código PHP como sendo vulneráveis ou não, e em que cada um deles possui uma reputação associada, com base no seu nível de experiência.

Em segundo lugar, esta nova versão do BugSpotting, a 2.0, atualiza e adiciona novos algoritmos que permitem a obtenção de um *dataset* fiável de forma célere. Os algoritmos são:

- Obter *slice*;
- Fase de reputação e resolver resultado igual a zero;
- Obter tendência da *slice*;

- Obter reputação dinâmica para cada utilizador;
- Resolver grau de indecisão.

Por fim, o BugSpotting inclui um modelo de dados que organiza e estrutura todas as entidades presentes na base de dados (BD), garantindo assim, o registo adequado de todas as classificações dadas às *slices* de código classificados, pelos utilizadores e SAST.

## 1.3 Estrutura do documento

Este documento está organizado da seguinte forma:

- **Capítulo 2 – Conceitos e Trabalho Relacionado**: Este capítulo apresenta informações essenciais, tais como, conceitos relacionados com vulnerabilidades, classes de vulnerabilidades, deteção, ferramentas SAST e uma análise sobre a versão anterior a esta, o BugSpotting 1.0.
- **Capítulo 3 – BugSpotting 2.0**: Este capítulo descreve a aplicação *web* BugSpotting 2.0 bem como a sua arquitetura, modelos e padrões utilizados no seu desenvolvimento.
- **Capítulo 4 – Desenvolvimento BugSpotting 2.0**: Neste capítulo, são detalhados todos os processos utilizados na implementação do BugSpotting 2.0, tais como o modelo de dados por si utilizado e as tecnologias.
- **Capítulo 5 – Avaliação**: Este capítulo apresenta os métodos e cálculos utilizados para avaliar o desempenho e fiabilidade do BugSpotting 2.0 Nesse sentido foram feitas várias simulações mostrando gráficos para comprovar os resultados obtidos.
- **Capítulo 6 – Conclusão**: O capítulo final encerra este trabalho, resumindo as principais conclusões, sugerindo melhorias e funcionalidades futuras.

# Capítulo 2

# Conceitos e Trabalho Relacionado

Neste capítulo é apresentada uma introdução ao BugSpotting 1.0 e seus algoritmos.

À medida que a tecnologia vai evoluindo, o *software* tem um crescimento proporcional, e a sociedade foi-se tornando cada vez mais dependente desta. Contudo, nem tudo o que está associado à evolução da tecnologia é necessariamente benéfico e positivo, pois em contrapartida, assistimos também num aumento do número de ciberataques, o que resulta numa necessidade urgente de prevenir e detetar de forma atempada ações maliciosas.

São considerados ciberataques todas as ações dirigidas contra sistemas de informação, com o objetivo de prejudicar pessoas, instituições ou empresas.

De entre os mais perigosos, destaca-se a exploração de vulnerabilidades no *software* mal implementado e com testes de segurança efetuados de forma incorreta ou inexistentes [27].

Existem diferentes métodos de prevenção e tratamento relativamente a este tipo de ataques que visam, por exemplo, educar os utilizadores, evitando que estes acedam a *links* suspeitos impedindo que enviem informações pessoais. Outra das técnicas utilizadas para evitar este tipo de ataques pressupõe a utilização de sistemas inteligentes, baseados em algoritmos e técnicas de ML, para identificação de padrões de comportamento que podem indiciar a existência de um ciberataque. Deste modo, utilizando *datasets* de qualidade, estes sistemas examinam esses padrões de comportamento na deteção [7], podendo evitar um ataque.

Esta abordagem utilizada na deteção automática de vulnerabilidades, com recurso a ML e dados reais, demonstra que é possível superar os métodos tradicionais de deteção de vulnerabilidades, alcançando taxas de sucesso mais elevadas.

Como referido anteriormente, pequenas falhas na implementação e desenvolvimento de *software* resultam em potenciais vulnerabilidades que comprometem sistemas e aplicações, sendo que as ferramentas de análise de *software* existentes apenas deteta um sub-conjunto de erros e vulnerabilidades, o que se torna um problema.

Contudo, com a recente disseminação de código aberto (*open source*), tornou-se possível usar as técnicas anteriormente descritas na descoberta de padrões de vulnerabilidades [29]. Assim, como anteriormente referido, as técnicas de ML afirmam-se como uma mais-valia enquanto forma de deteção automatizada de vulnerabilidades.

## 2.1 Vulnerabilidades

Segundo o *National Institute of Standards and Technology* (NIST), uma vulnerabilidade é definida como uma fraqueza num sistema de informação, seus procedimentos de segurança, controlos internos e externos ou implementação que poderá ser explorada por ameaças. É, como tal, necessária a adoção de medidas que mitiguem todo este processo[21].

Aquando do desenvolvimento do trabalho relacionado [9], foram selecionadas oito classes de vulnerabilidade (*Vulnerability Class*, VC) consideradas críticas, sendo que algumas *slices* PHP poderão ser vulneráveis para mais do que uma VC.

## 2.2 Deteção de vulnerabilidades

O *software* é um componente crítico dos sistemas de informação quando falamos de vulnerabilidades. Está provado que a maior parte dos incidentes são iniciados pelos *hackers* que exploram nos seus ataques, as vulnerabilidades presentes no próprio *software*.

No decorrer do processo de análise de código, estando o programa a ser verificado, existem dois tipos de deteção de vulnerabilidades: a dinâmica e estática. Neste caso, optámos por utilizar inicialmente, ferramentas SAST, pois estas têm provado ser efetivas em grande parte dos sistemas de *software*.

Todavia, estas podem-se tornar ”pesadas”, a nível de processamento, pois para além de consumirem muito tempo no decurso da análise [16], existem por outro lado, ferramentas mais leves (a nível de processamento), de análise estática que podem ser aplicadas, como o *Splint* que foi projetado para ser eficiente e escalável, permitindo a deteção de falhas de implementação aproveitando anotações adicionadas no código.

As SAST por norma, focam-se em casos mais específicos de vulnerabilidades, que, uma vez que se baseiam em probabilidades, podem gerar uma taxa de falsos positivos e negativos muito elevada.

## 2.3 Classes de Vulnerabilidades

### 2.3.1 *Cross-Site Scripting* (XSS)

O *Cross-Site Scripting* é um tipo de ataque que ocorre quando cibercriminosos injetam *scripts* maliciosos em sites legítimos, com o objetivo de explorar vulnerabilidades em campos específicos tais como, barra de pesquisa ou até no campo dos comentários. Esses *scripts* são executados no *browser* da vítima, permitindo o roubo de dados tais como, *cookies* de sessão, *passwords* ou até redirecionar o utilizador para sites controlados pelo atacante [13].

Existem três tipos de ataques XSS:

- *XSS Stored*- O código malicioso é armazenado no *website* e executado sempre que a página é acedida pelos utilizadores.

- XSS Reflected- A *payload* maliciosa é enviada na solicitação e refletida na resposta do servidor.
- XSS DOM-Based- O código é executado no lado do cliente, sem aparecer qualquer resposta do servidor.

### 2.3.2 *SQLinjection* (SQLi)

SQL Injection é um ataque que explora vulnerabilidades nos sistemas de segurança de base de dados (BD) [11], permitindo aos atacantes aceder e manipular informações lá armazenadas. Este ataque usa injeção de código malicioso SQL nas consultas efetuadas à BD, comprometendo a sua integridade, bem como permitindo o roubo de dados lá armazenados. Este tipo de injeções podem ocorrer por exemplo através de formulários presentes na página *web*.

### 2.3.3 *Remote File Inclusion* (RFI)

Remote File Inclusion é um tipo de vulnerabilidade que permite a inclusão de ficheiros nas aplicações *web* mal configuradas, permitindo assim aos atacantes o carregamento de ficheiros maliciosos para execução. RFI permite que o atacante inclua ficheiros localizados no servidor remoto, podendo assim executar *scripts* maliciosos [22] o que estejam ”hospedados”num servidor externo, ganhando assim acesso ao sistema, comprometendo a integridade do mesmo.

### 2.3.4 *Local File Inclusion* (LFI)

Local File Inclusion tal como o RFI é um tipo de vulnerabilidade que permite a inclusão de ficheiros nas aplicações *web* mal configuradas, mas neste caso LFI explora vulnerabilidades que permitem a inclusão de ficheiros locais no servidor. O atacante pode usar LFI para aceder a ficheiros sensíveis do sistema, como *passwords, logs* ou executar código malicioso já presente no servidor.

### 2.3.5 Directory Traversal

Directory ou Path Traversal é uma vulnerabilidade de segurança que permite a um atacante aceder a ficheiros ou diretórios fora do diretório raiz da aplicação *web* [24]. Esta falha ocorre quando a aplicação não valida corretamente a entrada do utilizador em relação aos caminhos dos ficheiros.

O atacante manipula parâmetros de entrada para incluir sequências de caracteres (../ ou equivalentes) que indicam ao sistema para subir níveis na estrutura de diretórios, podendo aceder assim a ficheiros confidenciais ou comprometer a integridade do sistema.

### 2.3.6 Source Code Disclosure

Source Code Disclosure ou divulgação do código fonte é uma vulnerabilidade que permite a um atacante aceder ao código fonte de uma aplicação. Esta falha expõe os ficheiros que contêm o código da aplicação *web*, variáveis, *scripts* ou configurações [26]. A divulgação do código fonte

pode acontecer devido a erros de configuração no servidor, aplicação ou até em ficheiros de *backup* ou temporários.

### 2.3.7 OS Command Injection

É uma vulnerabilidade de segurança que permite ao atacante executar comandos arbitrários diretamengte no sistema operativo onde a aplicação vulnerável está a ser executada. Esta falha ocorre quando uma aplicação passa dados do utilizador para um comando do sistema operativo (*Operating System* (OS)) sem a devida validação ou sanitização adequada [23].

### 2.3.8 PHP Code Injection

É uma vulnerabilidade que permite ao atacante injetar e executar código PHP arbitrário numa aplicação vulnerável. Esta falha ocorre quando uma aplicação processa todos os *inputs* do utilizador sem validar corretamente, incorporando-os diretamente em funções que os interpretam ou executam código PHP. Quando uma aplicação aceita os dados vindos do utilizador e os inclui no código PHP sem a devida sanitização, o atacante pode introduzir código malicioso que será executado no servidor [25].

## 2.4 Crowdsourcing

O *crowdsourcing* é uma abordagem em que é utilizado o trabalho previamente desenvolvido, de uma forma coletiva, por um elevado número de pessoas, na realização de tarefas de modo a alcançar uma solução a que os computadores, por si só, não conseguiriam chegar [34].

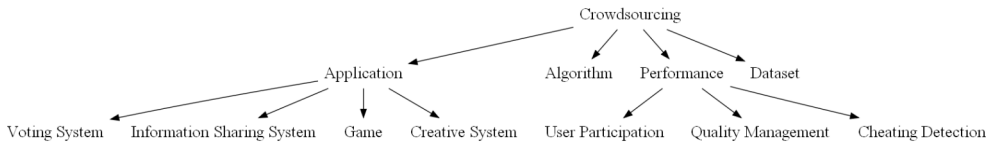



Figura 2.1: Vertentes do Crowdsourcing

O *crowdsourcing* é utilizada em várias vertentes, tal como representado na Figura 2.1 [34], mas também, nomeadamente na execução de tarefas repetitivas. Ainda que tais tarefas possam e devam ser automatizadas, elas exigem inteligência humana. A metodologia de *crowdsourcing* pode ser definida de duas formas: granularidade e incentivos [10].

A granularidade reporta-se à complexidade da tarefa a ser resolvida pelos utilizadores, podendo ser definida como micro ou macro.

Quando focado no incentivo, o *crowdsourcing* é descrito como um tipo de reconhecimento ou motivação atribuído ao utilizador, para que este possa participar na tarefa e, deste modo receber como incentivo dinheiro, serviços ou entretenimento com o objetivo de acelerar a realização das tarefas.

De todos os incentivos enumerados, o principal modo de compensação é o monetário, que se verifica sempre que o utilizador recebe um valor pela execução da tarefa. No caso das compensações com recurso a serviços e/ou entretenimento, os utilizadores são incentivados a responder a determinadas perguntas enquanto executam *quizzes*, desafios ou puzzles, na tentativa de se obter mais respostas e opiniões de um determinado grupo de pessoas.

No BugSpotting 2.0, o *crowdsourcing* é utilizado para gerir reputações, pesos e classificações dos utilizadores. Permite, por isso, que os utilizadores da aplicação desenvolvam as tarefas com habilidade, alcançando um consenso quanto a uma determinada *slice*.

Neste sentido, pretendemos adicionar e integrar uma nova vertente à metodologia de *crowdsourcing*, usando, como recurso, o gamificação, para obter mais classificações por parte dos utilizadores. Por intervenção do gamificação, teremos uma experiência próxima à de um jogo, envolvendo o utiliza na resolução de mais desafios [31]. No nosso caso, será possível obter uma maior quantidade de *slices* classificadas, em virtude de um melhor *dataset*. Um estudo revelou que o recurso a gamificação incentiva uma maior utilização de plataformas de *software* pelos utilizadores, o que influencia positivamente a sua motivação para aprender [3]. Posteriormente, poderão ser adicionados *NonPlayer Characters* (NPCs) [18], com o intuito de unir mais os utilizadores, no sentido de estes se ajudarem mutuamente, partilhando dicas. Estes jogadores não são humanos, mas têm como objetivo dar a sensação de que o utilizador não está sozinho na tarefa que está a desempenhar. A motivação é a chave.

## 2.5 Ferramentas análise estática

A análise estática é uma técnica que tem como principal objetivo analisar o código fonte de uma aplicação, sem a necessidade de este ser executado, na tentativa de encontrar erros, padrões e vulnerabilidades.

Tais ferramentas foram projetadas de modo a encontrar, de forma automática, vulnerabilidades no código. Porém, o desenvolvimento deste tipo de ferramentas, pressupõe alguns requisitos por parte de quem as desenvolve, desde logo, conhecimento das vulnerabilidades, linguagem de programação e do código que está a ser programado. O que os programadores pretendem, é utilizar este tipo de ferramentas de análise estática na deteção e resolução de vulnerabilidades deixadas no código desenvolvido.

No entanto, se estas ferramentas de análise estática não forem treinadas com recurso aos melhores métodos e práticas, tais como, regras e heurísticas, de modo a identificar padrões de código com erros ou vulnerabilidades, podem gerar muitos falsos negativos e positivos. Além disso, não podemos ignorar que as equipas de desenvolvimento de software podem produzir falhas, inconscientes, durante a escrita de código, surgindo erros que são introduzidos de forma involuntária [16].

## 2.6 Inteligência artificial

Nesta secção é apresentada uma das camadas pertencentes à inteligência artificial, *machine learning*, existindo também *deep learning*.

A inteligência artificial permite aos sistemas ter a capacidade de realizar tarefas que normalmente exigem inteligência humana, tal como a aprendizagem, raciocínio e decisão. Na segurança de *software* e análise de código, a Inteligência artificial é comumente usada para identificação de vulnerabilidades, detetar padrões de ataque bem como correção de falhas forma automática [35].

### 2.6.1 Machine Learning

Os modelos de ML e suas funcionalidades abriram horizontes para a criação de mecanismos inteligentes de apoio à decisão, tendo sido ultrapassados os limites de processamento do conhecimento humano.

Estes modelos permitem, como já referido, que as aplicações façam previsões baseadas em dados pré-existentes, sem supervisão humana [20].

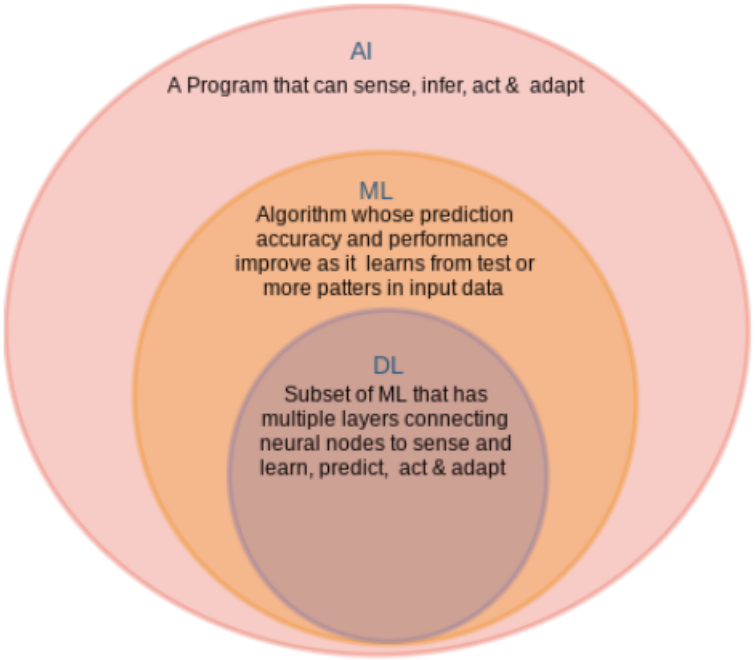



Figura 2.2: Camadas da inteligência artificial

ML é uma das camadas da inteligência artificial, tal como podemos ver na Figura 2.2, que tem como objetivo permitir que máquinas executem os seus trabalhos de forma capaz, usando, para o efeito, software inteligente [28]. Em ML, o processo de aprendizagem é dividido em duas fases, treino e teste.

Primeiro, são testadas iterações no sistema de modo a aprimorar o modelo, que é obtido através de exemplos de conjuntos de dados (*datasets*). No processo de teste, este usa o modelo de aprendizagem de forma a que este possa fazer previsão de dados. Como output, obtemos dados etiquetados

e classificados [28]. A Inteligência Artificial possibilita, há muito, que as máquinas realizem trabalho, tal como um ser humano, mesmo não estando explicitamente programadas, com instruções detalhadas, para o efeito.

Desde logo, o recurso a métodos estatísticos permite que as máquinas se desenvolvam e se capacitem a partir de dados ou de *datasets*, previamente definidos por humanos, através de ML. Por último, o *Deep Learning* (DL) utiliza redes neuronais, como uma combinação de múltiplas camadas de ML, sendo que, cada uma delas, aprende a partir da camada anterior [19]. A aprendizagem automática ou *Machine Learning* é um *subset* da inteligência artificial que pode ser adaptada a um grande conjunto de dados e instruções.

Pertence a um campo da ciência da computação que utiliza algoritmos que têm possibilidade de aprendizagem com dados, melhorando assim o seu desempenho ao longo do tempo. Estes algoritmos são capazes de identificar padrões num grande conjunto de dados e fazer previsões, bem como tomar decisões. Como o próprio nome diz, aprendizagem, neste processo, um algoritmo é treinado com um conjunto de dados, analisando e identificando padrões e relações entre eles. Posteriormente com base nesses padrões, o algoritmo faz previsões e toma decisões para dados novos após a fase de treino.

## 2.7 BugSpotting 1.0

O BugSpotting 1.0 consiste numa plataforma *web* para a criação de *datasets*, através da classificação de *slices* pelas ferramentas de análise estática (SAST) e *crowdsourcing*. A falta de *datasets* com a qualidade necessária é uma das principais fragilidades, quando falamos de modelos relacionados com ML, pois estes são tanto eficazes quanto melhor for a qualidade do *dataset* no âmbito do qual ele irá ser treinado.

Estas *slices* PHP podem ser classificadas como positivas ou negativas. Por vezes, as SAST fornecem resultados incorretos, tornando-se necessário o recurso ao *crowdsourcing* para aprimorar todo este processo. Quanto mais classificações positivas forem dadas a uma determinada *slice*, mais hipóteses tem esta de ser classificada como vulnerável.

### 2.7.1 Análise ao BugSpotting 1.0

O BugSpotting 1.0 tem como foco o código programado na linguagem PHP, sendo esta uma das linguagens frequente- mente utilizadas no desenvolvimento de *websites* [9]. Para que todo este processo funcione, foi desenvolvida uma plataforma que permite tanto às ferramentas de análise estática, como aos humanos, classificar estes pedaços de código, informando, deste modo, se são ou não vulneráveis a uma ou mais classes de vulnerabilidade.

Os dois tipos de classificadores são o *crowdsourcing* e ferramentas SAST, que têm uma reputação associada, significando isto que todo este processo de classificação é influenciado pela reputação da ferramenta e utilizador.

Na preparação e execução do BugSpotting 1.0 foram tomados em conta os seguintes requisitos funcionais, de modo a que os utilizadores obtenham uma melhor experiência:

- Facilidade de uso, pois é importante aplicar normas intuitivas e fáceis no *website*, tendo nesse sentido sido

  utilizadas as normas de *User Experience* (UX) e *Use Interface* (UI), tornando-o, assim, mais acessível e navegável;

- Precisão, considerando que é fundamental a criação de *datasets* fiáveis, combinando o resultado da execução da ferramenta SAST com o *feedback* dado pelos utilizadores do sistema. Deste modo, é dado um peso a cada um, chegando-se a um consenso sobre a classificação final, como vulnerável ou não vulnerável, a dar à *slice* PHP;

- Extensibilidade, com o passar do tempo, mais utiliza- dores utilizarão a plataforma e atribuirão a sua classificação.

  Nesse momento, contamos apenas com uma ferramenta de SAST, aplicada na fase inicial da solução e com a opinião dos utilizadores, que se vai aprimorando com o passar do tempo. Ao contrário do *Software Assurance Reference Dataset* (SARD), estes *datasets* não são totalmente realistas [5][9], uma vez que a maior parte dos programas usados por ele são sintéticos, não correspondendo ao *software* da vida real.

As *slices* de código presentes na base de dados do BugSpotting 1.0, por sua vez, pertencem a casos de uso de cenários reais. Deste modo, é possível treinar e melhorar os sistemas de ML, com código de casos reais, garantindo melhores oportunidades de deteção de vulnerabilidades, uma melhor fase de treino para os algoritmos de ML e um *dataset* mais otimizado.

# Capítulo 3

# BugSpotting 2.0

Neste capítulo será apresentada a aplicação *web* BugSpotting 2.0, que tem como principal objetivo a criação de um *dataset* fiável que contenha pedaços de código (*slices*) *Hypertext Preprocessor* (PHP) etiquetados de forma correta. O BugSpotting 2.0 corresponde a uma nova versão da anterior 1.0, trazendo várias melhorias.

Este capítulo está dividido em quatro secções. Na Secção 3.1 é apresentada uma breve introdução ao BugSpotting 2.0, explicando o seu objetivo e finalidade, posteriormente, na Secção 3.2 é efetuada uma explicação da arquitetura que suporta esta aplicação *web*. Por fim na Secção 3.3 é apresentada a proposta de solução, que tem como objetivo aprimorar e criar novos algoritmos e procedimentos que nos permitam atribuir com maior brevidade e eficácia a etiqueta final das *slices*.

## 3.1 Visão Geral

O avanço tecnológico trouxe consigo muitos benefícios, impulsionando uma revolução na forma como interagimos e trabalhamos no mundo digital. Contudo, este progresso além de benefícios, trouxe consigo algumas preocupações, que se enquadram no tema deste trabalho, nomeadamente a descoberta de novas vulnerabilidades nas linguagens de programação e código-fonte das aplicações.

As linguagens de programação também evoluíram com a tecnologia, por um lado oferecendo mais funcionalidades e bibliotecas, de modo a auxiliar o programador no desenvolvimento de aplicações. Por outro lado, aumentou também a superfície de ataque, criando novas oportunidades de ataque por agentes que possam explorar novas vulnerabilidades. Grande parte dessas vulnerabilidades descobertas não resultam apenas da própria linguagem, mas através do modo como os programadores as usam no desenvolvimento de aplicações, pois podem introduzir erros de código que podem ser explorados por outras pessoas.

Assim, de forma a minimizar ou mitigar este risco surge a necessidade de criação de ferramentas que automatizem o processo de deteção de vulnerabilidades. Essas ferramentas podem ser implementadas recorrendo a modelos construídos com técnicas de aprendizagem automática.

Os modelos de ML dependem de um *dataset* eficiente e de qualidade, uma vez que é com base nele que o seu treino é realizado. Este treino permite aos modelos de ML aprenderem com

padrões e comportamentos, possibilitando assim a descoberta de vulnerabilidades presentes no código de forma mais célere e apurada.

É com esse objetivo, a construção de um *dataset* sólido e de qualidade, composto por *slices* de código PHP bem etiquetadas, que surge o BugSpotting 2.0.

Foi utilizado o PHP como linguagem de programação presente nas *slices* que serão apresentadas na aplicação, pois esta é comumente usada no desenvolvimento e construção de páginas *web* dinâmicas.

Esta aplicação *web* é uma evolução da versão anterior BugSpotting 1.0, descrita na Secção 2.7, trazendo consigo melhorias ao nível dos algoritmos, procedimentos e *workflow*, mas seguindo o mesmo objetivo descrito anteriormente.

Esta plataforma *web* permite a criação um *dataset* recorrendo ao uso de duas metodologias para classificação de *slices*, uso de ferramentas *Static Application Security Testing* (SAST) pois providenciam uma forma rápida de análise de código procurando vulnerabilidades, mas em contrapartida estas podem produzir muitos falsos positivos e negativos. Assim é usada também a técnica de *crowdsourcing*, recorrendo ao conhecimento humano com vista a uma diminuição de eventuais classificações erradas.

Usando estas duas metodologias é esperado que se consiga gerar um *dataset* confiável e fidedigno, permitindo aos modelos de ML após aprendizagem, apontar com maior precisão vulnerabilidades presentes no código.

Assim, estão presentes dois tipos de classificadores na aplicação, ferramentas SAST e *crowdsourcing*, ambos possuindo uma reputação que é influenciada e atualizada de forma automática à medida que estes classificam *slices*, até que estas cheguem ao estado de consenso ou atingirem limite de classificação, dando hipótese para que outras *slices* sejam classificadas.

Como referido anteriormente, após a obtenção da classificação de *slices*, é atribuída uma reputação ao utilizador de acordo com a experiência dele, no sentido de obter uma boa classificação, sendo que essa reputação terá interferência no resultado da etiqueta final.

## 3.2 Arquitetura

Nesta secção é feita uma explicação sobre a arquitetura do BugSpotting 2.0, a qual é a mesma da versão anterior. Posteriormente são explicados os padrões de *design* utilizados com o objetivo de organizar e estruturar esta aplicação. Por fim são explicadas a *Application Programming Interface* (API) e tecnologias usadas na aplicação.

A arquitetura do BugSpotting combina os padrões *Model-View-Controller* (MVC) e *Model-View-ViewModel* (MVVM) com o objetivo de aproveitar e aliar o ”melhor dos dois mundos”.

O BugSpotting 1.0 e 2.0 utilizam uma arquitetura de três camadas, também conhecida como arquitetura cliente-servidor, pois é um modelo de desenvolvimento de *software* que tem como objetivo dividir a aplicação em três camadas distintas. Assim torna possível obter um código mais organizado, possibilitando uma manutenção evolutiva

### 3.2.1 Arquitetura de três camadas

A arquitetura de três camadas é comumente utilizada no desenvolvimento de *software* e aplicações *web*, com o objetivo de organizar priorizar e dar responsabilidades diferentes às três camadas. Estas são organizadas em três níveis distintos, cada um com uma função específica, tornando assim a aplicação mais organizada, fácil de entender e com manutenção evolutiva simples.

As três camadas são, apresentação (*frontend*), lógica do negócio (*backend*) e camada de dados (base de dados).

A camada de apresentação consiste na *interface* que é apresentada ao utilizador da aplicação e na qual este interage, tendo como responsabilidade mostrar dados, como por exemplo *slices*, bem como guardar a ação tomada. Ou seja, o objetivo desta camada consiste não só apenas na exibição da *interface* ao utilizador, mas também guardar a informação resultante da sua interação com a aplicação. Esta camada é executada através um *browser* utilizando tecnologias como *HyperText Markup Language* (HTML), *Cascading Style Sheets* (CSS) e *JavaScript* (JS).

A camada lógica de negócio é o *core* ou cérebro da aplicação, pois é esta camada que faz o sistema funcionar, utilizando para isso cálculos ou algoritmos. É nesta camada que são processadas as ações escolhidas pelo utilizador na camada de apresentação.

A camada de dados permite guardar ou persistir toda a informação que circula na aplicação.

A interação destas três camadas pode-se resumir com o seguinte exemplo: um utilizador pede uma *slice* para classificar através da camada de apresentação, esta por sua vez envia essa informação para a camada lógica de negócio que por seu turno a processa, pedindo assim se necessário à camada de dados informação que esteja na base de dados para entregar ao utilizador.

Temos várias vantagens ao usar a arquitetura de três camadas:

- **Organiza**- Cada camada tem a sua responsabilidade;
- **Mantêm**- Cada alteração numa das camadas não influencia as outras;
- **Escalável**- É possível expandir ou atualizar uma camada sem interferir nas outras.

### 3.2.2 Padrões MVC e MVVM

O padrão *Model-View-Controller* é bastante utilizado nos dias de hoje na construção de aplicações [17], tendo como objetivo a separação das mesmas em três módulos, sendo eles *Model, View, Controller*.

O módulo *Model* é responsável pela gestão dos dados. Guarda e retorna as entidades ou tabelas da base de dados, e contém a lógica implementada pela aplicação.

O módulo *View* é responsável pela apresentação dos dados provenientes do módulo *Model*.

O módulo *Controller* permite que os módulos *Model e View* trabalhem juntas. O *Controller* recebe uma requisição do cliente, invocando o *Model* para que este a trate através das operações necessárias, enviando os dados posteriormente para a *View*. A *View* formata os dados de forma a que estes sejam apresentados corretamente ao utilizador, na aplicação *web* com o formato HTML. O padrão MVC está representado na Figura 3.1 [17].

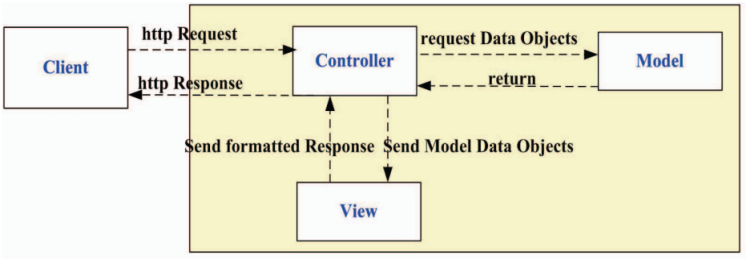



Figura 3.1: Padrão MVC

O padrão MVVM pertence à família de padrões MV, mas tem como principal foco no desenvolvimento de *interfaces* de utilizador, permitindo interações mais complexas e atualizações frequentes vindos da *interface*.

Este padrão é composto por três elementos principais, *Model*, *View* e *ViewModel*. A *View* exibe os dados provenientes do *ViewModel*, que é independente da lógica de negócio. O *Model* acede aos dados, e posteriormente envia os mesmos para a camada *ViewModel* processar. O *ViewModel* é o *core* do padrão, pois atua como ponte entre o *Model* e a *View*, processando os dados que circulam, aplicando regras de negócio, para que a *View* os possa apresentar. O *ViewModel* não manipula diretamente os componentes da *interface, View*, apenas fornece propriedades e comandos para os quais a *View* está vinculdada.

A interação entre componentes ocorre através de *data binding*, pois as mudanças no *ViewModel* são refletidas automaticamente na *View*. As interações do utilizador na *View* são encaminhadas para o *ViewModel*, que atualiza quando necessário o *Model*. Por fim, as alterações no *Model* notificam o *ViewModel*, que é necessário atualizar a *View*.

A principal vantagem consiste na separação entre a *interface* e a lógica de negócio, mas a camada *ViewModel* pode tornar-se complexa, pois gere toda a lógica, processamento de dados e comunicação entre *Model* e *View*.

Em ambas as versões do BugSpotting, os padrões MVC e MVVM são combinados, ou seja, existe um padrão híbrido. No lado do *frontend* é implementado o padrão *View-ViewModel*, onde os componentes gerem tanto a lógica da *interface* como a aplicação. Por outro lado, no lado do *backend*, este baseia-se no padrão *Model-View-Controller*, mas sem a camada *View*, funcionando como uma *Application Programming Interface* (API). O controlador processa os pedidos e delega a responsabilidade no *Model*, que lida com as regras de negócio e gestão de dados. A comunicação com o *frontend* funciona recorrendo ao formato *JavaScript Object Notation* (*JavaScript Object Notation* (JSON)).

### 3.2.3 API BugSpotting

A API do BugSpotting 1.0 e 2.0 foi desenvolvida seguindo os princípios de arquitetura *Representational State Transfer* (REST), garantindo uma comunicação eficiente entre o *frontend e backend*.

REST é um estilo de arquitetura que segue alguns princípios tais como, *interface* uniforme,

pois define um padrão de comunicação entre cliente e servidor garantindo consistência e simplicidade. É um sistema sem estado, ou *Stateless*, pois cada pedido do cliente para o servidor apenas deve conter as informações necessárias para ser processado, sem depender de qualquer contexto guardado no servidor [2].

REST tem outros benefícios tais como sistema em camadas onde cada camada não conhece os detalhes das outras, garantindo escalabilidade.

## 3.3 Algoritmos do processo para atribuição de etiquetas

Esta secção está dividida em três partes. Primeiro, na Secção 3.3.1, são explicados os conceitos e variáveis utilizados no BugSpotting 2.0, para uma fácil compreensão quanto aos algoritmos e cálculos. Segundo, na Secção 3.3.4, são descritos os algoritmos utilizados na aplicação.

### 3.3.1 Conceitos e variáveis

Os algoritmos do BugSpotting 2.0 recorrem a um conjunto de variáveis, sendo elas descritas pela ordem em que são declaradas e utilizadas ao longo da aplicação *web*, tanto na fase inicial de reputação do utilizador como na fase posterior à mesma.

#### 3.3.1.1 Variável classifierId

Esta variável armazena o número de identificação único de cada utilizador ou classificador na aplicação, de modo a que sempre que seja feita uma classificação, esta estará associada ao utilizador em questão.

#### 3.3.1.2 Variável vulnerabilityClassId

Esta variável armazena a classe de vulnerabilidade Classe de Vulnerabilidade (VC) escolhida pelos utilizadores no momento em que pedem uma *slice* para classificação.

Mais especificamente, esta variável guarda o valor da VC $z$, escolhida pelo utilizador $x$ para a *slice* $y$. Utilizaremos este padrão ao longo de todo o documento $x - y - z$, utilizador-*slice*-VC.

Atualmente o BugSpotting suporta oito VC's, pelo que o valor desta variável situa-se entre um e oito. Estas VC's encontram-se sumarizadas na Tabela 3.1.

#### 3.3.1.3 Variável phaseId

Esta variável contém o valor referente à fase atual do utilizador, permitindo saber quais as *slices* de reputação que ele já classificou e quais faltam da fase inicial de reputação.

#### 3.3.1.4 Variável label

Esta variável contém o valor referente à classificação dada pelos utilizadores à *slice* que acabaram de classificar, podendo ter o valor 0 ou 1, i.e., "não vulnerável" ou "vulnerável", respetivamente.

| Vulnerabilidade | Descrição |
|---|---|
| SQL Injection | SQL Injection (SQLi) é um tipo de ataque que explora vulnerabilidades nas aplicações web utilizando injeção de código SQL ou queries maliciosas nas consultas à Base de dados. O atacante executa este tipo de ataques com o objetivo de roubar dados ou até assumir o controlo do servidor. |
| Cross-Site Scripting | Cross-site scripting (XSS) é uma vulnerabilidade de segurança nas aplicações web, que permitem aos atacantes injetar código Javascript (JS) malicioso nas páginas dos utilizadores. Assim que o website é acedido, o código malicioso JS é executado, podendo roubar dados ao utilizador, como por exemplo cookies de sessão. Existem três tipos de ataques XSS, stored XSS (persistente), reflected XSS (não persistente) e DOM-based XSS. |
| Remote File Inclusion | Remote File Inclusion (RFI) é uma vulnerabilidade de segurança nas aplicações web, que permite ao atacante incluir ficheiros remotos ou executar código nas aplicações web vulneráveis. Ou seja, o atacante poderá incluir hiperligações de websites que este controla, e onde tem o código alojado, e assim, a aplicação web incluirá este código malicioso e posteriormente poderá será executado. |
| Local File Inclusion | Local File Inclusion (LFI) é uma vulnerabilidade de segurança nas aplicações web, idêntica ao RFI, mas neste caso o atacante apenas poderá incluir ficheiros que já estão presentes localmente no servidor. |
| Directory Traversal | Directory Traversal ou Path Traversal, também é uma vulnerabilidade que afeta aplicações web que tem como base os dois ataques anteriores, RFI e LFI, e permite que o atacante aceda a ficheiros presentes no servidor, fora da diretoria atual, introduzindo caminhos para ficheiros críticos no sistema, roubando posteriormente dados. |
| Source Code Disclosure | Source Code Disclosure (SCD) é uma vulnerabilidade de segurança que consiste na obtenção do código fonte da aplicação web e assim que é exposto, permite aos atacantes aceder e analisá-lo, procurando vulnerabilidades. Posteriormente poderão expor informações confidenciais. |
| PHP Code Injection | PHP Code Injection é uma vulnerabilidade de segurança que ocorre quando o atacante insere código PHP malicioso na aplicação web, e posteriormente este poderá ser executado, permitindo-o controlar a aplicação. As vulnerabilidades também poderão estar presentes nas frameworks ou bibliotecas da aplicação. |
| OS Command Injection | OS Command Injection ou Shell Injection é uma vulnerabilidade de segurança que permite ao atacante executar comandos no sistema operativo (SO) no qual a aplicação web está inserida. Este tipo de ataques acontece devido ao facto da aplicação web não verificar adequadamente o input dado pelos utilizadores, e assim o atacante injeta código malicioso que é interpretado como comandos legítimos do SO. |

Tabela 3.1: As oito classes de vulnerabilidades contempladas pelo BugSpotting 2.0

Numa fase inicial, após o registo de um novo utilizador, este entra em fase preparatória de cálculo de reputação. Nesta fase, o utilizador procede à classificação de três *slices* das quais é conhecida a sua etiqueta (variável $correctLabel$). Para cada *slice* que o utilizador classifica é guardado o valor da sua classificação, e determina-se se acertou ou não na etiqueta da *slice*. As *slices* de reputação, explicadas na Sub-secção 3.3.1.5, já possuem uma etiqueta final atribuída (variável $correctLabel$), pois servem apenas para testar o conhecimento de cada utilizador, e atribuir-lhe assim uma reputação.

Assim que os utilizador atribuem uma classificação à *slice* de código, esta é posteriormente guardada na variável $label$, depois o sistema verifica se naquela fase de reputação o valor é igual ao da $correctLabel$, se sim aumenta a sua reputação, caso contrário terá uma reputação inferior.

Após o término da fase de classificação das três *slices* de reputação, é iniciado outro ciclo de classificação de *slices*, aquelas em que não é sabida a sua etiqueta final. Chegar a esta etiqueta final é o nosso objetivo, deste modo, nesta fase as variáveis $correctLabel$ e $sliceReputDifficulty$ deixam de fazer sentido.

Nesta fase, o valor desta variável é uma decisão ou escolha do utilizador que está a classificar a *slice*. É expectável que o seu valor se reflita na etiqueta final e que convirja para "vulnerável" ou "não-vulnerável" à medida que forem sendo recebidas mais classificações, podendo, contudo, existir casos patológicos em que tal não suceda (e.g., metade dos utilizadores considera "vulnerável" e a outra metade "não-vulnerável").

#### 3.3.1.5 Variável correctLabel

Esta variável contém o valor atribuído à classificação das *slices* de reputação, ou seja, já possui um valor para a sua etiqueta final seja esta 0 ou 1, consoante "não vulneráveis" ou "vulneráveis", respetivamente.

Assim, à medida que são classificadas as *slices* de reputação, por parte dos utilizadores sabemos logo se eles acertaram ou não na classificação da *slice*, atribuindo mais ou menos reputação para cada caso.

O objetivo desta fase é adquirir essa informação adicional acerca do conhecimento de cada utilizador, pois assim cada um deles terá de classificar as três *slices* de reputação fica com valor final atribuído. Estas *slices* perfazem um total de três, contribuindo juntas para a reputação inicial do utilizador, considerando que esta fase preliminar é essencial para nivelar todos os utilizadores da aplicação BugSpotting.

Sempre que um utilizador classifica cada uma das três *slices* de reputação é feita uma verificação entre duas variáveis, a $label$ ou classificação dada pelo utilizador (vulnerável ou não vulnerável) e a variável $correct_label$ que é a etiqueta final daquela *slice*. No fim é feita a média das três *slices* classificadas e guardado o valor na variável $init_reputation$, explicada na Sub-Secção 3.3.1.8.

Estas *slices* de reputação já possuem um valor pré-definido e compreendido entre 0 e 1 (não vulnerável ou vulnerável), pois já foram classificadas anteriormente, sendo apenas usadas para a fase de reputação inicial.

#### 3.3.1.6 Variável confidenceDegree

No momento em que são classificadas as *slices*, tanto na fase de reputação como na fase pós-reputação, os utilizadores, mesmo os mais experientes podem ter dúvidas na classificação que estão a atribuir. Esta variável permite guardar um valor referente ao grau de confiança que estes atribuem à sua classificação da *slice* $y$.

Deste modo, pretende-se que a variável tenha valores compreendidos no intervalo definido entre 0.1 e 1, pois se o intervalo começar em 0 poderá complicar a utilização da variável em operações de divisão ou multiplicação. O valor 0.1 foi escolhido pois é perto de zero e bastante baixo a nível de confiança.

Se um utilizador indicar o valor 1, conclui-se que está totalmente confiante com a classificação dada. Por outro lado, se este associar o valor 0.1, indica um elevado grau de incerteza, pois é o mínimo. Para conseguir chegar a este valor, o utilizador, depois de analisar a *slice* de uma determinada VC, atribui o valor mais apropriado àquela classificação, tendo por base o seu conhecimento

atual sobre vulnerabilidades.

#### 3.3.1.7 Variável sliceReputDifficulty

As *slices* de reputação são definidas como fáceis, intermédias ou difíceis, tendo em consideração o seu grau de complexidade.

Numa fase inicial, a complexidade é medida através do número de linhas de código que tem a *slice*.

Decidimos, assim atribuir o valor de 0.2 a uma *slice* fácil, 0.5 a uma *slice* intermédia e 0.8 a uma *slice* difícil. A primeira *slice* a entregar aos classificadores é a fácil, depois a média e por fim a difícil, em termos de dificuldade.

Uma vez que as *slices* não são todas iguais, foi escolhido um método que as diferencia, atenta a existência de dificuldades diversas entre elas. Foi, assim, criada esta variável, que está diretamente relacionada com o grau de dificuldade da *slice* $y$.

Importa referir que esta variável apenas é utilizada na fase inicial de reputação, fase em que são apresentadas ao utilizador as três *slices* de reputação.

#### 3.3.1.8 Variável initReputation

Esta variável presente na tabela $classifiers$ verifica se o utilizador já possui ou não uma reputação inicial atribuída, $phaseId$ (usada antes de começar a classificar as três *slices* de reputação).

Esta variável tem valor 0 inicialmente e assim que cada utilizador termina a fase de reputação é feita a média dos valores obtidos pelas três *slices* de reputação. Esta variável corresponde ao resultado do cálculo da reputação inicial do utilizador, representado na Fórmula 3.1, em que foi dado um peso de 0.2 ao grau de confiança, 0.5 à sua classificação e classificação correta e por fim 0.3 ao grau de dificuldade da *slice*. Esta fórmula é aplicada três vez para cada utilizador, que corresponde a cada uma das três *slices* de reputação. Estas quatro variáveis, $confidenceDegree_y$, $sliceReputDifficulty_y$, $correctLabel_y$ e $label_y$ estão presentes na tabela $sliceReput$.

$$\begin{aligned} initReputation_y = {} & 0.2 \cdot confidenceDegree_y \\ & + 0.5 \cdot (\oplus(label_y \cdot correctLabel_y)) \\ & + 0.3 \cdot sliceReputDifficulty_y \end{aligned} \tag{3.1}$$

Para calcular um valor para esta variável foram consideradas as seguintes variáveis:

- Grau de confiança do utilizador para a *slice* $y$ ($confidenceDegree_y$);
- Grau de dificuldade atribuído para a *slice* $y$ ($sliceReputDifficulty_y$);
- Resultado da classificação da *slice* de reputação $y$, previamente classificada como correta ($correctLabel_y$);
- A classificação dada pelo utilizador à *slice* de reputação $y$ ($label_y$);

- Quando é feita a multiplicação entre a variável $label_y$ e $correctLabel_y$ é feito o XOR ($\oplus$) ou inversão do valor caso uma delas ou ambas tenham valor zero, pois iria dar desvantagem ao utilizador que classifique uma *slice* como não vulnerável e na verdade é mesmo, mas obteria zero na classificação quando multiplicado por 0.5, o que não seria justo. A classificação da *slice* de reputação foi definida como vulnerável ou não vulnerável com base na classificação dada por ferramentas de inteligência artificial (IA). Uma vez que é complicada a procura de *slices* devidamente classificadas na *web*, e de modo a solucionar este problema, foram utilizadas ferramentas de IA para resolver as vulnerabilidades nas *slices*. As ferramentas IA utilizadas foram, *ChatGPT da OpenIA, Gemini da Google e Copilot da Microsoft*.

### 3.3.2 Processo configuração BugSpotting

O perfil de administrador é responsável pela gestão e configuração de todos os dados armazenados na BD do BugSpotting. Este administrador é responsável pela inserção de *slices* na BD, permitindo que sejam posteriormente classificadas pelos utilizadores e ferramentas SAST.

As principais tarefas do administrador incluem:

- Inserir/remover *slices* de reputação;
- Inserir/remover *slices* para as quais desejamos obter uma etiqueta;
- Adicionar/remover ferramentas SAST no sistema.

Este algoritmo ou processo é responsável por realizar todas estas tarefas, cujos procedimentos que o administrador pode adotar estão apresentados na Figura 3.2.

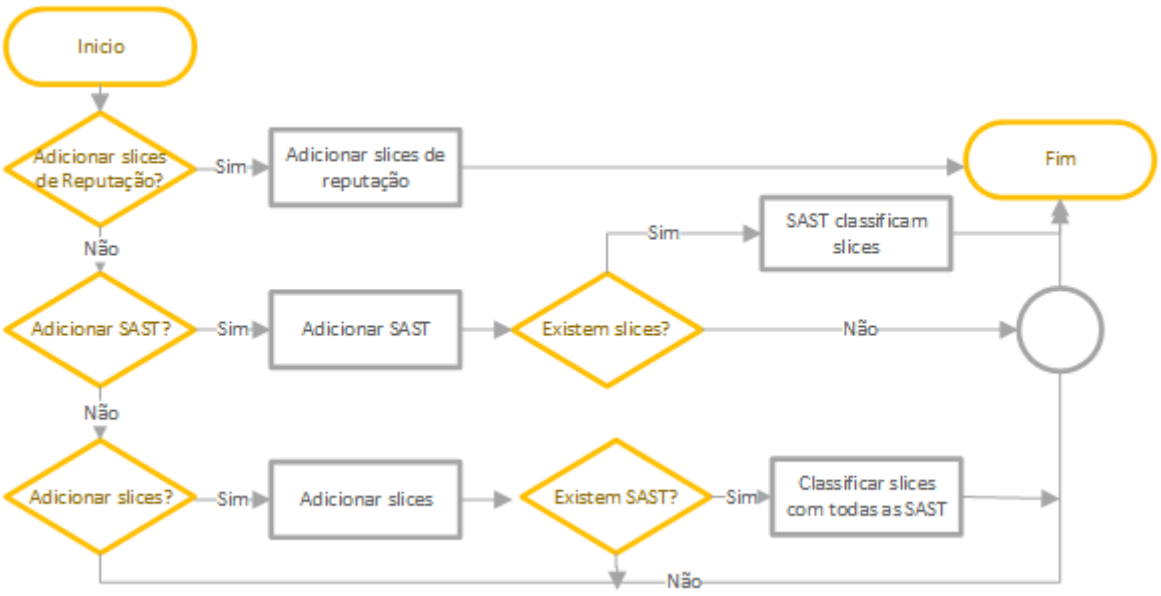



Figura 3.2: Diagrama da configuração do sistema

Assim que as SAST são introduzidas na BD, executam de imediato a sua classificação para todas as *slices*, nas oito VC's, antes destas serem entregues aos utilizadores para classificação.

O algoritmo utilizado para entregar *slices* aos utilizadores e ferramentas SAST está explicado na sub-secção 3.3.4.1.

### 3.3.3 Execução da ferramenta SAST

É uma função que é executada após as fases *ObterSliceNaoClassificada, ObterSliceMenosClassificacoes e ObterSliceMaiorIndecisao* tendo como *input* ou argumento a *slice* já filtrada. Se a ferramenta SAST já tiver efetuado a sua classificação da *slice* recebida, esta já não a classifica e essa *slice* é entregue ao utilizador, caso contrário a *slice* só é enviada para o utilizador após a SAST a classificar.

A função *ExecuteToolAndCreateClassification* é uma etapa que corresponde a um processo de análise, ou seja, primeiro recebe uma *slice* já previamente filtrada, depois analisa e verifica se já a classificou, se sim encaminha para o utilizador, senão classifica-a e guarda na BD, e posteriormente envia para o utilizador.

É importante salientar que quando o administrador insere *slices* na BD, e caso existam ferramentas SAST, estas classificam logo todas as *slices* já inseridas.

### 3.3.4 Algoritmos

Diversas melhorias podem ser implementadas ao longo do tempo com o objetivo de otimizar a qualidade do *dataset*, que contém as etiquetas finais das *slices* classificadas, e para que tal aconteça torna-se necessário chegar a um consenso entre classificadores de *slices*.

Atualmente o BugSpotting integra oito VC's, permitindo cada utilizador selecionar uma delas para dar início ao processo da fase de reputação. Quando esta fase é concluída, dá-se início à classificação e avaliação de *slices* ainda não etiquetadas.

Neste contexto é essencial decidir qual *slice* se deve entregar aos utilizadores, e para apoiar este processo, foram desenvolvidos algoritmos específicos, detalhados nas sub-secções seguintes, que auxiliam desde a seleção das *slices* até à obtenção das respetivas etiquetas finais.

#### 3.3.4.1 Obter *slice*

Este algoritmo tem como principal objetivo entregar ao utilizador uma *slice* para que este a classifique. Ocorre um processo de filtragem e execução da ferramenta de análise estática para todas as *slices*, antes da entrega da *slice* ao utilizador. Ou seja, após inserção de *slices* pré e pós-fase reputação por parte do administrador, torna-se necessária a sua entrega aos utilizadores quando as solicitam para classificação. É neste sentido que foi criado este algoritmo, obter *slice*, representado através de um diagrama na Figura 3.3 e no respetivo Algoritmo 1.

Esta função recebe como argumento um identificador único, *userId*, e um identificador de classe de vulnerabilidade, *vulnerabilityClassId*. Primeiro, é feita uma verificação sobre a existência do utilizador, caso exista começa o processo de obtenção de *slices*, caso contrário retorna valor *null*.

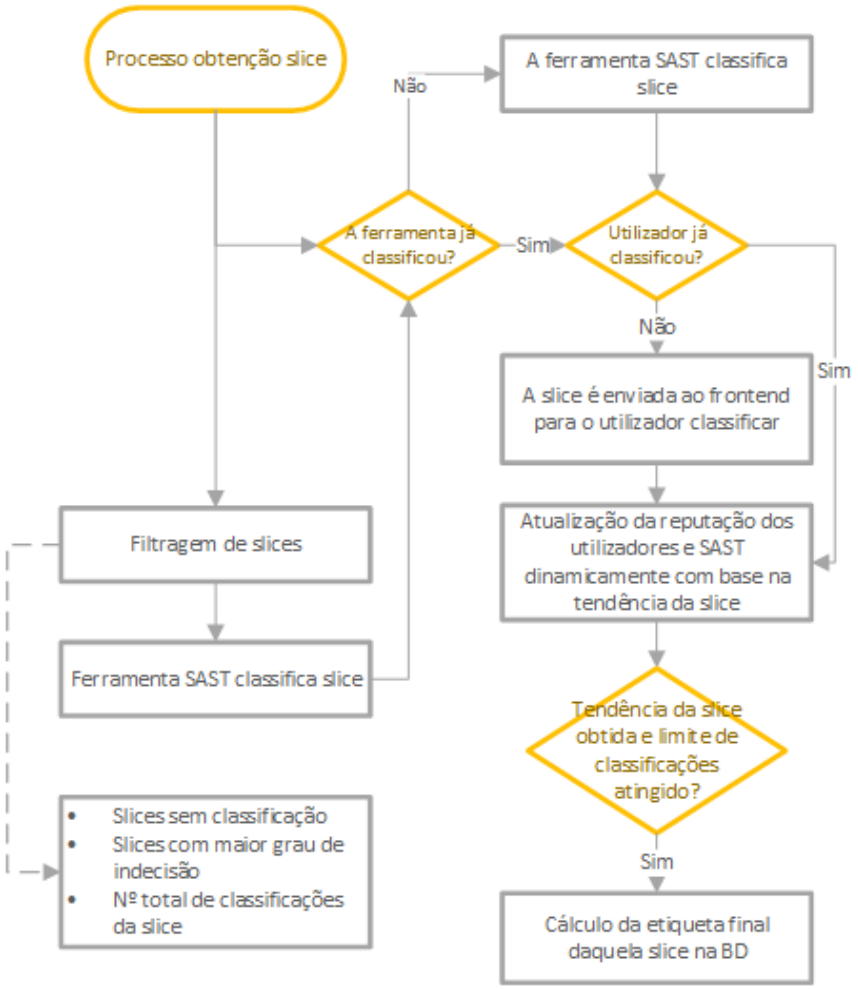



Figura 3.3: Diagrama obter slice

Posteriormente, antes de obter uma *slice* da BD, é executada uma das seguintes fases até obter uma *slice*, *GetUnclassifiedSlice, GetSliceWithFewestClassifications e GetSliceWithMoreIndecisionValue*, representadas no Algoritmo 1. No caso de nenhuma destas fases retornar uma *slice* é interrompido este processo. Caso uma destas fases retorne uma *slice*, como podemos verificar no Algoritmo 1 entre as linhas 10 e 16, esta é classificada pela ferramenta SAST, guardada na BD e enviada para o utilizador, para que este a classifique posteriormente.

Este algoritmo tem como objetivo determinar qual a próxima *slice* a entregar ao utilizador para classificação, baseando-se em diferentes fases ordenadas, pois caso uma delas não retorne uma *slice*, o processo continua e avança para a fase seguinte. As fases são as seguintes:

- **Obter *slice* ainda não classificada** - Algoritmo 1, linha 10

  O objetivo é obter uma *slice* que o utilizador ainda não tenha classificado.

  Este procedimento é executado após o utilizador indicar no *frontend* que deseja classificar uma *slice* de uma determinada VC.

  Esta função não recebe argumentos como *input* e tem como *output* ou retorno a pri-

meira *slice* de entre as que ainda não tenham sido classificadas, senão retorna *null*.

Posteriormente, é feito um pedido do *frontend* ao *backend* para que seja executada a ferramenta SAST. Assim que o utilizador escolhe uma VC para classificar é executado o algoritmo obter *slice*, e é-lhe retornada uma *slice* que ele ainda não tenha classificado, da BD.

- **Obter *slice* com menor número de classificações** - Algoritmo 1, linha 12

  O objetivo deste algoritmo é selecionar a *slice* com menor número de classificações dentro de uma determinada classe de vulnerabilidade. A função é chamada quando todas as *slices* já foram classificadas pelo menos uma vez.

  É feita uma verificação inicial que consiste em verificar se a função 1 na linha 10 retornou *null*, que indica que não existem *slices* sem classificação para aquela VC.

  Esta função pode retornar *null* perante as seguintes situações em que, todas as *slices* têm o mesmo número de classificações (máximo ou *threshold*), não existindo assim uma com menor número de classificações. Outra situação surge quando todas as *slices* já tenham sido classificadas pelo utilizador para a VC atual, e já não existam *slices* disponíveis com menos classificações. Uma outra situação que poderá acontecer é caso a lista de *slices* candidatas esteja vazia e não retorne nenhuma, devido à exclusão de *slices* já completamente classificadas.

  Esta função recebe dois argumentos, *userId* e *vulnerabilityClassId*, e antes de entregar uma *slice* ao utilizador é chamada uma nova função que seleciona a *slice* ideal a entregar.

  Este processo de seleção chama-se *GetCandidateSlices* e retorna uma lista de *slices* candidatas para entrega ao utilizador, que, se estiver vazia retorna *null*, caso contrário é calculado o número máximo de classificações entre as *slices* candidatas. Se todas as *slices* obtiverem o mesmo número de classificações é retornado *null*, caso contrário seleciona a *slice* com menor número de classificações. Como a lista já está ordenada por número de classificações em ordem crescente, basta retornar a primeira *slice*.

  Esta função primeiro obtém o limite máximo de classificações por VC, o *threshold* configurado no servidor. Posteriormente para construção de *slices* candidatas verifica quais o utilizador já classificou por VC, e vai criando uma lista de exclusão com as *slices* já classificadas para aquela VC e as que já atingiram o *threshold*, e exclui estas. Assim garante que só as *slices* que são retornadas têm menos classificações que o *threshold* definido.

  Se o utilizador já tiver classificado aquela *slice* daquela VC, esta não voltará a aparecer. Não obstante, poderá surgir para outra VC.

- **Obter *slice* com maior grau de indecisão** - Algoritmo 1, linha 15

  O objetivo deste algoritmo consiste na entrega da *slice* com maior grau de indecisão dado pelos classificadores anteriores, baseando-se nos cálculos que incluem as reputações dinâmicas e linha de tendência a cada momento temporal.

**Algorithm 1** Obter Slice

```
1:  Input: userId, vulnerabilityClassId
2:  Output: Retorna uma slice para classificação
3:  function GETSLICE(userId, vulnerabilityClassId)
4:      userId ← ProcuraUtilizador(userId)
5:      if userId = null then
6:          Log "Utilizador com userId não encontrado"
7:          throw UserNotFoundException
8:          exit
9:      end if
10:     slice ← GetUnclassifiedSlice()
11:     if slice is null then
12:         slice ← GetSliceWithFewestClassifications(userId, vulnerabilityClassId)
13:     end if
14:     if slice is null then
15:         slice ← GetSliceWithMoreIndecisionValue(userId, vulnerabilityClassId)
16:     end if
17:     if slice is null then
18:         Log "A slice não existe"
19:         throw SliceDoesNotExistException
20:         exit
21:     end if
22:     ExecuteToolAndCreateClassification(slice)
23:     return slice
24: end function
```

Importa reter que este algoritmo é crucial para seleção de qual a próxima *slice* a entregar ao utilizador ou ferramenta SAST, baseando-se nos critérios explicados anteriormente.

#### 3.3.4.2 Reputação inicial

Com vista à obtenção e aperfeiçoamento do *dataset* final tornou-se necessária a criação de uma fase introdutória, a reputação inicial, que permite obter mais informação acerca do conhecimento de quem classifica as *slices*. O principal objetivo desta fase introdutória é a atribuição inicial de uma reputação a cada utilizador, pois como sabemos nem todos os utilizadores possuem o mesmo conhecimento de programação, neste caso, em PHP.

Uma vez que teremos utilizadores com menor nível de conhecimento, espera-se que a sua reputação seja inferior quando comparada com um utilizador mais experiente. Na construção do *dataset* final pretende-se qualidade, tornando-se assim necessária a atribuição de maior reputação aos utilizadores com maior conhecimento.

A fase introdutória é igualmente importante porque, com a introdução do *crowdsoursing*, será possível obter mais classificações por parte dos utilizadores, e considerando que o nível de conhecimento deles é diferente, torna-se conveniente organizá-los por reputação.

Concluída esta fase introdutória, e após organização de todos os utilizadores em diferentes níveis de reputação inicial, os mais experientes terão, à partida, uma melhor reputação do que

aqueles com menor experiência, isto num cenário ideal.

Esta reputação inicial contém três fases (variável $phaseId$) para os utilizadores classificarem, em que a cada fase é guardada a sua classificação para a *slice*. Estas *slices* possuem três níveis de dificuldade diferentes, com VC's *random*, de entre as oito disponíveis. Importa referir que todas estas *slices* de reputação já estão devidamente classificadas e etiquetadas (variável $correctLabel$), bastando esperar que a classificação dos utilizadores (variável $label$) coincida com ela.

Cada utilizador possui um grau de confiança ao classificar a *slice*, posteriormente após a primeira classificação é guardado o valor dado pelo utilizador naquela fase, verificando se coincide com a etiqueta final daquela *slice*, se coincidir terá um valor mais elevado na reputação, pois acertou, caso contrário, será penalizado. O mesmo procedimento é feito para as três *slices*, e no fim é calculada a média das três fases e é obtida a reputação inicial.

Posteriormente, entram as *slices* pós-fase de reputação, não se sabendo ainda, numa fase inicial, a sua etiqueta final, objetivo do BugSpotting, sendo utilizados assim, outros algoritmos que serão explicados de seguida.

Na Figura 3.4, é possível verificar que o processo de reputação inicial começa assim que o utilizador pede uma *slice* para classificar, mas, se este já tiver uma reputação inicial, o processo termina e o algoritmo de obtenção de *slice* encarrega-se de entregar ao utilizador uma *slice* da fase pós-reputação inicial.

Caso o utilizador não tenha ainda uma reputação inicial é dado início ao processo de entrega das três *slices* de reputação, com diferentes graus de dificuldade. Como referido, esta fase permite a atribuição de uma reputação ou valor inicial, usado para diferenciar os conhecimentos de cada utilizador.

Importa novamente referir que cada uma das três *slices* classificas têm diferentes graus de dificuldade (variável $sliceReputDifficulty$), sendo estes, fácil, médio e difícil, que correspondem a 0.2, 0.5 e 0.8, respetivamente.

Se realmente a *slice*, após classificação for vulnerável, ou seja valor igual a 1, é aplicada a Fórmula 3.2 de reputação inicial, caso contrário é aplicado o inverso e usando o $XOR$ ($\oplus$) no algoritmo auxiliar 3 nas linhas 8 e 13.

$$\begin{aligned} initReputation_y = {} & 0.2 \cdot confidenceDegree_y \\ & + 0.5 \cdot (\oplus(label_y \cdot correctLabel_y)) \\ & + 0.3 \cdot sliceReputDifficulty_y \end{aligned} \tag{3.2}$$

Após classificação das três *slices* de reputação, é atribuída a reputação inicial.

Após análise da Figura 3.4, é possível observar com maior detalhe o funcionamento do algoritmo que está descrito no Algoritmo 2.

Este algoritmo começa por receber como argumentos o *confidenceDegree, label, correctLabel, sliceReputDifficulty, userId, vulnerabilityClassId*.

**Cálculo auxiliar na reputação inicial** Ao efetuar o cálculo na fase de reputação inicial do

**Algorithm 2** Fase de reputação inicial

**Input:** confidenceDegree, label, correctLabel, sliceReputDifficulty, userId, vulnerabilityClassId
**Output:** Reputação inicial - Valor inicial de reputação
**if** $initReputation$ = 0 **then**
    O utilizador não tem reputação inicial atribuída
    **exit**
**else**
    **function** GETCLASSIFIEDSLICESUSER(userId)
        $phaseId \leftarrow$ GetClassifiedSlicesUser($userId$)
        **if** $phaseId < 3$ **then**
            return phaseId disponíveis para classificação

            **function** GETSLICEREPUT(userId, vulnerabilityClassId)
                Utilizador classifica *slice* de reputação
                $initReputation \leftarrow$ GetSliceReput(confidenceDegree$_y$, $correctLabel_y$, $label_y$, $sliceReputDifficulty_y$)

                **function** CALCULATESCORE(SliceReput)
                    **if** $label$ = $correctLabel$ **then**
                        Utilizador classificou corretamente
                        return initReputation
                        **exit**
                    **end if**
                    Utilizador classificou incorretamente
                    return initReputation
                    **exit**
                **end function**
                return initReputation
            **end function**

**Algorithm 3** CalculateScore

**Input:**$SliceReput$
**if** $label$ = $correctLabel$ **then**
    Reputação inicial para a *sliceId* atribuída ao utilizador.
    $initReput = 0.2 \cdot ReputDifficul + 0.5 + 0.3 \cdot confDeg$;
    **exit**
**else**
    **if** $label$!=0 && $correctLabel$!=0 **then**
        Reputação inicial obtida através da dificuldade, etiqueta e grau de confiança.
        $initReput = 0.2 \cdot ReputDifficul + 0.5 \cdot (label \cdot correctLabel) + 0.3 \cdot confDeg$
    **end if**

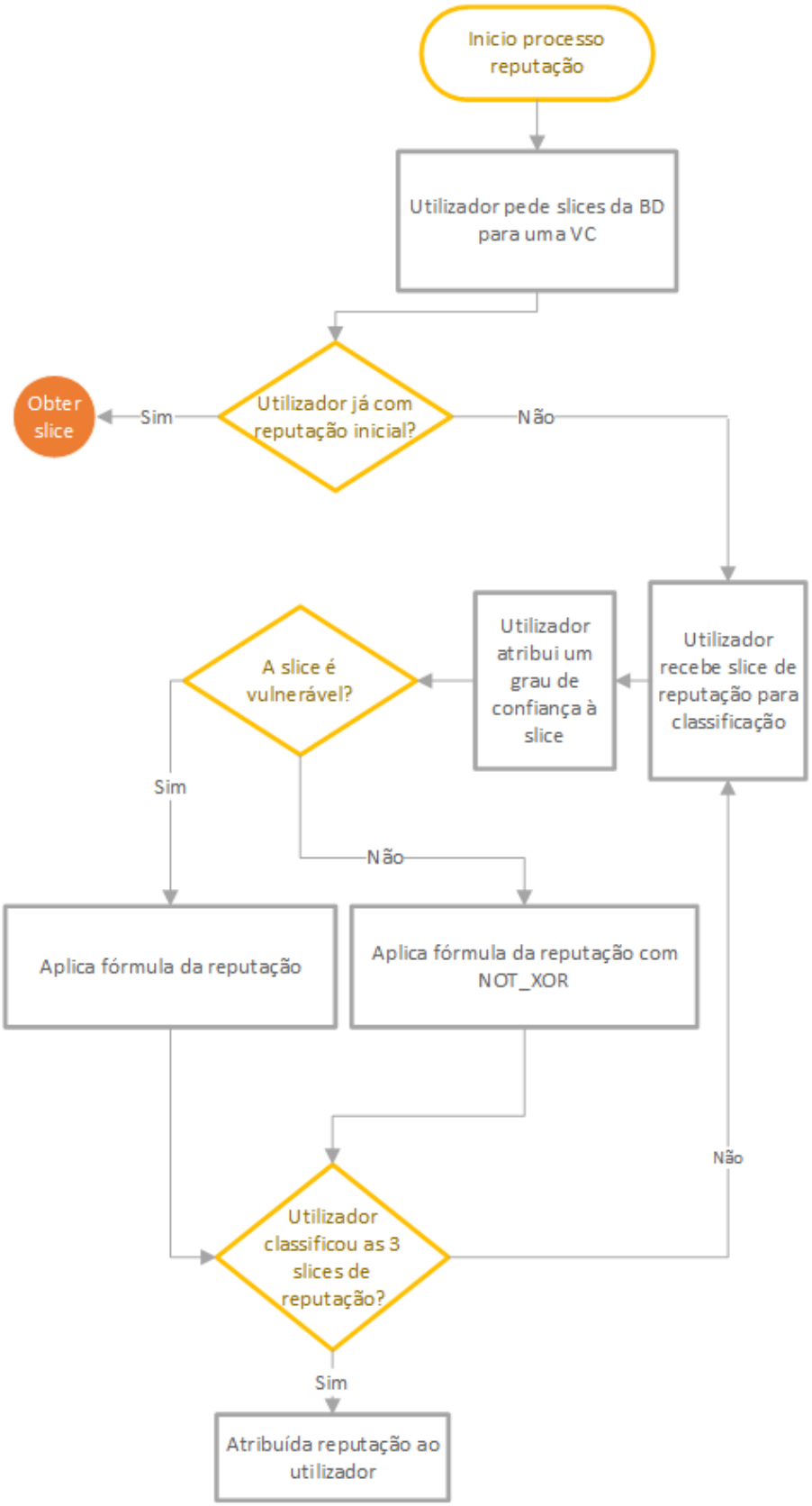



Figura 3.4: Diagrama obter reputação

utilizador a cada uma das três *slices* para classificação, este cálculo auxiliar previne que, após aplicação da fórmula, o valor das variáveis $correctLabel$ e $label$ seja nulo. Isto porque, ao multiplicar por zero, obteríamos sempre zero, o que daria desvantagem a quem classificar a *slice* como não vulnerável.

Podemos verificar este cálculo no Algoritmo 2 linha 17, utilizando para isso o Algoritmo auxiliar 3.

#### 3.3.4.3 Tendência

Este Algoritmo 4 permite calcular a tendência para onde a classificação das *slices* tende, a cada momento temporal diferente. O valor da tendência está compreendido entre 0 e 1, sendo que existe uma linha no valor 0.5 que separa a *slice* vulnerável da não vulnerável.

Assim, uma vez atribuída a reputação inicial aos utilizadores do BugSpotting, é iniciada uma nova fase, obtenção de tendência que é um parâmetro necessário para o cálculo da etiqueta final. Este valor resulta da tendência de todas as classificações já dadas e por todos os utilizadores, até ao momento.

Vejamos o seguinte panorama, um utilizador, quando classificam inicialmente uma *slice* de reputação, a aplicação *web* já tem conhecimento sobre a sua etiqueta final, atribuindo assim a reputação correspondente às classificações dadas.

Nesta nova fase, torna-se necessária a criação de novas variáveis que permitam fazer uma estimativa ou previsão quanto à etiqueta final para todas as *slices* já com classificações.

O panorama descrito anteriormente não pode ser aplicado nas *slices* pós-fase de reputação, uma vez que não se sabe a sua etiqueta final.

É neste momento que é introduzida a variável tendência, pois permite uma previsão crescente ou descendente relativamente ao número de classificações, ao longo das diferentes rondas ou momentos temporais. Para este cálculo são usadas as variáveis reputação, grau de confiança e classificação dada em conjunto com a tendência da *slice* classificada naquele momento.

Para o cálculo da tendência foram utilizadas as seguintes fórmulas, consoante seja a primeira classificação daquele utilizador para aquela *slice* (tempo=0), pois é ele que define a classificação tendencial inicial, como na fórmula 3.3:

$$tendenciaInicial_y = \frac{\sum confidenceDegree_y \cdot \sum label \cdot \sum initReputation}{\sum confidenceDegree_y \cdot \sum initReputation} \tag{3.3}$$

A partir desse tempo (0) todos os outros utilizadores, utilizam a seguinte fórmula baseada na classificação tendencial do tempo anterior, como na fórmula 3.4:

$$tendencia_y = \frac{\sum confidenceDegree_y \cdot \sum NOT \oplus label \cdot \sum dinamReput}{\sum confidenceDegree_y \cdot \sum dinamReput} \tag{3.4}$$

Para que seja possível obter a tendência inicial da *slice*, é necessário obter o valor referente ao grau de confiança dado pelo utilizador, a sua classificação ($label$) e a sua reputação inicial ($initReputation$). Isto para a primeira classificação, referente à *slice* $y$.

Posteriormente, para o cálculo da tendência atual é utilizada, para os novos utilizadores, com a sua reputação inicial, e para os utilizadores que já classificaram a *slice*, sua reputação inicial é influenciada pela tendência atual, resultando na reputação dinâmica. Resumindo, o primeiro utilizador a classificar a *slice* não é influenciado pela tendência, pois é o primeiro, e assim é ele a influencia pois é t=0. No momento temporal t=1, já temos uma tendência anterior dada pelo utilizador do t=0, que vai influenciar os novos utilizadores que classificarem aquela *slice*.

Assim para o primeiro utilizador, a sua reputação adquirida no t=0, vai ser aquela que irá ser utilizada no t=1, para o cálculo da reputação dinâmica, e não a sua reputação inicial, pois esta já foi alterada.

Para os utilizadores novos que classifiquem aquela *slice* o valor que vão utilizar é o da sua reputação inicial, mas com o valor da classificação tendencial anterior. No t=i+1, estes novos utilizadores, já obtiveram uma reputação inicial e é com ela que a sua reputação irá ser afetada. Esta explicação está representada no Algoritmo 4.

**Algorithm 4** Obter tendência

1: **Input:** userId, VulnClassId, sliceId, $confidenceDegree_y$, $label$, $initReputation$, $dinamReputation$
2: **Output:** Retorna a tendência daquela *slice*
3: O Utilizador pede uma *slice* para uma VC, t=0
4: **if** $initReputation_y$ != 0 && $tendenciaInicial_y$ = 0 **then**
5: Cálculo da tendência inicial:

$$tendenciaInicial_y = \frac{\sum confidenceDegree_y \cdot \sum NOT \oplus label \cdot \sum initReputation}{\sum confidenceDegree_y \cdot \sum initReputation}$$

6: Valor da tendência obtida para momento temporal = 0
7: Return tendenciaInicial
8: **else**
9: Cálculo da tendência t=i+1

$$tendencia_y = \frac{\sum confidenceDegree_y \cdot \sum NOT \oplus label \cdot \sum dinamReput}{\sum confidenceDegree_y \cdot \sum dinamReput}$$

10: Valor da tendência obtida para momento temporal = 0
11: Return tendencia
12: **end if**

Imaginemos o seguinte panorama, temos o utilizador 1 que classifica a *slice* 1 e VC 1, ou seja, 1-1-1, num determinado momento temporal, denominado como t=1.

Posteriormente, teremos os utilizadores seguintes a classificar a mesma *slice*, nos momentos temporais seguintes (t=2,3,4), até que seja possível obter uma *label* estável e final.

Com a continuação de classificações, por parte dos demais utilizadores, para a mesma *slice*, será possível obter uma tendência na classificação, sendo esta definida como "vulnerável" ou "não vulnerável", e é nesse sentido que o cálculo é feito.

Vejamos outro exemplo, para a mesma *slice* o utilizador, no momento 1-1-1, tem valor de 0,39 ou 39% de reputação e, após a aplicação da fórmula da tendência, a sua reputação dinâmica

aumentou para 0,85 (85%), uma vez que, estando esta classificação no início, t=2, estará longe da etiqueta final. Como temos poucos utilizadores esta é imprecisa.

Posteriormente, entra o segundo utilizador, que irá influenciar a etiqueta e gerar uma nova para aquele momento temporal t=2, como vimos anteriormente. Nos momentos temporais seguintes, a etiqueta final irá sendo alterada, até chegar a um consenso final.

Nas Tabelas 3.2 e 3.3, é possível verificar que, no momento de classificação da *slice* 1-1-1, este começou com as *slices* de reputação e adquiriu uma nova classificação dinâmica.

Este processo é contínuo e dinâmico. Assim, à medida que um utilizador $x$ se aproxima da tendência final da *slice* $x-1-1$, este aumenta ou diminui a sua reputação, de forma dinâmica, a cada momento temporal, contribuindo, também, para a etiqueta final da *slice*. A variável $sliceReputDifficulty_y$ deixa de fazer sentido neste momento, pois a fase de reputação já passou, sendo, assim, removida das fórmulas posteriores.

O utilizador 1 entra e classifica a *slice_y*. Neste caso, a VC 1, com a reputação que obteve na fase de reputação, 0,51 (51%), como é possível verificar na Tabela 3.2.

Ao entrar o utilizador 2, é possível verificar a sua classificação, referente à fase de reputação, bem como a classificação tendencial, como demonstrado na Tabela 3.3.

Nas Tabelas 3.4 e 3.6 verificámos uma nova alteração, neste caso com a mudança da reputação inicial para a dinâmica atual.

Ao entrar o utilizador 3,4,5 e 6, podemos verificar alterações nestes bem como no utilizador 1 e 2, referente à sua reputação bem como a classificação tendencial, como demonstrado na Tabela 3.6.

No tempo=8, é possível observar que, com cento e vinte utilizadores, a *slice* mostrou uma tendência para "vulnerável", como demonstrado na Tabela 3.7.

Nas tabelas seguintes é possível verificar a evolução ao longo dos momentos temporais.

Tabela 3.2: Utilizador 1 classifica a *slice* 1

| Utilizador | label | confidenceDegree | initReputation | ClassifTendencial |
|---|---|---|---|---|
| 1 | 0,5 | 0,5 | 0,507 | 1 |

Tabela 3.3: Utilizador 2 classifica a *slice* 1

| Utilizador | label | confidenceDegree | initReputation | tendency |
|---|---|---|---|---|
| 2 | 0,5 | 0,5 | 0,387 | 1 |

Tabela 3.4: Cálculo tendência tempo=1

| Utilizador | label | confidenceDegree | initReputation | tendency | Label atual |
|---|---|---|---|---|---|
| 1 | 0,5 | 0,507 | 1 | 1,000 | |

Tabela 3.5: Cálculo tendência tempo=2

| Utilizador | label | confidenceDegree | initReputation | tendency | Label atual |
|---|---|---|---|---|---|
| 1 | 1 | 0,5 | 0,387 | 1,000 | |
| 2 | 1 | 0,5 | 0,507 | 1,000 | |

Tabela 3.6: Cálculo tendência tempo=3

| Utilizador | label | confidenceDegree | initReputation | tendency | Label atual |
|---|---|---|---|---|---|
| 1 | 1 | 0,5 | 0,507 | 0,4162 | |
| 2 | 1 | 0,5 | 0,387 | 0,4162 | |
| 3 | 1 | 0,5 | 0,593 | 0,4162 | |
| 4 | 0 | 0,5 | 0,387 | 0,4162 | |
| 5 | 0 | 0,5 | 0,507 | 0,4162 | |
| 6 | 0 | 0,5 | 0,387 | 0,4162 | |

Tabela 3.7: Cálculo tendência tempo=8

| Utilizador | label | confidenceDegree | initReputation | tendency | Label atual |
|---|---|---|---|---|---|
| 1 | 1 | 0,5 | 0,507 | 0,8793 | |
| 2 | 1 | 0,5 | 0,387 | 0,8793 | |
| .. | .. | .. | .. | .. | |
| 120 | 1 | 0,5 | 0,387 | 0,8793 | |

Posteriormente, foi inserida uma *slice* 2, no panorama, e assim, foi utilizada a reputação já adquirida da *slice* anterior, a 1. Para cálculos de reputação relativamente à *slice* 2, foi utilizada a reputação dinâmica já adquirida na *slice* 1 em vez do valor adquirido da fase de reputação inicial ($initReputation$), uma vez que já adquiriu um valor de reputação atualizado, daí ser dinâmico.

Se um utilizador, na fase de reputação, obteve 0,39 e, no final da classificação da primeira *slice*, de acordo com a tendência, acertou na classificação, este vê a sua reputação aumentar para 0,80. Esta será a classificação final, naquele momento temporal, atualizada na base de dados.

O utilizador não espera que todos os outros classifiquem até obter uma nova reputação. Esta é atualizada sempre que alguém classifique a *slice*.

A partir desse momento, com entrada de novas classificações por parte dos utilizadores, as suas reputações também serão alteradas dinamicamente, usando a tendência das classificações anteriores.

Resumindo, a classificação de cada utilizador aumenta ou diminui de acordo com a classificação tendencial, até que uma determinada *slice* estabilize e chegue a um consenso, ficando, assim, decidida a sua *label* final.

Este processo de classificação final é dinâmico e contínuo, sendo que executado tanto para os utilizadores que já classificaram a *slice* atual, como para os futuros utilizadores, que também irão classifica-la.

A classificação dos utilizadores que já classificaram a *slice* também é influenciada pela maioria das classificações dos outros utilizadores, em cada momento temporal.

Como se constatou anteriormente, foi possível verificar a alteração da etiqueta final ao fim de algum tempo.

Para o demonstrar, foram feitos cálculos de reputação dinâmica, desde o tempo=1 até ao tempo=8, analisando a tendência da classificação e usando esse cálculo para influenciar a reputação dinâmica do utilizador.

Após a aplicação da fórmula da reputação dinâmica 3.3 ou 3.4, dependendo da situação, é possível calcular a reputação, em cada momento temporal, para cada utilizador. Podemos observar, nas tabelas seguintes, com t=1 na Tabela 3.11, t=2 na Tabela 3.12, t=3 na Tabela 3.13 e, por fim, na Tabela 3.14 t=8, que tem mais utilizadores, sendo que, neste momento temporal, foi possível chegar a um consenso e etiqueta final.

A tendência é uma medida utilizada para definir a etiqueta atual, para aquela *slice* e para um determinado momento temporal.

Inicialmente, quando o utilizador 1 classifica a *slice*, no momento temporal t=1, ainda não existe uma tendência anterior, pois é a primeira, e neste sentido, de modo a resolver esta situação, foi alterada a fórmula da tendência.

Outra condição aplicada na fórmula, consiste na aplicação de um $NOT_XOR$, na classificação dada à *slice*.

Na secção anterior foi explicada a necessidade de existirem vários momentos temporais, necessários para obtenção de uma tendência e, posteriormente, uma etiqueta final.

Na presente secção, são apresentados os cálculos necessários para a obtenção inicial de reputação do utilizador, nos momentos temporais anteriormente referidos.

A fórmula que atribui a reputação ao utilizador está divida entre 3 valores (grau de confiança, classificação do utilizador e o resultado final da *slice*) que podem ser alterados consoante o peso que queremos atribuir a cada variável.

A fórmula escolhida para calcular o valor da variável $initReputation$, para cada uma das três *slices*, na fase de reputação, é apresentada de seguida na Equação 3.5.

$$\begin{aligned} initReputation_y = {} & 0.2 \cdot confidenceDegree_y \\ & + 0.5 \cdot (\oplus(label_y \cdot correctLabel_y)) \\ & + 0.3 \cdot sliceReputDifficulty_y \end{aligned} \tag{3.5}$$

De modo a testar a fórmula, foi atribuído ao utilizador $x$, a verde, o nível máximo de confiança (1), máxima dificuldade da *slice* (1), em que este a classificou como "vulnerável" (1) e a *slice* de reputação era efetivamente "vulnerável" (1).

Foi feito o mesmo para o utilizador Y, a vermelho, usando a mesma *slice*, mas, desta vez, errando a classificação da *slice*, obtendo como resultado metade da classificação (0.5), como é possível observar na Tabela 3.8.

Tabela 3.8: Exemplo com dois utilizadores e duas classificações

| Utilizador | slice | label | confidenceDegree | sliceReputDifficulty | initReputation |
|---|---|---|---|---|---|
| X | 1 | 1 | 1 | 1 | 1 |
| Y | 1 | 1 | 1 | 0 | 0,5 |

Segundo a Equação 3.5 representada anteriormente, caso um utilizador falhe na classificação da *slice* ou esta seja "não vulnerável", esta variável $correctLabel$ ou $label$ iria influenciar o resultado obtido pela parte integrante da fórmula $0.5 \cdot (NOT \oplus (label * correctLabel))$, pois a

multiplicação teria sempre como resultado zero, uma vez que aconteceria $0,5 \cdot (0 \cdot 1)$, o que, matematicamente, não é possível nem correto.

Para resolver este problema, foi criado uma função que avalia se ambas as variáveis são iguais a zero e aplica o inverso (ou um $OR$ ($\|$)) fazendo com que o cálculo seja equivalente a $0,5 \cdot (1 \cdot 1)$, como visto anteriormente, mas apenas unilateral. Deste modo, basta que uma das variáveis seja verdadeira, $OR(label * correctLabel)$ e, ao aplicar o $OR$, retorna verdadeiro (1), se, pelo menos, uma das duas variáveis for verdadeira. De igual modo, se ambas forem falsas (ou a 0), aplica o $NOT\oplus$ e, ao inverter, também dá 1 como resultado.

De seguida, foram selecionados três utilizadores diferentes, mais uma vez fictícios, como é possível observar na tabela 3.9.

Tabela 3.9: Cálculo da avaliação inicial de três utilizadores com 80% de confiança

| Utilizador | slice | label | confidenceDegree | sliceReputDifficulty | initReputation |
|---|---|---|---|---|---|
| 1 | 0,8 | 1 | 1 | 1 | 0,96 |
| 2 | 0,8 | 0 | 1 | 1 | 0,46 |
| 3 | 0,8 | 0 | 1 | 0 | 0,96 |

No primeiro caso, a vermelho, temos o utilizador 1, que classificou a *slice* como "vulnerável" (1), com a máxima confiança (1) e a *slice* de reputação era mesmo "vulnerável" e obteve uma classificação de 96%. Não obteve a classificação de 100%, pois a *slice* tinha um grau de dificuldade 0,8 (ou 80%).

No segundo caso, a verde, temos o utilizador 2, que classificou a mesma *slice* como "não vulnerável" com a máxima confiança. Sendo esta "vulnerável", obteve a classificação de 46% para aquela *slice*, pois errou na classificação e foi penalizado.

Por fim, a laranja, para outra *slice*, o utilizador 3 classificou como "não vulnerável", acertando na classificação, obtendo, assim, a mesma reputação que o utilizador 1 na outra *slice*, ou seja, 96%.

Estes três utilizadores serviram de exemplo para demonstrar que a classificação dada à slice é bastante importante, fazendo parte do cálculo e sua reputação, tendo os valores sido levados ao máximo, para verificar se os seus limites máximo e mínimo estariam corretos.

Na primeira *slice* da fase de reputação, começámos com os 20 utilizadores iniciais, todos classificando a mesma *slice* com $slice$ fácil (0,2).

Cada um destes utilizadores atribuiu uma classificação à *slice*, entre 0 ou 1, sendo este valor guardados na variável $ClassifSlice$. Uma vez que as *slices* CVE ($ClassifSliceReput$) já estão devidamente classificadas, não é necessária nenhuma ação adicional por parte dos utilizadores ou ferramenta SAST, sendo esta já representada automaticamente com a etiqueta final na base de dados.

Cada um desses 20 utilizadores executa o mesmo procedimento explicado anteriormente para aquela *slice*, bem como para a segunda e terceira *slice* da fase de reputação, alterando para cada uma delas as suas classificações e grau de confiança.

Para melhor exemplificar o resultado final do cálculo das três *slices* de reputação, foram selecionados 2 utilizadores dos 19, o 6 e o 17, para as três *slices* de reputação com os respetivos

graus de dificuldade e com o mesmo grau de confiança. O utilizador 6, neste exemplo, foi o pior utilizador, pois errou as três *slices* com a máxima certeza, e pelo contrário o utilizador 17 foi o melhor pois acertou em todas, com a máxima certeza e experiência.

Podemos observar na Tabela 3.10 que se segue, a reputação final dos utilizadores 6 e 17, a vermelho e verde, respetivamente.

Existe uma nuance que faz com que o utilizador 6 no fim das três *slices* de reputação não seja o utilizador com menor reputação de todos os 20 iniciais, uma vez que o grau de confiança também entra nos cálculos e o utilizador 6 e 17 tiveram maior confiança.

Utilizadores com menor grau de confiança que classifiquem a *slice* de modo errado obtêm uma menor pontuação, em contrapartida, caso a classifiquem com maior obtém um valor um pouco mais elevado, apesar da *slice* de reputação estar classificada de forma errada.

A reputação inicial ($initReputation$) é obtida após a classificação destas três *slices* e calculada posteriormente como a média da reputação delas, e por fim esta sim é a sua reputação inicial antes de avançar e classificar a primeira *slice*, sem etiqueta definida.

Tabela 3.10: Exemplo com dois utilizadores e duas classificações

| Utilizador | Slice 1 | Slice 2 | Slice 3 | Reput |
|---|---|---|---|---|
| 6 | 0,340 | 0,400 | 0,460 | 0,400 |
| 17 | 0,840 | 0,900 | 0,960 | 0,900 |

Tabela 3.11: Reputação dinâmica tempo=1

| Utilizador | initReputation | dinamReputation | Classificação tendencial | Label atual |
|---|---|---|---|---|
| 1 | 0,507 | 0,653 | 1 | |

Tabela 3.12: Reputação dinâmica tempo=2

| Utilizador | initReputation | dinamReputation | Classificação tendencial | Label atual |
|---|---|---|---|---|
| 1 | 0,653 | 0,846 | 1 | |
| 2 | 0,387 | 0,766 | 1 | |

Tabela 3.13: Reputação dinâmica tempo=3

| Utilizador | initReputation | dinamReputation | Classificação tendencial | Label atual |
|---|---|---|---|---|
| 1 | 0,846 | 0,6386 | 0,521 | |
| 2 | 0,766 | 0,880 | 0,521 | |
| 3 | 0,593 | 0,882 | 0,521 | |
| 4 | 0,387 | 0,746 | 0,521 | |
| 5 | 0,593 | 0,828 | 0,521 | |
| 6 | 0,683 | 0,885 | 0,521 | |

Isto acontece, pois, se o utilizador 1 classificar a *slice* como "não vulnerável" (0), irá influenciar negativamente toda a fórmula (multiplicação por zero), sendo deste modo é aplicado o $NOT_XOR \oplus$.

Tabela 3.14: Reputação dinâmica tempo=8 x-1-1

| Utilizador | initReputation | dinamReputation | Classificação tendencial | Label atual |
|---|---|---|---|---|
| 1 | 0,7315 | 0,7952 | 0,91 | |
| .. | .. | .. | .. | |
| 120 | 0,387 | 0,915 | 0,91 | |

Assim, a *slice* fica classificada como "vulnerável" (1), apenas para efeitos de cálculo, sendo que, ao longo do tempo a tendência terá de demonstrar a veracidade da *label* final daquela *slice*.

É possível verificar, na fórmula seguinte 3.6, a modificação feita, de uma forma inicial, para que não seja feita a multiplicação por zero, além de outra nuance em que o grau de confiança está dentro do intervalo 0.1 (se for 0 o cálculo não funcionará) e 1.

$$tendencia_y = \frac{\sum((NOT \oplus label_y) \cdot confidenceDegree_y \cdot initReputation_y)}{\sum initReputation_y \cdot confidenceDegree_y} \tag{3.6}$$

O mesmo procedimento foi também aplicado para o momento temporal t=1 na fórmula da reputação dinâmica, como podemos verificar na Equação 3.7.

$$\begin{aligned} dinamReputation_y = {} & 0.3 \cdot (NOT \oplus label_y) \\ & + 0.3 \cdot confidenceDegree_y \\ & + 0.4 \cdot initReputation \end{aligned} \tag{3.7}$$

Quanto ao cálculo da tendência e o seu respetivo somatório, caso o utilizador classifique a *slice* como "não vulnerável" (0) o resultado do cálculo iria baixar e penalizar muito a sua reputação dinâmica, bem como a tendência anterior. Assim, caso a *slice* seja classificada como 0, ao valor é aplicado o $NOT_XOR \oplus$, ficando, assim, com o valor invertido, 1.

Ao longo dos próximos momentos temporais ou rondas, o verdadeiro valor da tendência ou *label* final será revelado. A ideia a reter é inverter a classificação de 0 para 1, para não afetar negativamente o cálculo ao criar a tendência. Nos momentos temporais seguintes, já não será aplicado o $NOT_XOR \oplus$.

Caso um utilizador classifique a *slice* como "não vulnerável" (0), qualquer valor multiplicado por 0 irá dar 0. Se for um bom utilizador, com boa reputação, não queremos que dê 0, como é óbvio, nem devemos dar um valor (por exemplo 0.5), pois pode estar certo na classificação que fez.

Decidimos, então, que, se o utilizador classificar como "não vulnerável" (0), o valor deve inverter para 1, pois pode estar correto, devendo, assim, entrar no cálculo. Se, porventura, não estiver correto, não haverá problema pois, no futuro a tendência demonstrará essa previsão na *label* final da *slice*.

Os casos vulneráveis devem ser priorizados em relação às outras *slices* "não vulneráveis", pelo que nos devemos preocupar mais com a vulnerabilidade e gravidade do que com falsos negativos, devendo ser tratados como casos diferentes e distintos.

Porquê sempre a 1, é a questão, porque são sempre vulneráveis ao início as *slices*, isto porque, como referimos anteriormente, se as *slices* fossem não vulneráveis na fórmula iriam provocar uma desvantagem para todos os utilizadores que classifiquem uma *slice* que por ventura seja vulnerável, uma vez que na fórmula obteria sempre 0, pois divisões e multiplicações por 0 dá sempre 0.

Assim, todos esses utilizadores que classifiquem *slices* como não vulnerável estariam numa situação em que seriam penalizados injustamente, entrando num estado de desvantagem em relação aos demais.

No caso atual e a forma justa seria no seguinte caso, apesar de um utilizador classificar como vulnerável 1 ou não vulnerável 0 uma *slice* em que não se sabe atualmente a sua *label* final, esta leva valor de 1 na fórmula, pois se for 0, entra em desvantagem e não é justo. Por que não envolver mais cálculos e é logo o inverso de 0 a 1? Porque a fórmula da tendência e a reputação dinâmica de cada utilizador irão revelar a cada ronda e a cada momento temporal a verdadeira *label* final daquela *slice*.

Assim todas as *slices* tendem para vulnerável? Não propriamente, segundo os cálculos alguma percentagem pode tender para vulnerável mas, é importante reter se é realmente vulnerável, pois o perigoso tem de ser posto à prova do que um falso negativo que não é perigoso. Mais vale começar em vulnerável e saber que está a descer abruptamente e sabemos que realmente não é vulnerável do que o contrário.

**Fórmula da tendência de diferentes *slices* e momentos temporais**

Um momento temporal (t) pode ser definido como uma janela de tempo em que um ou vários utilizadores da aplicação terão oportunidade de classificar uma ou mais *slices* de várias VCs. Estes momentos temporais iniciam-se com t=0 e crescem da seguinte forma:

- **Regra das 100 classificações-** O momento temporal vai avançar, se todos os utilizadores juntos já tiverem realizado um total de 100 classificações de *slices*, após essas 100 classificações, avança novo momento temporal (de t=i para t=i+1).

A cada momento temporal todos os utilizadores do *BugSpotting* terão a sua oportunidade de classificar uma ou mais *slices*, em tempos diferentes e ao seu ritmo de classificação, por exemplo, no tempo t=4 podem já existir várias classificações daquela *slice* quando o utilizador $x$ der a sua classificação, dadas pelo utilizador $x1$ e $x2$. Já existindo assim uma classificação tendencial e uma reputação dinâmica para os utilizadores $y$ e $z$.

Podemos observar na 3.15, que durante o t=1 o utilizador 1, obteve as *slices* 1 e 4, classificando-as com o grau de confiança que achou apropriado naquele momento. Posteriormente, foi calculada a sua reputação dinâmica com base na tendência, ou seja, *slices* que ele já classificou anteriormente, 2 e 3.

Durante esta janela temporal t=1, outros utilizadores podem influenciar esta classificação, pois continuam a classificar esta e outras *slices*.

Podemos observar posteriormente a classificação do utilizador 1 nas *slices* 2 e 5 no t=2, na tabela 3.16.

A cada tempo ou momento temporal como podemos verificar na tabela seguinte 3.17, temos o 2-2-1, 2-2-3 e 2-2-4, em que o utilizador 2 classificou respetivamente as *slices* 1, 3 e 4, mas no mesmo momento outros utilizadores não esperam este acabar de classificar e irão classificando as suas *slices* também. Resumindo, já existem mais classificações de outros utilizadores para aquela *slice*, e assim a reputação dinâmica de ambos os utilizadores irá ser afetada e alterada de acordo com a sua reputação e tendência naquele momento.

A classificação do utilizador e o seu grau de confiança atribuído à *slice* daquela VC será sempre estático ao longo de todos os momentos temporais, não sendo alterada, mas naquele mesmo momento temporal, outros utilizadores irão seguir o mesmo caminho, classificando aquela e outras *slices* alterando assim a tendência e suas reputações dinâmicas.

Com o passar de várias classificações e momentos temporais vamos obtendo muita informação sobre os vários utilizadores e suas classificações. Vejamos o seguinte exemplo, o utilizador 1 classificou uma *slice* como não vulnerável, outros 9 utilizadores classificaram a mesma como vulnerável, resultando assim para o utilizador 1 uma péssima reputação.

Com o passar de vários momentos temporais, este utilizador 1 mostrou-se de acordo com a tendência em todas as *slices* que este classificou, revelando-se então um bom classificador de *slices*.

Podemos concluir que, afinal o utilizador 1 é na verdade um ótimo classificador, pois a sua reputação dinâmica assim o revelou, e a maioria irá baixar a sua reputação. De modo a conseguirmos perceber melhor este panorama, podemos ver na Figura 3.5, que quando é iniciado um novo momento temporal, é necessária a informação de todas as classificações dadas pelo utilizador e grau de confiança atribuído, pois só deste modo é possível calcular a tendência e *label* final da *slice* naquele momento.

Posteriormente, usando os valores estáticos, $label$ e $confidenceDegree$ bem como a tendência da *slice*, é possível calcular a sua reputação dinâmica, ou seja, é utilizada a informação do passado de cada utilizador e com a tendência, que juntas permitem o calculo da reputação dinâmica.

Explicando de outra forma, ao iniciar a primeira ronda, no t=1, é calculada a tendência com base nas primeiras classificações daquela *slice*, pelo utilizador.

Posteriormente, com a entrada de novas classificações por parte dos utilizadores, estes irão alterar a tendência bem como a reputação dinâmica de todos os utilizadores que já classificaram aquelas *slices* no passado, até que aquele momento temporal termine.

Se um utilizador ainda não classificou nenhuma daquelas *slices*, é usada a reputação inicial, senão é utilizada a reputação dinâmica.

Resumindo, nas Tabelas 3.16 e 3.15 podemos verificar que a tendência foi definida por VC, pois, o utilizador é muito bom em SQLi (VC=1) e não em XSS (VC=2) e deste modo acaba por reduzir a sua reputação em XSS, ou não contribui nas classificações, como pode acontecer o contrário, sabe muito da VC1 SQLi e só classifica essa VC e fica por exemplo com uma

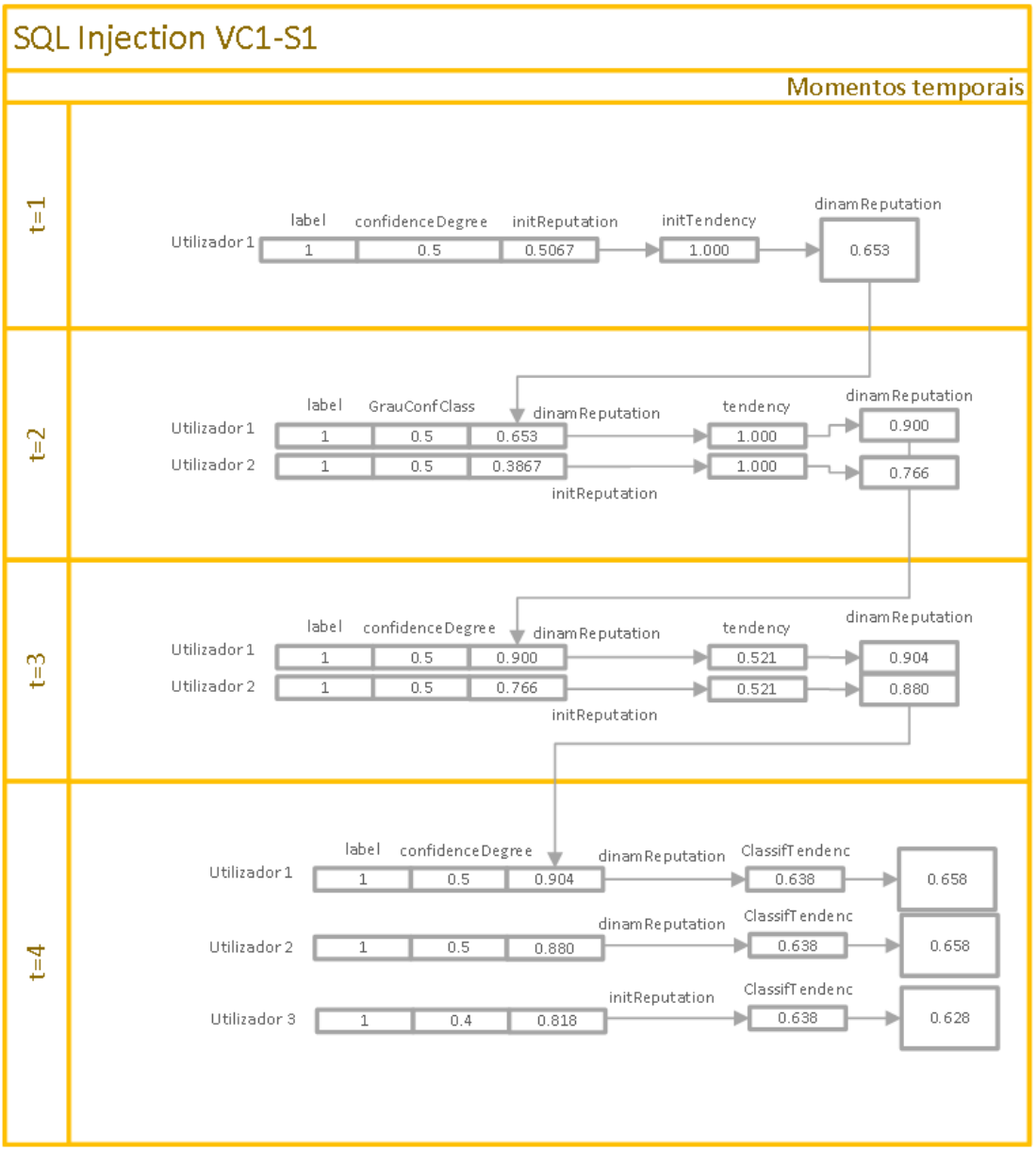



Figura 3.5: Diagrama obter reputação dinâmica

classificação ótima. Depois ao classificar outra VC, e caso acerte a *slice* ficará com boa reputação nessa VC.

Tabela 3.15: Classificação tendencial e evolução da reputação dinâmica do utilizador 1 nas VC1-S1 e S4 no t=3

| Utilizador | Slice | initReputation | dinamReputation | Classificação tendencial | Label atual |
|---|---|---|---|---|---|
| 1 | 1 | 0,387 | 0,653 | 1 | |
| 1 | 4 | 0,498 | 0,550 | 0 | |

Tabela 3.16: Classificação tendencial e evolução da reputação dinâmica do utilizador 1 nas VC1-S1 e S4 no t=4

| Utilizador | Slice | initReputation | dinamReputation | Classificação tendencial | Label atual |
|---|---|---|---|---|---|
| 1 | 1 | 0,850 | 0,904 | 0,639 | |
| 1 | 4 | 0,550 | 0,150 | 0,639 | |

Tabela 3.17: Classificação tendencial e evolução da reputação dinâmica do utilizador 2 no tempo 8 nas VC1-S1, S3 e S4

| Utilizador | Slice | initReputation | dinamReputation | Classificação tendencial | Label atual |
|---|---|---|---|---|---|
| 2 | 1 | 0,702 | 0,7633 | 0,903 | |
| 2 | 3 | 0,702 | 0,4162 | 0,899 | |
| 2 | 4 | 0,691 | 0,7643 | 0,8963 | |

A reputação de uma *slice* depende da classificação de vários utilizadores, e a reputação do utilizador depende das classificações que ele fez nas várias *slices* da VC, e das reputações de vários utilizadores associados a essas *slices* para aquela VC.

A cada momento temporal a classificação de cada utilizador e o seu grau de confiança mantêm-se estático desde início, mas a cada momento temporal, outros utilizadores foram classificando e alterando aquela tendência, modificando também a sua reputação dinâmica, como podemos verificar na Figura 3.5.

A reputação de uma *slice* depende da classificação de vários utilizadores e a reputação do utilizador depende das classificações que ele fez nas várias *slices* daquela VC, e das reputações associadas a essas *slices* para aquela VC.

#### 3.3.4.4 Reputação dinâmica

Este Algoritmo 5 e elaborado de acordo com a Figura 3.5 foi pensado e criado com o objetivo de registar a reputação de todos os utilizadores a cada momento temporal (t) diferente, isto na fase pós-reputação. Uma vez que é dinâmica, a cada momento temporal é diferente, assim é calculada a reputação para cada utilizador de acordo com as classificações dadas às *slices* para aquele momento. Todos os utilizadores têm uma reputação dinâmica, e esta é influenciada pela classificação tendencial, que é obtida com base em todas as classificações dadas pelos outros utilizadores.

Não podemos esquecer que, uma vez que ainda não se sabe inicialmente a etiqueta da *slice*, utilizamos o valor da tendência no cálculo para cada utilizador, obtendo assim um valor tendencial.

Uma vez atribuída a reputação inicial para cada utilizador, verificámos que o valor de reputação inicial atribuído não podia ser fixo pois, a cada momento temporal o conhecimento e classificação atribuída a diferentes *slices* por diferentes utilizadores é também diferente. Nesse sentido, foi decidido criar uma reputação dinâmica, que permite distinguir os diferentes utilizadores à medida que estes classificam as diversas *slices*.

Como explicado anteriormente, um utilizador adquire uma reputação inicial, na fase de reputação, e é esse o valor que entra no cálculo da primeira *slice* que classifica. Posteriormente, a

**Algorithm 5** Obter reputação dinâmica

**Input:** $confidenceDegree$, $label$, $vulnerabilityClassId$
**Output:** Reputação dinâmica
**if** $dinamReputation = 0$ **then**
    O utilizador não tem reputação dinâmica atribuída
    **exit**
**else**
    **Output:** Retorna a reputação dinâmica do utilizador
    O utilizador pede uma *slice* para uma VC
    A ferramenta de SAST classifica a *slice*
    **if** $initReputation \neq 0$ **then**
        Se existir valor de tendência anterior
        **if** $tendencia$ existe **then**

$$\begin{aligned} dinamReputation_y = {} & 0.4 \cdot (NOT \oplus label_y) \\ & + 0.2 \cdot confidenceDegree_y \\ & + 0.4 \cdot lastTendency_y \end{aligned}$$

            **return** $dinamReputation_y$
        **else**
            Se não existir valor de tendência anterior / 1ª classificação do utilizador

$$\begin{aligned} dinamReputation_y = {} & 0.3 \cdot (NOT \oplus label_y) \\ & + 0.3 \cdot confidenceDegree_y \\ & + 0.4 \cdot initReputation_y \end{aligned}$$

            **return** $dinamReputation_y$
        **end if**
    **end if**
**end if**

momento temporal seguinte, o valor que irá ser utilizado para a reputação desse mesmo utilizador, já não será o inicial, mas sim o valor de reputação dinâmica, calculada com base na tendência da *slice*, que tem em conta as reputações, graus de confiança e classificações de todos os utilizadores daquele momento temporal. Importa referir que a reputação dada noutras *slices* por um ou mais utilizadores influencia e altera a sua reputação dinâmica, ou seja, se o utilizador 1 acertar a primeira *slice* e falhar as outras três seguintes, a sua reputação dinâmica baixa, e tem menos ”poder”na classificação da *slice* 1, pois falhou mais do que acertou. O mesmo acontece para os outros utilizadores.

#### 3.3.4.5 Grau de indecisão

Uma vez obtida a reputação dinâmica de todos os utilizadores no último momento temporal, que é o atual para aquela *slice* $y$ daquela VC $z$, é utilizada essa reputação junta com a tendência da *slice*. Posteriormente são selecionadas as *slices* com maior grau de indecisão, ou seja, aquelas que estão situadas mais próximas do valor central (0.5), pois é neste intervalo que se situa maior indecisão e

divergência nas classificações.

Para que fosse possível, foi necessário balizar e separar quais as *slices* que possuem um maior grau de indecisão, não apenas as de valor 0.5, mas também as definidas no intervalo entre 0.3 e 0.7, pois são essas que tem uma maior urgência na descoberta da etiqueta final.

Para que seja fácil compreender o funcionamento o cálculo da tendência descrito anteriormente, foi elaborado o Algoritmo 6.

**Algorithm 6** Resolver grau de indecisão

```
1: Input: userId, VulnClassId, sliceId, tendency
2: Output: Retorna a slice com maior grau de indecisão
3: O Utilizador pede uma slice de uma VC
4:
5: if initReputation != 0 then
6:     if tendency != 0 then
7:         if nº classificadores > 5 then
8:             if dinamReputation is true then
9:                 É feita a contagem de utilizadores no range de indecisão e no total
10:                indecisao = n/N + 0.1/N
11:                return slice com maior grau de indecisão
12:            end if
13:        end if
14:    end if
15: end if
```

Posteriormente à obtenção de reputação dinâmica, utilizando a tendência da classificação da *slice*, entra um novo cálculo, para o grau de indecisão da *slice*. Pretende-se com a utilização deste algoritmo balizar quais as *slices* com maior grau de indecisão, para que possam ser essas as próximas *slices* a entregar aos utilizadores, pois são elas que devem ser priorizadas.

Para o cálculo de obtenção do valor do grau indecisão é utilizada a reputação dinâmica de cada utilizador, bem como a tendência da *slice* para aquele momento temporal. Posteriormente são selecionadas as classificações que se situam entre os valores 0.3 e 0.7, pois são valores perto da medida central (0.5) que separa a *slice* vulnerável da não vulnerável.

Neste momento é necessário para elaborar o cálculo da tendência, tendo em consideração as classificações dadas pelos utilizadores e sua reputação no momento, às *slices*, de modo a que, fosse possível a verificação da existência de divergência, ou falta de consenso e confiança por parte dos utilizadores.

Assim, foi criado um cálculo que é alterado de acordo com a classificação dada pelos utilizadores àquela *slice*, em diferentes momentos temporais.

Estes cálculos foram todos feitos para a *x-1-1*, depois varia para *x-y-z*. Foram definidos dois limites, ou métricas a verde, que identificam a zona de divergência ao longo dos momentos temporais, bem como, uma linha, a vermelho, que simboliza a separação entre a zona vulnerável e não vulnerável de acordo com a tendência e classificação da *slice* com base na reputação dinâmica do utilizador.

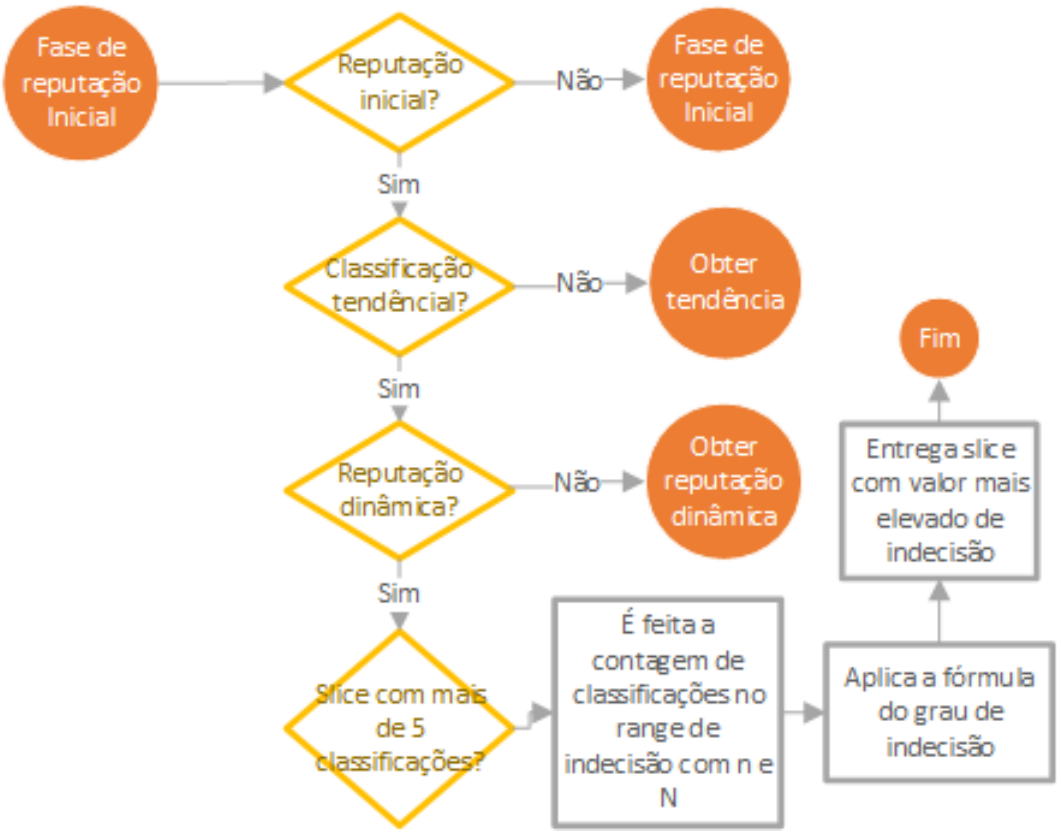



Figura 3.6: Diagrama obter grau de indecisão

Esta zona de divergência, entre linhas verdes, representam o grau de indecisão, e *slices* com divergência a selecionar. Neste sentido, torna-se necessário para o cálculo, verificar o número de pontos situados nesta zona, e avaliar o grau de divergência.

Podemos definir ***n*** como o número de utilizadores no *range* de divergência, e ***N*** como o número de classificações dos utilizadores naquele momento.

$$GrauIndecisao_x = \frac{n}{N} + \frac{0.1}{N} \tag{3.8}$$

Após os momentos temporais daquela ronda é calculado o grau de indecisão para cada uma das *slices* usando a fórmula 3.8, que nos permite no eixo do $x$, n, obter o utilizador que classificou aquela *slice* e no eixo $y$, N, a tendência que ele obteve com base na reputação naquele momento.

O objetivo deste cálculo passa por chegar a um valor mais próximo de 0, significando que a *slice* deixou de ter relevância e deixará de ser entregue ao utilizador.

Podemos observar aqui um exemplo, quando um utilizador pede uma *slice* para classificar, num cenário em que temos 1000 *slices*, cada uma delas correspondente a um gráfico definido anteriormente com um grau de indecisão calculado. No caso deste trabalho, foram efetuados todos os cálculos para quatro *slices* da mesma VC, a 1, e uma para VC 2.

Complementando os gráficos anteriores referentes ao grau de indecisão, foi possível constatar e chegar a uma estatística válida sobre qual a *slice* a entregar ao utilizador, como podemos verificar

na Tabela 3.18:

Tabela 3.18: Grau de indecisão referente às 5 slices

| VC1-S1 | VC1-S2 | VC1-S3 | VC1-S4 | VC2-S1 |
|---|---|---|---|---|
| 0,468 | 0,703 | 0,476 | 0,276 | 0,193 |

Neste momento temos três soluções para resolução ao exemplo anterior:

- **Maior grau de indecisão**: Entrega a *slice* que tem um maior grau de indecisão das 1000.
- **Sub-conjunto mais elevado**: Entrega ao utilizador a *slice* que se repete com maior frequência, a moda, pois são aquelas com maior grau de indecisão. Posteriormente poderá ser aplicado um novo *threshold* com diferentes métricas e critérios. Como 1º critério poderiamos aplicar 60%, mas, se todos os graus de indecisão forem inferiores e não existir nenhum, irá ser aplicado o 2º critério, aplicando menor *threshold* ou aplicar um *random*.
- **Percentagem da indecisão**: Calcular a percentagem referente ao número de *slices* com maior grau de indecisão, por exemplo 10% e decidir qual delas entregar, seja por ordenação ou *random*.

# Capítulo 4

# Desenvolvimento BugSpotting 2.0

Neste capitulo irá ser abordado o modelo estrutural do BugSpotting 2.0, bem como as tecnologias utilizadas no seu desenvolvimento.

## 4.1 Tecnologias

Nesta secção será apresentada uma visão geral sobre as tecnologias usadas no BugSpotting 2.0, separada pelos dois principais componentes maioritariamente implementados no desenvolvimento de aplicações *web, frontend e backend*.

### 4.1.1 Componentes utilizados

As duas componentes utilizadas na construção do BugSpotting, o *frontend* e *backend*, permitem alcançar a finalidade proposta para esta aplicação *web*.

Estes dois componentes são explicados de seguida e encontram-se representados na Figura 4.1.

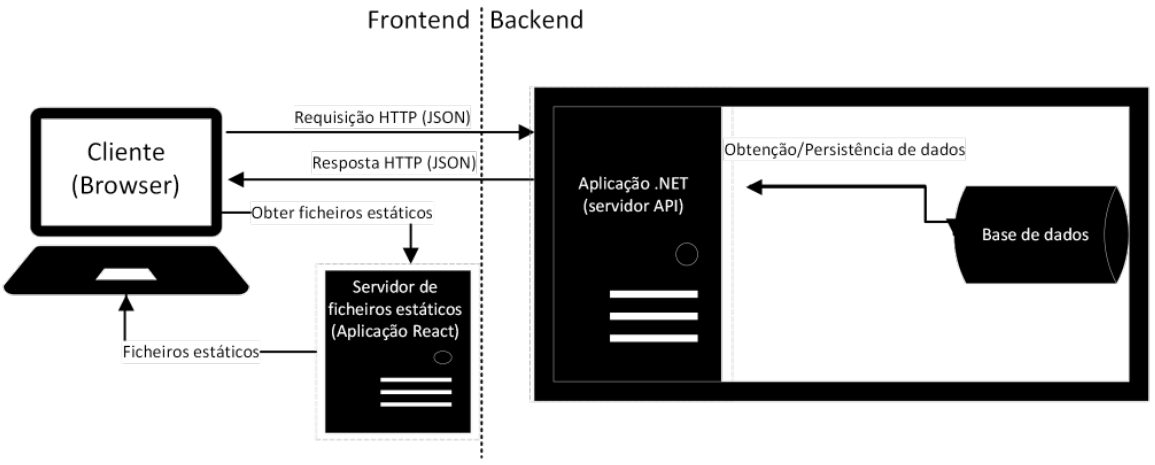



Figura 4.1: Componentes utilizadas no BugSpotting 2.0

#### 4.1.1.1 *Frontend*

*Frontend* numa aplicação refere-se ao lado do cliente, toda a parte visível, desde botões, menus e imagens, através do qual os utilizadores interagem diretamente, de forma geral através do *browser*.

Para contextualizar, e para que seja possível perceber de forma simples como está construída esta aplicação, é feita de seguida uma explicação quanto às linguagens de programação empregues nesta componente.

- **O HTML-** O *HyperText Markup Language* (HTML) é a base da construção de páginas *web*. É uma linguagem de marcação utilizada para estruturar o conteúdo e *layout*que é apresentado nessa página, organizando elementos como textos, imagens, vídeos ou *links*, definindo o que aparece na página e em que ordem. Ou seja, HTML descreve a estrutura e conteúdo de uma página *web*, sendo visto como o seu ”esqueleto”.

  [HTML, CSS, and JavaScript All in One Covering HTML5, CSS3, and ES6”de Julie C. Meloni e Jennifer Kyrnin]

- **O CSS-** O *Cascading Style Sheets* (CSS) é uma linguagem *stylesheet* usada para introduzir estilos nas páginas *web*. Controla o *design* e apresentação do conteúdo criado com HTML, incluindo cores, tamanhos ou espaçamentos. Ou seja, define a aparência e experiência visual da página *web*, sendo vista como a ”pele”por cima do ”esqueleto”.

- **O Javascript - React-** O *JavaScript* (JS) é uma linguagem de programação interpretada, pois não precisa de ser compilada antes de ser executada, que adiciona interatividade e funcionalidade às páginas *web*. É utilizado para criar comportamentos dinâmicos, tal como responder a cliques de botões presentes na página, bem como validação de formulários. Ou seja, adiciona lógica e interatividade ao *website*, sendo visto como o seu ”cérebro”. Esta analogia de ”esqueleto”, ”pele”e ”cérebro”está representada na Figura 4.2.

  O React é uma biblioteca de código aberto do JS [14] criada pelo Facebook que se baseia em componentes ou blocos de construção independentes e reutilizáveis. É usada principalmente no desenvolvimento de *interfaces* de utilizador (UI), especialmente nas aplicações *web* de página única (SPA). Tem como características principais a reutilização de componentes, atualização do *Document Object Model* (DOM) virtual em vez de atualizar o DOM real diretamente, melhorando o desempenho, uma vez que quando o estado do componente é alterado, o React atualiza apenas os elementos necessários no DOM real. DOM é visto como o ”mapa”da página *web* que contém uma estrutura com todos os elementos HTML.

  Outra das suas características consiste no uso de *JavaScript Syntax Extension* (JSX), uma extensão que combina *JavaScript* com uma sintaxe semelhante ao HTML, tornando o código da *interface* mais intuitivo e legível. Posteriormente o JSX é convertido em JS e de seguida em HTML para que o *browser* possa interpretar.

  No contexto da nossa solução, foi utilizada esta biblioteca pela sua eficiência e flexibilidade,

tornando a criação de interfaces um processo mais simples e de desenvolvimento rápido e organizado.

- **Figma-** O Figma é uma ferramenta utilizada na criação de *interfaces* aplicada por *designers User Interface e User Experience* pela sua simplicidade e rapidez. O Figma foi utilizado no desenho das páginas do BugSpotting, o que facilitou a sua implementação e organização.

Estas três componentes HTML, CSS e JS são parte fundamentais na construção e desenvolvimento de páginas *web*, e são utilizados na nossa implementação.

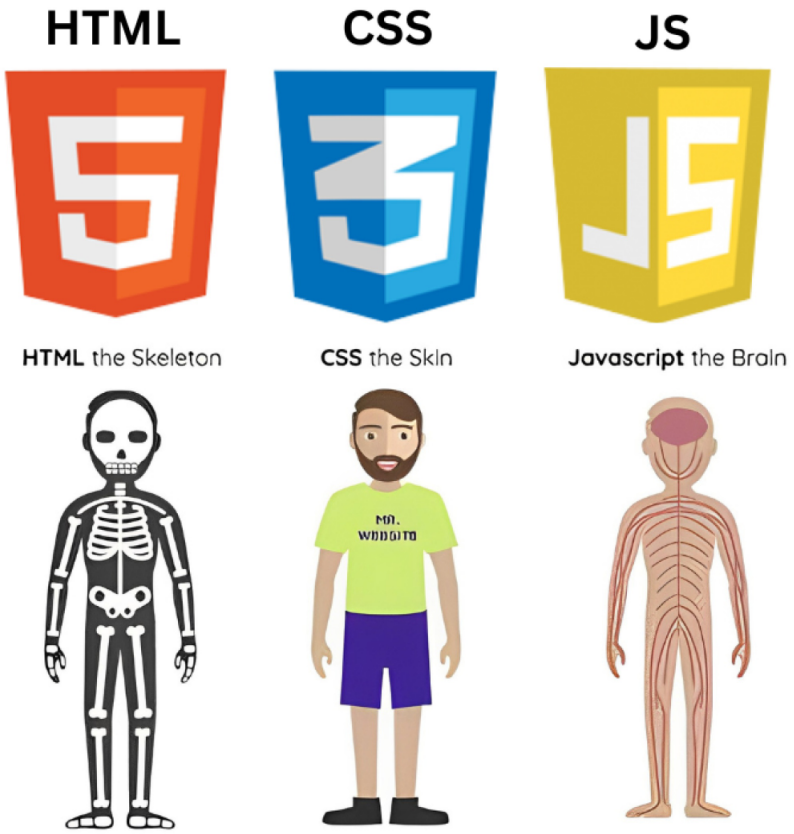



Figura 4.2: HTML, CSS e JavaScript

Nesta aplicação, a componente *frontend* engloba duas entidades que podem atuar no sistema, o administrador e o utilizador da aplicação. O administrador tem privilégios de gestão de *slices* e classificadores (SASTs e utilizadores) da base de dados (BD). O utilizador tem como tarefas autenticar-se no *website* e classificar as *slices* de código PHP introduzidas pelo administrador.

Na Figura 4.1, está representado o esquema principal de comunicação HTTP (*Hypertext Transfer Protocol*), efetuada entre o utilizador (usando o seu *browser* e acedendo ao *website*) e o servidor. O protocolo HTTP é usado para comunicação e transferência de dados, em que os clientes enviam pedidos ou requisições HTTP aos servidores e estes devolvem uma resposta [6]. As trocas de mensagens são feitas em formato JSON (*JavaScript Object Notation*), para acesso

e troca de dados com a BD. O utilizador, ao receber as *slices* dadas pelo *backend*, classifica as mesmas.

#### 4.1.1.2 *Backend*

Nesta secção é apresentado o processo realizado do lado do servidor, *backend*, ou seja, o que acontece ”atrás dos panos”, seja a gestão de *slices*, tratamento das classificações e construção das etiquetas finais. Esta componente tem como objetivo tratar e devolver respostas a todos os pedidos provenientes do *frontend*, e se necessário, pedindo a informação à BD.

O *backend* é responsável por processar todas as informações provenientes dos clientes, pois é o ”cérebro”da aplicação, fazendo com que o *frontend* funcione corretamente. Todos os algoritmos usados no BugSpotting estão presentes nesta componente.

É apresentada de seguida uma explicação quanto às tecnologias empregues nesta componente, *backend*.

- **.NET Framework.** Para que seja possível compreender o que é o .NET *framework* é importante perceber primeiro o que é uma *framework*. Em desenvolvimento de *software* é definida uma *framework* como uma estrutura que facilita a implementação de sistemas, fornecendo um modelo base que permite adicionar funcionalidades. Uma das suas características é o uso da inversão de controlo ou *Inversion of Control* (IOC), onde a *framework* controla o código do programador, facilitando a reutilização e modularidade, pois os componentes do sistema têm maior independência, facilitando a fase de testes apenas a algum módulo. O .NET *framework* é uma plataforma de desenvolvimento da Microsoft, e é composta por duas partes centrais, o *Common Language Runtime* (CLR) e a *Class Library*. O CLR é o ”motor”de execução que gere as tarefas como gestão de *threads*, *garbage collection*, exceções e compilação para código que o computador perceba. A *Class Library* oferece API's e classes para tarefas necessárias, como leitura/escrita de ficheiros e ligações a BD por exemplo. O .NET *framework* suporta várias linguagens de programação, tais como C#, VB.NET e F#, tornando-a uma ferramenta poderosa.

- **MySQL.** O sistema Sistema de Gestão de Base Dados Relacional (SGBDR), *open-source*, usado nesta aplicação é o MySQL, que organiza todos os dados em tabelas relacionadas. O MySQL permite aos utilizadores armazenar, organizar e recuperar dados de forma eficiente. É reconhecido pela sua potência e flexibilidade, podendo assim gerir grandes volumes de dados, como também ser executado em servidores locais ou na *cloud*. Por ser um projeto de código aberto beneficia de contribuições por parte da comunidade.

### 4.1.2 Base Dados

De seguida segue-se uma explicação acerca de todas as entidades que compõem o modelo de Entidade Relacional (ER) no BugSpotting 1.0, e posteriormente, as respetivas modificações necessárias e atualizadas para o correto funcionamento na nova versão, BugSpotting 2.0. O modelo

relacional está representado na Figura 4.3.

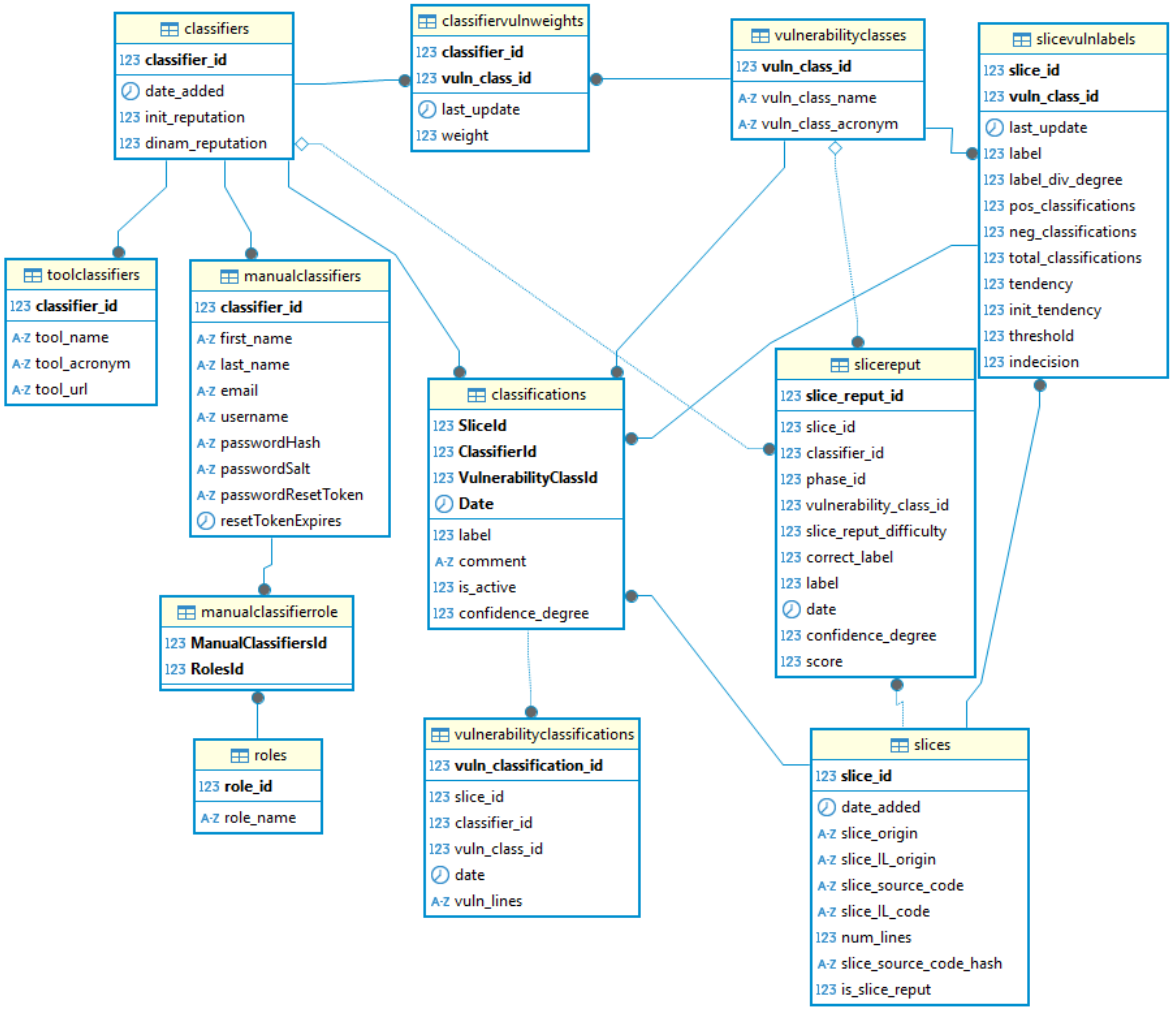



Figura 4.3: Modelo ER

- ***Classifier***

  Começando pela entidade *Classifier*, esta foi criada com o objetivo de guardar informação do utilizador ou SAST que está a classificar uma *slice* no momento, bem como a data em que foram adicionados ao sistema.

  Esta entidade depende e engloba duas outras, a entidade *Tool Classifier e Manual Classifier*.

  A entidade *Tool Classifier* corresponde à ferramenta SAST e a entidade *Manual Classifier* aos utilizadores humanos, e ambas guardam informação referente à sua identificação.

  É feita uma relação de um-para-um entre estas duas entidades pois são dependentes da entidade *Classifier*, que não existe sem as entidades *Tool Classifier e Manual Classifier*.

  Só após introdução de dados nas entidades *Classifier Manual* e *Tool Classifier* é são populados dados na entidade *Classifier*.

- ***VulnerabilityClass***

  A entidade *VulnerabilityClass* integra todas as classes de vulnerabilidades existentes na aplicação, neste momento oito VC's. É composta pelo id, nome da vulnerabilidade respetiva, $vuln_class_name$ e pelo seu acrónimo $vuln_class_acronym$.

- ***ClassifierVulnWeight***

  Um classificador, seja ele humano ou ferramenta SAST, pode classificar mais que uma classe de vulnerabilidade, e por outro lado uma *slice* pode conter mais que um tipo de vulnerabilidades. Por fim, uma VC pode ter vários tipos de classificadores.

  Esta entidade possui uma relação de muitos-para-muitos entre a entidade *Classifier* e *VulnerabilityClass* (filhos).

  A entidade *ClassifierVulnWeight* possui ainda um método chamado *weight*, que representa o peso que é atribuído ao utilizador e ferramenta SAST com base nas suas classificações corretas ou erradas.

  Como foi possível verificar anteriormente, antes da descoberta da etiqueta final da *slice*, é calculada a tendência e posteriormente a reputação dinâmica.

  Inicialmente, no BugSpotting 1.0, foi definido um valor de 50 pontos (na nova versão é vista em percentagem %) de reputação para os utilizadores no momento em que classificam a *slice* $x$ daquela VC $z$, que posteriormente vai sendo atualizada de acordo com as novas classificações dadas. Assim que é atingido o consenso, ou seja, a classificação dada pelos classificadores tender para o mesmo valor, cada classificador soma cinco pontos, caso contrário são-lhes subtraídos cinco pontos.

- ***Slice***

  Uma *slice* é um pedaço de código PHP que é classificado a cada momento temporal pelos classificadores, como vulnerável ou não vulnerável.

  Esta entidade está relacionada com a entidade *Classification* de um-para-muitos, pois uma *slice* pode ter várias classificações dadas pelos classificadores, e por outro lado uma *slice* tem uma classificação dada por um classificador.

  O atributo $num_lines$ representa o número de linhas que a *slice* contém, que futuramente irá ser utilizado como grau de dificuldade da mesma.

- ***SliceVulnLabel***

  O objetivo desta entidade consiste no facto desta ser atualizada consoante as classificações dadas pelos utilizadores e ferramentas SAST, pois é ela que guarda o valor da etiqueta final no momento.

  Esta entidade resulta da relação de muitos-para-muitos entre as entidades *VulnerabilityClass* e *Slice*. Quer isto dizer que, uma *slice* pode estar associadas a várias VC's bem como por outro lado, uma VC pode conter várias *slices*.

Esta, possui vários campos que são atualizados ao longo do tempo com base nas classificações dadas às *slices* com dados introduzidos na entidade *Classifications*.

É nesta entidade que se encontra o atributo $label$, que é atualizado de acordo com a classificação dos classificadores a determinada *slice* $y$ de uma VC $z$ tem. Esta classificação pode ter dois valores, 0 ou 1, não vulnerável ou vulnerável respetivamente.

Existe um outro atributo chamado $label_div_degree$ que contêm o valor relativo ao grau de divergência entre os vários classificadores perante uma determinada *slice* $x$.

Existem por fim três outros atributos que fazem uma contagem quanto às classificações positivas, negativas bem como o seu total, $pos_classifications$, $neg_classifications$ e $total_classifications$, respetivamente.

Cada um destes campos é atualizado à medida que as classificações são dadas pelos classificadores, em que, caso a classificação dada à *slice* seja vulnerável, o atributo $pos_classifications$ é incrementado.

O mesmo acontece para as classificações negativas, $neg_classifications$, caso o valor da classificação dada à *slice* seja não vulnerável, este tem o seu valor incrementado.

O atributo $total_classifications$ corresponde ao número total de classificações, ou seja, a soma dos dois atributos anteriores. Está também relacionada com o limite ou *threshold* que é uma variável do sistema que indica qual o momento em que se devem parar as classificações de determinadas *slices* $y$ de uma VC $z$, assim que este valor é atingido.

- ***Classification***

Esta entidade tem o objetivo de guardar todas as classificações dadas pelos utilizadores e ferramentas SAST, e para isso guarda a sua identificação, *slice* classificada, VC e a data em que foi feita a classificação.

Na altura foi adicionada uma data como chave primária desta entidade, com o objetivo de futuramente, caso necessário classificar a mesma *slice*, pelo mesmo classificador, na mesma VC, esta *slice* poderá aparecer novamente para ele.

Todos estes registos ficam guardados na entidade *VulnerabilityClassification*.

Esta entidade *classification*, também possui um atributo chamado $label$ que corresponde à etiqueta obtida na classificação dada pelo utilizador e ferramenta SAST, sendo que ao contrário da entidade *SliceVulLabel*, esta etiqueta não é atualizada.

Outro atributo que existe nesta entidade é $comment$, que permite aos classificadores inserir um comentário no pedaço de código que estão a classificar.

O atributo $is_active$ indica ao sistema se a *slice* a classificar já atingiu o *threshold* ou limite, e se ainda poderá ser classificada ou não.

Por fim, temos o atributo $confidence_degree$ que guarda o valor relativo ao grau de confiança com que os classificadores classificam a *slice*.

- ***VulnerabilityClassification***

  Esta entidade guarda todos os registos relativos às classificações dadas pelos utilizadores e ferramentas SAST, como, registos relativos da *slice*, classificador, VC, data da classificação bem como das linhas assinaladas pelos utilizadores onde acham que existe vulnerabilidade.

  É uma relação de um-para-muitos com a entidade *Classification*, isto porque, como referido anteriormente, um dia poderá ser necessário guardar registo de mais que uma classificação para a mesma *slice* $y$ pelo mesmo utilizador $x$ da mesma vc $z$.

  A entidade *Classification* apenas é associada a uma entidade *Classification*, pois cada classificação é única em que a condição é definida pelo atributo $date$.

  Posteriormente foi feita uma análise pormenorizada quanto ás novas entidades e atributos a aplicar na nova versão BugSpotting 2.0, no sentido em que não fosse necessária uma grande mudança ao nível do modelo ER anterior.

  Nesse sentido, foi adicionada uma nova entidade *SliceReputPhase* com o objetivo de permitir guardar registos quanto às *slices* de reputação dadas pelos utilizadores nas três fases existentes, na fase de reputação. Estas permitem uma avaliação inicial da reputação de cada utilizador, através de *slices* já previamente etiquetadas.

  Assim, é possível identificar os seguintes atributos dentro da entidade $SliceReputPhase$, além de alguns já existentes na entidade $Slice$, $slice_origin$, $source_code$, $IL_origin$, $IL_code$ e por fim $_source_code_hash$:

- **slice_reput_Id**

  Este é o atributo que representa o identificador de cada *slice* de reputação, é único e é a chave primária desta entidade $SliceReput$.

- **slice_Id**

  Este atributo foi criado com o objetivo de guardar a *slice* de reputação que já foi classificada pelos classificadores, tendo em atenção que esta *slice* pertence à entidade *Slice*.

- **phase_Id**

  Este atributo representa o número da fase atual (das três possíveis) de classificação de *slice*, em que cada utilizador ou ferramenta SAST se encontra a classificar, de modo a que não fiquem sempre em *loop* dentro da mesma fase.

- **correct_label**

  Este atributo representa o valor associado à etiqueta final daquela *slice*, sendo o valor final 0 ou 1, não vulnerável ou vulnerável, respetivamente.

- **initTendency**

Esta variável permite guardar o valor correspondente à tendência inicial da *slice*, que permitirá obter a reputação dinâmica dos utilizadores, bem como o valor da tendência a partir daquele momento temporal inicial.

- **tendency**

  Esta variável guarda o valor da tendência dinâmica, pois a cada momento temporal este valor é influenciado pelas classificações dos utilizadores. É usado este valor para calcular as reputações dinâmicas dos utilizadores.

- **comment**

  Temos este atributo *comment* que representa uma breve descrição quanto à *slice* de reputação que foi classificada.

- **slice_reput_difficulty**

  Como referido anteriormente na explicação da fase de reputação, temos três *slices* que vão sendo classificadas, e cada uma delas com um diferente nível de dificuldade, que será guardado neste atributo.

- **date_added**

  Este atributo irá ser utilizado nesta entidade bem como na entidade *Slices*, para efeitos de cálculo do grau de dificuldade da *slice* na fase de reputação, registado na variável descrita anteriormente, $sliceReputDifficulty$.

- **score**

  Este atributo representa a pontuação ou reputação adquirida ou não naquela fase, caso acerte ou não na etiqueta da *slice*. São três diferentes fases, e no fim é feita a média destas para cada utilizador, de forma a obter o resultado para a variável $initReputation$.

  Na entidade $SliceReput$ o atributo $confidence_degree$ é utilizado para efeitos de cálculo de *slices*.

  A reputação de cada classificador no momento vai sendo guardada entre cada fase, no campo *score* da entidade $SliceReput$ e posteriormente é feita a média das três *slices* de reputação e guardadas na entidade $Classifiers$, no campo $init_reputation$.

- **init_reputation**

  Este atributo permite guardar registo a reputação inicial obtida por cada um dos classificadores, dada na fase inicial de reputação. É a variável usada no início das classificações pós-fase de reputação.

- **dinam_reputation**

  Este atributo guarda registo da reputação dinâmica para o classificador a cada classificação efetuada às *slices*.

- **threshold**

  Este atributo define qual o limite em que é possível parar as classificações de determinada *slice*, pois já se chegou a um consenso e já se sabe a tendência para qual tende etiqueta final.

- **indecision**

  Este atributo é usado para os cálculos da indecisão da *slice* através do cálculo da tendência e reputação dinâmica de cada utilizador. É definida por dois valores $n$ e $N$.

- **classifier_id**

  Este atributo permite saber quem classificou aquela *slice* naquele momento temporal. Todos os cálculos anteriores de tendência, indecisão e reputação dinâmica são aplicados a este utilizador.

## 4.2 BugSpotting 2.0 Interface

Nesta secção são apresentados duas imagens que mostram como ficou a *interface* do BugSpotting 2.0. Uma das imagens é referente ao aspeto do BugSpotting 2.0 no momento da escolha de fase na reputação inicial, como representado na Figura 4.4. Assim que o utilizador faz registo na aplicação e começa a classificar *slices*, apenas lhe é permitida a classificação de *slices* de reputação. Só após esta fase poderá escolher a VC e classificar *slices* construtivas para o *dataset*. Na outra imagem podemos observar o momento em que é dada uma *slice* de reputação ao utilizador, Figura 4.5.




Figura 4.4: Interface BugSpotting 2.0

Nesta fase de reputação inicial foi utilizado o algoritmo 2 da fase de reputação representada na Secção 3.3.4.2 que permite a obtenção de uma reputação inicial para cada utilizador na aplicação, separando utilizadores mais experientes dos menos experientes.

Tal como referido anteriormente a fase de reputação é composta por três *slices* de reputação, e no fim da classificação destas *slices* por parte dos utilizadores, estes obtêm uma reputação inicial, e é esta que é usada posteriormente na reputação dinâmica.

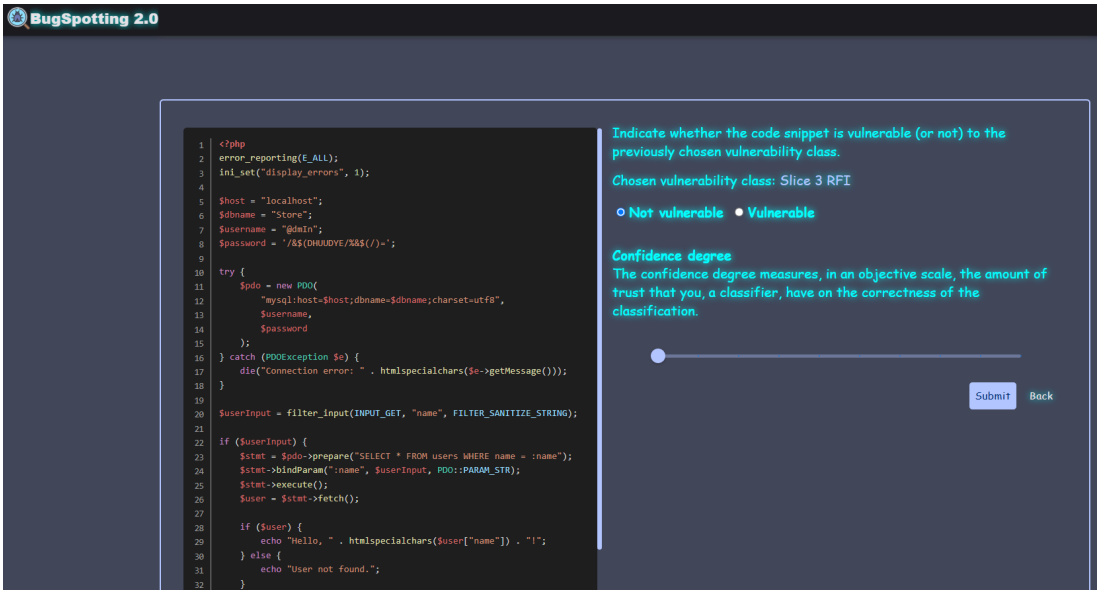



Figura 4.5: Classificação de Slice de reputação BugSpotting 2.0

Assim que um utilizador entra na fase de reputação, é verificada a fase atual de classificação, e apenas lhe é permitida a classificação das *slices* não classificadas, de entre o total das três.

Após fase de reputação é iniciada a fase de classificação de *slices* pós fase reputação, como podemos observar na Figura 4.6.




Figura 4.6: Classificação de Slices BugSpotting 2.0

Assim que um utilizador entra na fase de classificação de *slices* escolhe uma das oito VC e começa a classificar.

# Capítulo 5

# Avaliação

Nesta Secção estão representados os cálculos necessários para que os algoritmos funcionem da forma pretendida. Foram deste modo elaborados vários gráficos que nos permitiram obter conclusões sobre a veracidade dos algoritmos construídos. São apresentadas de seguida essas mesmas simulações:

## 5.1 Simulações de alguns cenários

Foram efetuadas várias simulações que nos permitiram obter uma análise ao longo do tempo da tendência e evolução da classificação dos utilizadores perante as *slices* classificadas.

Posteriormente, foram separados por fases todos os cálculos, identificando-se de seguida as quatro fases mais importantes que gostaríamos de simular:

- Três *slices* iniciais, para criação de reputação na fase inicial de reputação;
- Obtenção de reputação inicial, através da média das três *slices* de reputação iniciais;
- Criação de reputação dinâmica para os utilizadores que classifiquem novas *slices*, bem como, criação de uma tendência que sirva, no futuro, para uma possível previsão da *label* final da respetiva *slice*;
- Obtenção do grau de indecisão referente às classificações das *slices*.

Assim, na fase de reputação, foram definidos inicialmente com três graus de dificuldade diferentes para classificação, vinte utilizadores e três *slices* iniciais.

Após a fase de reputação, é calculada a média das três *slices* classificadas pelo utilizador, obtendo-se, assim, a sua reputação final.

Foram, por fim, após a fase de reputação, adicionadas quatro *slices* novas para classificação, mas, desta vez, *slices* pós-reputação inicial, de modo que seja possível verificar uma evolução da *label* final, através da classificação feita pelos utilizadores. Estas slices introduzidas são $x$-1-1, $x$-2-1, $x$-3-1, $x$-4-1, $x$-1-2, ou seja, 4 *slices* da VC1 e uma *slice* da VC2.

Posteriormente, foram introduzidos mais utilizadores, em rondas ou tempos diferentes, tendo sido constatada uma alteração na tendência das classificações, assim como uma mudança de *label* final, diferente para cada momento temporal.

O utilizador número 1 é o primeiro a classificar aquela *slice* $y$ daquela classe de vulnerabilidade $z$ (VC), ou seja, $1 - y - z$. Seguindo o mesmo caminho do utilizador 2, entram no momento temporal seguinte os utilizadores 3, 4, 5 e 6, classificando a mesma *slice* $y$, obtendo, assim, uma nova tendência, que resulta numa nova *label* final, atual, referente àquele momento temporal.

### 5.1.1 Classificação de *Slices*

Nesta sub-secção estão representados os cálculos necessários para que os algoritmos funcionem da forma pretendida, assim, foram elaborados vários gráficos que nos permitiram obter conclusões sobre a veracidade quanto aos dados.

Diferentemente do que sucede com as *slices* CVE´s, as novas *slices* dependem de uma tendência para obtenção de um resultado correto e criação da sua etiqueta final. Como tal, é necessária uma previsão com base em classificações anteriores ou em histórico a partir do qual seja possível obter uma estimativa.

Foi criada uma linha temporal, iniciada em t=1 (t=tempo) em que a cada momento temporal, foi verificada a tendência e variação da classificação, até um possível consenso final.

No primeiro momento temporal, t=1, para a classe de vulnerabilidade (VC) escolhida pelo utilizador, que irá analisar e classificar as três *slices* de reputação que lhe irão ser atribuídas obtendo assim, a sua reputação inicial.

Com a entrada de novos utilizadores, em momentos temporais diferentes, esta informação será complementada, ao longo do tempo, verificando-se uma evolução até que a *slice* convirja para uma etiqueta final.

A fórmula escolhida para calcular o valor da variável $initReputation$, para cada uma das três *slices*, na fase de reputação, é apresentada de seguida, na Equação 5.1.

$$\begin{aligned} initReputation_y = {} & 0.2 \cdot confidenceDegree_y \\ & + 0.5 \cdot (\oplus(label_y \cdot correctLabel_y)) \\ & + 0.3 \cdot sliceReputDifficulty_y \end{aligned} \tag{5.1}$$

De modo a testar a fórmula, foi atribuído ao utilizador X, a verde, o nível máximo de confiança (1), máxima dificuldade da *slice* (1), em que este a classificou como "vulnerável" (1) e a *slice* de reputação era efetivamente "vulnerável" (1).

Foi feito o mesmo para o utilizador Y, a vermelho, usando a mesma *slice*, mas, desta vez, errando a classificação da *slice*, obtendo como resultado metade da classificação (0.5), como é possível observar na Tabela 5.1.

Segundo a Tabela 5.1, representada anteriormente, caso um utilizador falhe na classificação da *slice* ou esta seja "não vulnerável", esta variável $correctLabel$ ou $label$ iria influenciar o re-

Tabela 5.1: Exemplo com dois utilizadores e duas classificações

| **Utilizador** | $\mathbf{sliceReputDifficulty_y}$ | $\mathbf{label_y}$ | $\mathbf{confidenceDegree_y}$ | $\mathbf{correctLabel_y}$ | **initReputation** |
|---|---|---|---|---|---|
| X | 1 | 1 | 1 | 1 | 1 |
| Y | 1 | 1 | 1 | 0 | 0,5 |

sultado obtido pela parte integrante da Equação $0.5 \cdot (NOT \oplus (label * correctLabel))$, pois a multiplicação teria sempre como resultado zero, uma vez que aconteceria $0,5 \cdot (0 \cdot 1)$, o que, matematicamente, não é possível nem correto.

Para resolver este problema, foi criado um *script* ou função que avalia se ambas as variáveis são iguais a zero e aplica o inverso ou um $OR$ ($\|$) fazendo com que o cálculo seja equivalente a $0,5 \cdot (1 \cdot 1)$, como visto anteriormente, mas apenas unilateral. Deste modo, basta que uma das variáveis seja verdadeira, $OR(label * correctLabel)$ e, ao aplicar o $OR$, retorna verdadeiro (1), se, pelo menos, uma das duas variáveis for verdadeira. De igual modo, se ambas forem falsas ou a 0, aplica o $NOT\oplus$ e, ao inverter, também dá 1 como resultado.

De seguida, foram selecionados três utilizadores diferentes, mais uma vez fictícios, como é possível observar na Tabela 5.2.

No primeiro caso, a vermelho, temos o utilizador 1, que classificou a *slice* como "vulnerável" (1), com a máxima confiança (1) e a *slice* de reputação era mesmo "vulnerável" e obteve uma classificação de 96%. Não obteve a classificação de 100%, pois a *slice* tinha um grau de dificuldade 0,8, ou 80%.

No segundo caso, a verde, temos o utilizador 2, que classificou a mesma *slice* como "não vulnerável" com a máxima confiança. Sendo esta "vulnerável", obteve a classificação de 46% para aquela *slice*, pois errou na classificação e foi penalizado.

Por fim, a laranja, para outra *slice*, o utilizador 3 classificou como "não vulnerável", acertando na classificação, obtendo, assim, a mesma reputação que o utilizador 1 na outra *slice*, ou seja, 96%.

Estes três utilizadores serviram de exemplo para demonstrar que a classificação dada à slice é bastante importante, fazendo parte do cálculo e sua reputação, tendo os valores sido levados ao máximo, para verificar se os seus limites máximo e mínimo estariam corretos.

Tabela 5.2: Cálculo da avaliação inicial de três utilizadores com 80% de confiança também)

| **Utilizador** | $\mathbf{sliceReputDifficulty_y}$ | $\mathbf{label_y}$ | $\mathbf{confidenceDegree_y}$ | $\mathbf{correctLabel_y}$ | **initReputation** |
|---|---|---|---|---|---|
| 1 | 0,8 | 1 | 1 | 1 | 0,96 |
| 2 | 0,8 | 0 | 1 | 1 | 0,46 |
| 3 | 0,8 | 0 | 1 | 0 | 0,96 |

Na primeira *slice* da fase de reputação, começámos com os 19 utilizadores iniciais e uma SAST, todos classificando a mesma *slice* com $GrauDificSlice$ fácil (0,2).

Cada um destes utilizadores atribuiu uma classificação à *slice*, entre 0 ou 1, sendo este valor guardados na variável $ClassifSlice$. Uma vez que as *slices* CVE ($ClassifSliceReput$) já estão devidamente classificadas, não é necessária nenhuma ação adicional por parte dos utilizadores,

sendo esta já representada automaticamente com a etiqueta final na base de dados.

Cada um desses 20 utilizadores executa o mesmo procedimento explicado anteriormente para aquela *slice*, bem como para a segunda e terceira *slice* da fase de reputação, alterando para cada uma delas as suas classificações e grau de confiança.

Para melhor exemplificar o resultado final do cálculo das três *slices* de reputação, foram selecionados 2 utilizadores dos 19, 6 e 17, para as três *slices* de reputação com os respetivos graus de dificuldade e com o mesmo grau de confiança.

O utilizador 6, neste exemplo, foi o pior utilizador, pois errou as três *slices* com a máxima certeza, e pelo contrário o utilizador 17 foi o melhor pois acertou em todas, com a máxima certeza e experiência.

Podemos observar na tabela 5.3 que se segue, a reputação final do utilizador 6 e 17, a vermelho e verde, pior e melhor utilizador, respetivamente.

Existe uma nuance que faz com que o utilizador 6 no fim das três *slices* de reputação não seja o utilizador com menor reputação de todos os 19 iniciais, uma vez que o grau de confiança também entra nos cálculos e o utilizador 6 e 17 foram os com maior confiança.

Utilizadores com menor grau de confiança que classifiquem a *slice* de modo errado obtêm uma menor pontuação, em contrapartida, caso a classifiquem com maior obtém um valor um pouco mais elevado, apesar da *slice* estar classificada de forma errada.

A reputação inicial ($Reput_x$) é obtida após a classificação destas três *slices* e calculada posteriormente a média da reputação delas, e por fim, esta sim é a sua reputação final antes de avançar e classificar a primeira *slice*, sem etiqueta definida.

Tabela 5.3: Exemplo com dois utilizadores e duas classificações

| **Utilizador** | **Slice 1** | **Slice 2** | **Slice 3** | **Reput** |
|---|---|---|---|---|
| 6 | 0,340 | 0,400 | 0,460 | 0,400 |
| 17 | 0,840 | 0,900 | 0,960 | 0,900 |

Tabela 5.4: Reputação dinâmica tempo=1

| **Utilizador** | **initReputation** | **dinamReputation** | **Classificação tendencial** | **Label atual** |
|---|---|---|---|---|
| 1 | 0,507 | 0,750 | 1 | |

Tabela 5.5: Reputação dinâmica tempo=2

| **Utilizador** | **initReputation** | **dinamReputation** | **Classificação tendencial** | **Label atual** |
|---|---|---|---|---|
| 1 | 0,387 | 0,850 | 1 | |
| 1 | 0,507 | 0,750 | 1 | |

Isto acontece, pois, se o utilizador 1 classificar a *slice* como "não vulnerável" (0), irá influenciar negativamente toda a fórmula (multiplicação por zero), sendo deste modo é aplicado o $NOT_XOR \oplus$.

Assim, a *slice* fica classificada como "vulnerável" (1), apenas para efeitos de cálculo, sendo que, ao longo do tempo a tendência terá de demonstrar a veracidade da *label* final daquela *slice*.

Tabela 5.6: Reputação dinâmica tempo=3

| Utilizador | initReputation | dinamReputation | Classificação tendencial | Label atual |
|---|---|---|---|---|
| 1 | 0,387 | 0,6386 | 0,4714 | |
| 2 | 0,593 | 0,6086 | 0,4714 | |
| 3 | 0,387 | 0,2786 | 0,4714 | |
| 4 | 0,593 | 0,6386 | 0,4714 | |
| 5 | 0,683 | 0,4286 | 0,4714 | |
| 6 | 0,507 | 0,6386 | 0,4714 | |

Tabela 5.7: Reputação dinâmica tempo=8 x-1-1

| Utilizador | initReputation | dinamReputation | Classificação tendencial | Label atual |
|---|---|---|---|---|
| 1 | 0,387 | 0,7935 | 0,9037 | |
| .. | .. | .. | .. | |
| 120 | 0,387 | 0,9135 | 0,9037 | |

### 5.1.2 Tendência

De modo a que fosse possível verificar a veracidade de todos os cálculos referentes à tendência e fosse possível analisar para onde a classificação das *slices* tende, a cada momento temporal diferente foram elaborados vários gráficos. O valor da tendência está compreendido entre 0 e 1, sendo que existe uma linha no valor 0.5 que separa a *slice* vulnerável da não vulnerável.

A variável tendência permite uma previsão crescente ou descendente relativamente ao número de classificações, ao longo das diferentes rondas ou momentos temporais. Para este cálculo são usadas as variáveis reputação, grau de confiança e classificação dada em conjunto com a tendência da *slice* classificada naquele momento.

Ou seja, a tendência é uma medida utilizada para definir a etiqueta atual, para aquela *slice* e para um determinado momento temporal.

Inicialmente, quando o utilizador 1 classifica a *slice*, no momento temporal t=1, ainda não existe uma tendência anterior, pois é a primeira, e neste sentido, de modo a resolver esta situação, foi alterada a fórmula da tendência.

Para o cálculo da tendência foram utilizadas as seguintes equações, consoante seja a primeira classificação daquele utilizador para aquela *slice* (tempo=0), pois é ele que define a classificação tendencial inicial, como na Equação 5.2:

$$tendenciaInicial_y = \frac{\sum confidenceDegree_y \cdot \sum label \cdot \sum initReputation}{\sum confidenceDegree_y \cdot \sum initReputation} \tag{5.2}$$

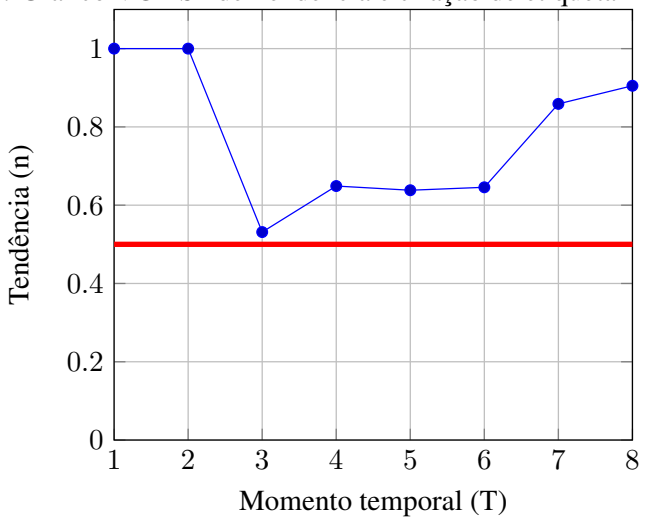
A partir desse tempo (0) todos os outros utilizadores, utilizam a seguinte fórmula baseada na classificação tendencial do tempo anterior, como na Equação 5.3:

$$tendencia_y = \frac{\sum confidenceDegree_y \cdot \sum NOT \oplus label \cdot \sum dinamReput}{\sum confidenceDegree_y \cdot \sum dinamReput} \tag{5.3}$$

Após os oito momentos temporais demonstrados anteriormente, podemos analisar graficamente a evolução da tendência, ao longo do tempo, na seguinte Figura 5.1.

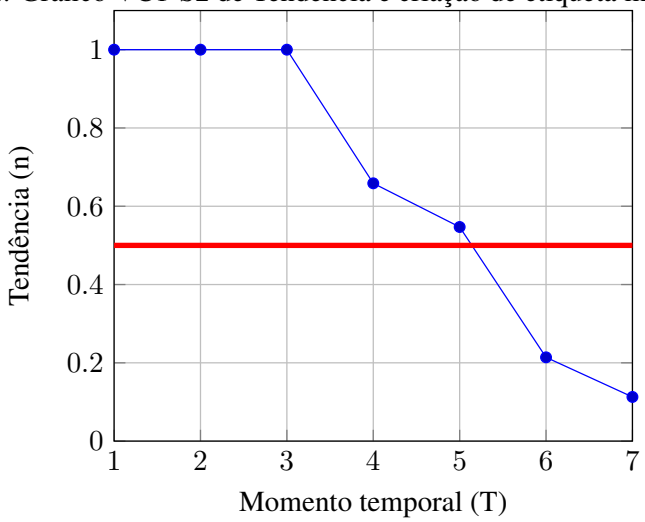
Os Gráficos 5.1, 5.2, 5.3 e 5.4, abaixo permitem analisar e verificar a evolução da tendência ao longo do tempo.

Figura 5.1: Gráfico VC1-S1 de Tendência e criação de etiqueta final em t=8



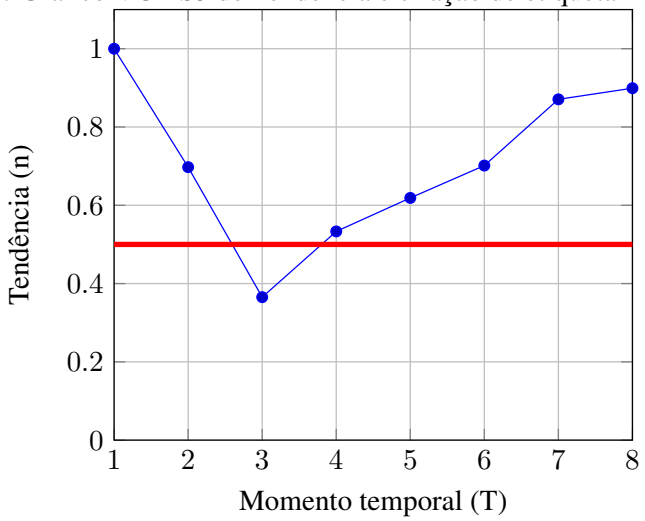

Para a *slice* seguinte representada no Gráfico 5.2 VC1-S2, no momento temporal 7 é possível verificar que a tendência aguentou até ao tempo 3, mas depois desceu abruptamente para valores próximos de 0, a partir do momento 4 até ao 7, mostrando assim que aquela *slice* é claramente não vulnerável.

Figura 5.2: Gráfico VC1-S2 de Tendência e criação de etiqueta final em t=7



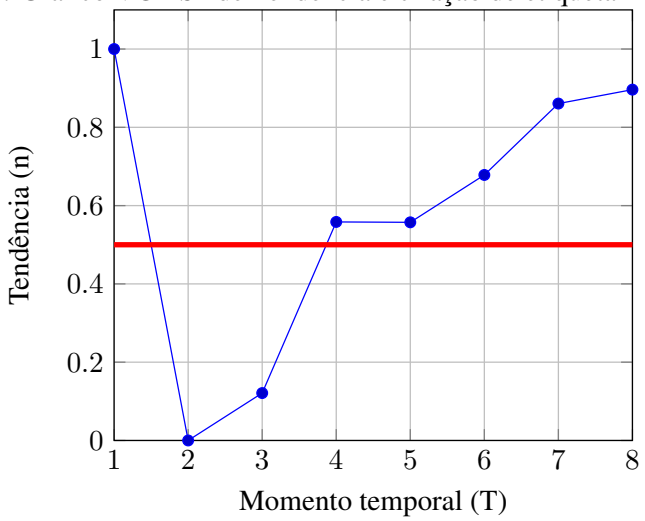

A mesma linha de ação é verificada na *slice* VC1-S3 5.3 ao criar uma etiqueta final no momento temporal 8, mas desta vez, começou como vulnerável, passou a linha de indecisão para não vulnerável e no final revelou-se como uma *slice* vulnerável.

Figura 5.3: Gráfico VC1-S3 de Tendência e criação de etiqueta final em t=8



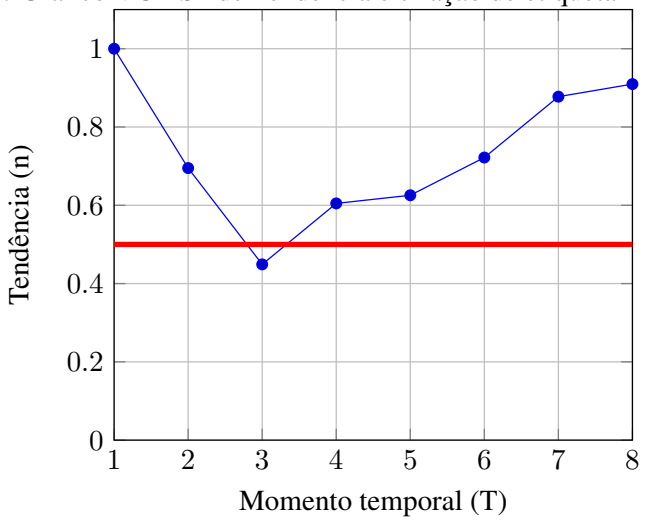

Seguiu o mesmo exemplo da *slice* VC1-S3, esta VC1-S4 5.4 ao criar uma etiqueta final no momento temporal 8, começou como vulnerável, passou a linha de indecisão de forma abrupta para não vulnerável e no final cresceu de forma exponencial para uma *slice* vulnerável.

Figura 5.4: Gráfico VC1-S4 de Tendência e criação de etiqueta final em t=8




No seguinte Gráfico 5.5 a *slice* pertence a outra VC, a 2, e foi estimada a tendência e criação da etiqueta final ao final do tempo 8. Podemos verificar que a *slice* começou como vulnerável, passou a linha de indecisão e demonstrou-se como etiqueta final vulnerável.

Figura 5.5: Gráfico VC2-S1 de Tendência e criação de etiqueta final em t=8

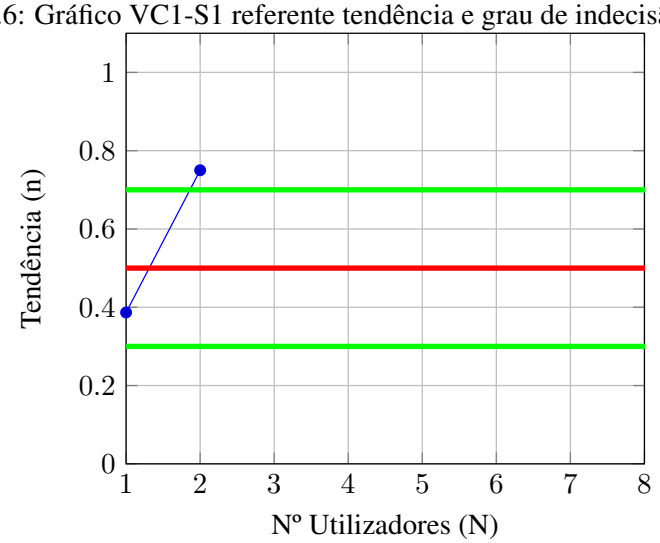



### 5.1.3 Indecisão

De seguida são apresentados os gráficos que demonstram fase a fase o comportamento da tendência e grau de indecisão. Deste modo são apresentados no eixo $x$ o número de utilizadores por momento temporal t=8, definido como $N$, e no eixo $y$ definida a tendência da *slice* consoante cada utilizador e sua reputação dinâmica $n$.

É apresentado no Gráfico 5.6 VC1-S1 referente à tendência e grau de indecisão no momento temporal 2. Podemos verificar que alguns utilizadores classificaram a *slice* como vulnerável, fazendo a linha de tendência subir.

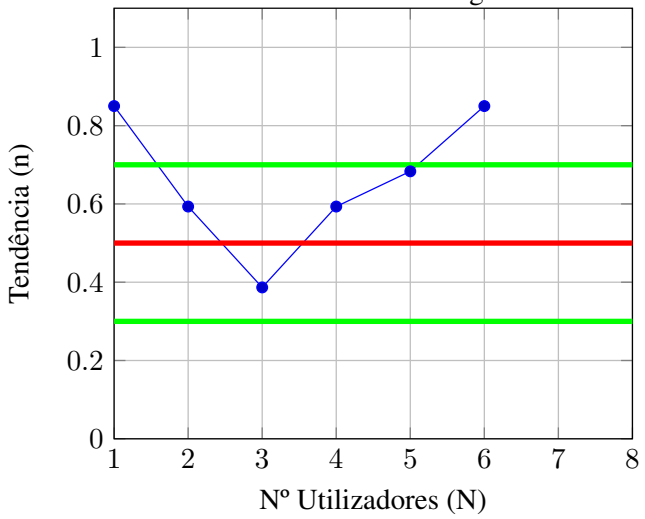
Figura 5.6: Gráfico VC1-S1 referente tendência e grau de indecisão em t=2




É apresentado no Gráfico 5.7 VC1-S1 referente à tendência e grau de indecisão no momento temporal 3, e podemos verificar que alterou totalmente, pois todas as reputações de todos os utilizadores estão em constante mudança. É possível verificar que temos neste momento a tendência

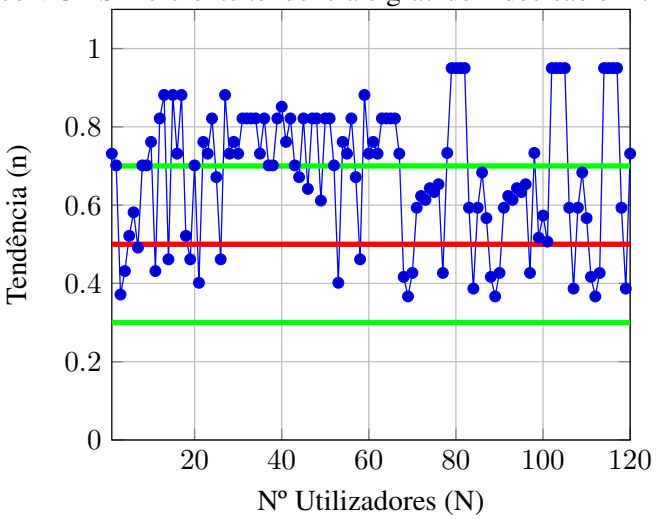
da *slice* está na subida para vulnerável.

Figura 5.7: Gráfico VC1-S1 referente tendência e grau de indecisão em t=3



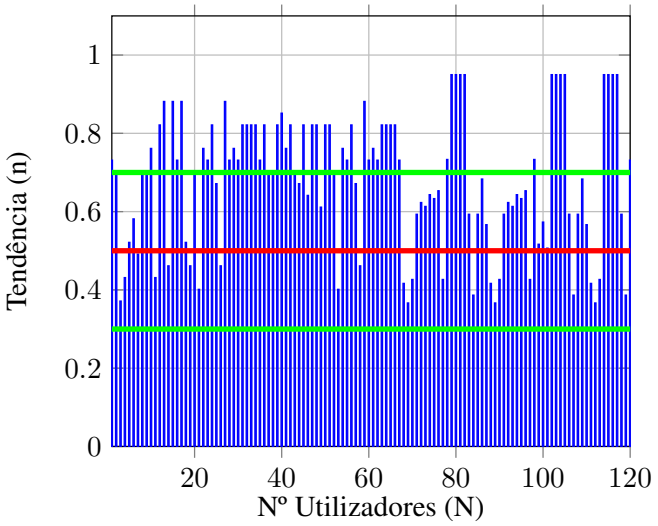

Após estes dois momentos temporais demonstrados anteriormente, é possível ainda analisar graficamente a evolução da reputação do utilizador ao longo do tempo 8, representado na seguinte figura 5.8:

Figura 5.8: Gráfico VC1-S1 referente tendência e grau de indecisão em t=8 n=56 e N=120




Como podemos verificar na figura em cima, na Figura 5.8, temos um ***N*** total de utilizadores de 120 e um ***n*** de 56 utilizadores no grau de indecisão.

Figura 5.9: Gráfico de barras VC1-S1 referente tendência e grau de indecisão em t=8 n=56 e N=120




Os mesmos cálculos efetuados para a VC1-S1 descritos anteriormente foram feitos também para a VC1-S2 ,S3 e S4, bem como para a VC2-S1.

Figura 5.10: Gráfico VC1-S2 referente tendência e grau de indecisão em t=7 n=21 e N=30

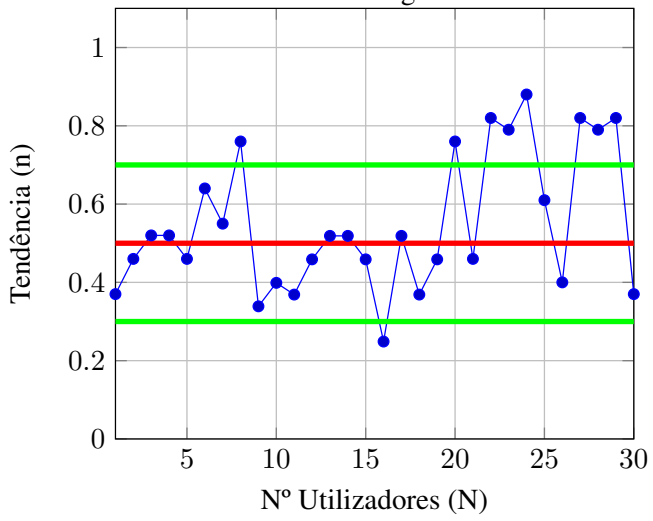


Figura 5.11: Gráfico VC1-S3 referente tendência e grau de indecisão em t=8 n=57 e N=120

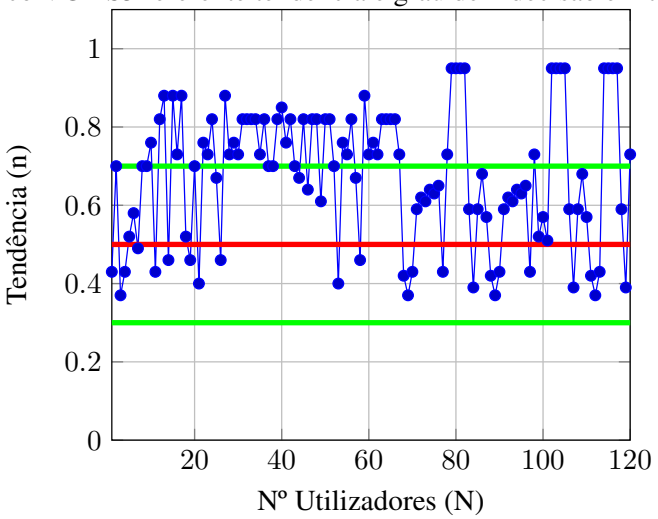



Figura 5.12: Gráfico VC1-S4 referente tendência e grau de indecisão em t=8 n=33 e N=120

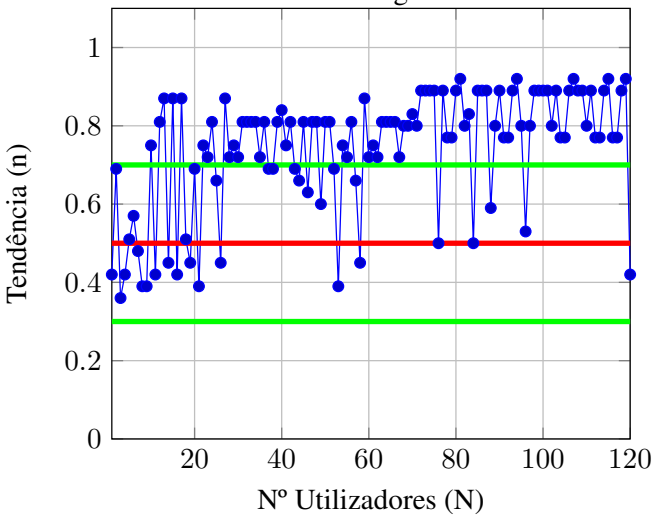
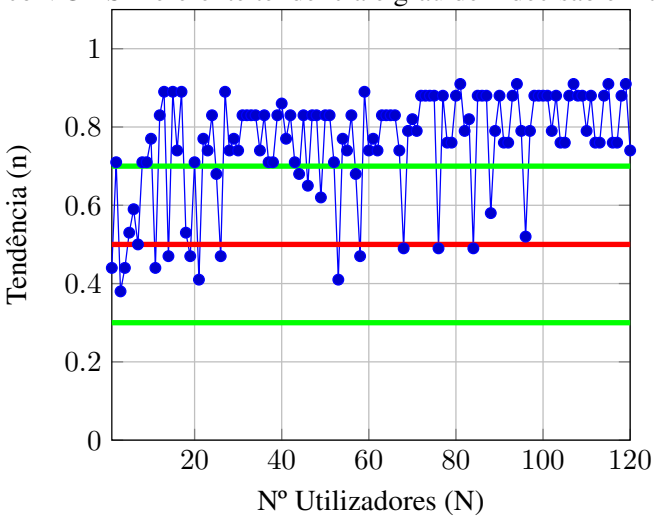


Figura 5.13: Gráfico VC2-S1 referente tendência e grau de indecisão em t=8 n=24 e N=120



# Capítulo 6

# Conclusão

Este trabalho apresenta a evolução do antigo BugSpotting 1.0, uma plataforma *web* desenvolvida para a criação de *datasets* compostos por *slices* de código PHP, classificados como vulneráveis ou não vulneráveis para determinadas VC's. Esta nova versão, BugSpotting 2.0, introduziu melhorias significativas, especialmente na atribuição de *slices* aos utilizadores, *interface*, fase de reputação inicial e dinâmica, bem como no aperfeiçoamento dos algoritmos.

Esta aplicação manteve as boas práticas de modo a garantir uma experiência de utilização simplificada. Foram introduzidas novas funcionalidades para uma melhor gestão da reputação dos utilizadores e revisão de classificações anteriores e tendência para as próximas *slices* a classificar.

Apesar de algumas limitações, tais como, a necessidade de uma maior participação de utilizadores para uma validação mais abrangente, os resultados obtidos foram promissores. Assim, esta aplicação demonstrou um funcionamento robusto e eficaz, com potencial para futuras melhorias explicadas de seguida, consolidando-se como uma ferramenta relevante na deteção e classificação de vulnerabilidades em código PHP.

## 6.1 Trabalho Futuro

Numa versão posterior, o BugSpotting 2.0 poderá obter ainda algumas melhorias, nomeadamente:

- **Desenvolvimento de uma versão móvel** do BugSpotting, o que iria permitir a obtenção de um maior volume de classificações, tornando a plataforma mais acessível e aumentando o envolvimento dos utilizadores, bem como a criação mais rápida do *dataset*.

- **Expansão da gamificação**, através da introdução de níveis, *badges*, recompensas e desafios temáticos que incentivem a participação ativa. Poderiam ser implementadas missões com o objetivo de aumentar a motivação dos utilizadores.

- **Expansão para outras linguagens de programação**, permitindo que o BugSpotting seja utilizado para a análise de vulnerabilidades noutras linguagens de programação como *JavaScript, Python* ou C.

- **Sistema de *chat* entre utilizadores**, onde poderiam falar entre eles sobre alguma das vulnerabilidades de determinada *slice*, promovendo o espírito de equipa, cooperação e colaboração entre eles.
- **Deteção de utilizadores mal-intencionados**, permitindo ao BugSpotting detetar ou identificar padrões de resposta incoerentes ou até múltiplas classificações feitas de forma anormal, assim seria possível mitigar potenciais tentativas de manipulação dos dados.
- **Grau de dificuldade das *slices***, poderá ter outras formas de avaliação da dificuldade da *slice* associadas ao seu conteúdo, tal como condições, por exemplo, *if, else, foreach*, ciclos ou variáveis, de modo a que seja possível definir um valor para a variável $slice_reput_dif ficulty$, de forma automática.

Com estas melhorias, o BugSpotting poderá tornar-se ainda mais robusto e eficiente para os utilizadores, garantindo classificações mais precisas e fiáveis na deteção de vulnerabilidades.

# Bibliografia


[1] A. Ahmad, F. Zeshan, M. S. Khan, R. Marriam, A. Ali, and A. Samreen. The impact of gamification on learning outcomes of computer science majors, 2020. ACM Transactions on Computing Education, vol.20, no. 2, pp. 1–25, apr.

[2] Imam Ahmad, Emi Suwarni, Rohmat Indra Borman, Asmawati, Farli Rossi, and Yessi Jusman. Implementation of restful api web services architecture in takeaway application development, 2021. 2021 1st International Conference on Electronic and Electrical Engineering and Intelligent System (ICE3IS).

[3] M. M. Alhammad and A. M. Moreno. Gamification in software engineering education: A systematic mapping, 2018. Journal of Systems and Software, vol. 141, pp. 131–150, jul.

[4] P. E. Black. A software assurance reference dataset: Thousands of programs with known bugs, 2018. Journal of Research of the National Institute of Standards and Technology, vol. 123.

[5] H. Booth, D. Rike, and G. A. Witte. The national vulnerability database (nvd): Overview, 2013. `https://nvd.nist.gov`.

[6] Jiajia Chen and Weiqing Cheng. Analysis of web traffic based on http protocol.

[7] Atul S Choudhary, Pankaj P Choudhary, and Shrikant Salve. A study on various cyber attacks and a proposed intelligent system for monitoring such attacks, 2018. Proceedings of the International Conference on Inventive Computation Technologies (ICICT-2018) IEEE Xplore Part Number: CFP18F70-ART; ISBN:978-1-5386-4985-5.

[8] Sana Conté. Software weaknesses detection using static-code analysis and machine learning techniques, 2023. Licenciatura em Engenharia Informática.

[9] Miguel Ferreira. Development of a website for creation of vulnerability datasets. Master's thesis, Faculdade de Ciências da Universidade de Lisboa, 2022.

[10] Hector Garcia-Molina, Manas Joglekar, Adam Marcus, Aditya Parameswaran, and Vasilis Verroios. Challenges in data crowdsourcing, 2016. IEEE Transactions on Knowledge and Data Engineering, VOL. 28, NO. 4, April.

[11] Kashish Gaur, Manoj Diwakar, Kaamya Gaur, Prabhishek Singh, Tanya Sachdeva, and Neeraj Kumar Pandey. Sql injection attacks and prevention, 2023. 2023 6th International Conference on Information Systems and Computer Networks (ISCON).

[12] A. Iosup and D. Epema. An experience report on using gamification in technical higher education, 2014. In Proceedings of the 45th ACM technical symposium on Computer science education. ACM, mar.

[13] Kaspersky. O que é um ataque de cross-site scripting? https://www.kaspersky.com.br/resource-center/definitions/what-is-a-cross-site-scripting-attack.

[14] Pankaj Keshari, Priya Maurya, Galgotias University, Dr. Alok Katiyar, and Pankaj Kumar. Web development using reactjs, 2023. 2023 5th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N).

[15] Yogesh Kumar, Komalpreet Kaur, and Gurpreet Singh. Machine learning aspects and its applications towards different research areas, 2020. International Conference on Computation, Automation and Knowledge Management (ICCAKM) Amity University.

[16] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities, 2001.

[17] Xiaohong Li and Na Liu. Research on l-mvc framework, 2016.

[18] Jane McGonigal. Reality is broken: Why games make us better and how they can change the world, 2011.

[19] Mckinsey. What is machine learning?, 2024. https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-machine-learning.

[20] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Dekant: A static analysis tool that learns to detect web application vulnerabilities, 2016.

[21] Computer Security Resource Center NIST, Information Technology Laboratory. Vulnerability. https://csrc.nist.gov/glossary/term/vulnerability.

[22] OffSec. File inclusion vulnerabilities. https://www.offsec.com/metasploit-unleashed/file-inclusion-vulnerabilities.

[23] PortSwigger. Os command injection. https://portswigger.net/web-security/os-command-injection.

[24] PortSwigger. Path traversal. https://portswigger.net/web-security/file-path-traversal.

[25] PortSwigger. Php code injection. https://portswigger.net/kb/issues/00100c00_php-code-injection.

[26] PortSwigger. Source code disclosure. https://portswigger.net/kb/issues/006000b0_source-code-disclosure.

[27] Kavya Rani S R, Soundarya B C, and Gururaj H L. Comprehensive analysis of various cyber attacks, 2021. EEE Mysore Sub Section International Conference(MysuruCon).

[28] IEEE Network Generations Roadmap. Artificial intelligence and machine learning, 2022. IEEE International Network Generations Roadmap.

[29] Rebecca L. Russell, Louis Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning, 2018. 17th IEEE International Conference on Machine Learning and Applications.

[30] Tommaso Dal Sasso, Andrea Mocci, Michele Lanza, and Ebrisa Mastrodicasa. How to gamify software engineering, 2017.

[31] Sandro Speth, Leon Hofmeister, Steffen Becker, and Uwe Breitenbucher. Gamify-it – a web-based gaming platform for software engineering education, 2023. IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG).

[32] Sandro Speth, Leon Hofmeister, Steffen Becker, and Uwe Breitenbucher. Gamify-it – a web-based gaming platform for software engineering education, 2023. IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG).

[33] Fang Wu, Jigang Wang, Jiqiang Liu, and Wei Wang. Vulnerability detection with deep learning, 2013. Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, 3rd IEEE International Conference on Computer and Communications.

[34] Man-Ching Yuen, Irwin King, and Kwong-Sak Leung. A survey of crowdsourcing systems, 2011. IEEE International Conference on Privacy, Security, Risk, and Trust, and IEEE International Conference on Social Computing.

[35] Noah Ziems and Shaoen Wu. Security vulnerability detection using deep learning natural language processing, 2021. IEEE INFOCOM WKSHPS: BigSecurity 2021.