Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Cibersegurança e Informática Forense

EXPLORAÇÃO DE VULNERABILIDADES COM ALGORITMOS DE INTELIGÊNCIA ARTIFICIAL

KEVIN BAPTISTA

Leiria, novembro de 2021

Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Cibersegurança e Informática Forense

EXPLORAÇÃO DE VULNERABILIDADES COM ALGORITMOS DE INTELIGÊNCIA ARTIFICIAL

KEVIN BAPTISTA
Número: 2190371

Projeto realizado sob orientação da Professora Doutora Anabela Bernardino e da Professora Doutora Eugénia Bernardino, na Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria.

Leiria, novembro de 2021

RESUMO

O presente relatório foi elaborado no âmbito do projeto do Mestrado em Cibersegurança e Informática Forense, da Escola Superior de Tecnologia e Gestão (ESTG), do Instituto Politécnico de Leiria (IPLeiria) e tem como tema: *Exploração de Vulnerabilidades com Algoritmos de Inteligência Artificial*.

Os ataques a sistemas de informação estão a tornar-se cada vez mais sofisticados e catastróficos, tendo como resultado a perda de informação pessoal e, até, em casos extremos, a perda de vidas humanas. Os criminosos aproveitam-se muitas vezes de falhas nos sistemas para desencadear ataques. A *Open Web Application Security Project* (OWASP) (fundação sem fins-lucrativos que trabalha no sentido de melhorar a segurança informática) considerou, em 2017, *injeção* como um dos maiores riscos em Aplicações *web*. Nesta medida, existem várias ferramentas automatizadas com o intuito de auxiliar profissionais da área a identificar estas vulnerabilidades. No entanto, manter estas ferramentas atualizadas com a evolução tecnológica tem-se demonstrado um desafio. Para identificar e explorar vulnerabilidades de um sistema podem ser utilizados algoritmos de Inteligência Artificial (IA).

Neste trabalho é proposta e desenvolvida uma abordagem recorrendo a IA para identificar vulnerabilidades de *Structured Query Language* (SQL) *Injection*. Esta abordagem é dividida em duas fases. Numa primeira fase procura por *queries* SQL no código-fonte de uma aplicação *web* *Hypertext Preprocessor* (PHP) e, numa segunda fase são aplicados algoritmos de IA para procura dos melhores vetores de ataque. Neste sentido, foram estudados três algoritmos de IA: Algoritmos Genéticos (AG), *Artificial Bee Colony* (ABC) e *Ant Colony Optimization* (ACO). Para cada algoritmo implementado, foi efetuado um estudo para encontrar os parâmetros de entrada que obtêm os melhores resultados.

Foram ainda desenvolvidas duas modelações para este problema, em que a representação de um indivíduo difere. Numa representação, cada gene de um indivíduo é um vetor de ataque completo, enquanto noutra, o indivíduo completo é um vetor de ataque.

Para testar empiricamente os valores obtidos, a ferramenta foi aplicada a projetos de código aberto, propositadamente vulneráveis: *Bricks*, *bWAPP* e *Twitterlike*.

A abordagem desenvolvida obteve resultados muito satisfatórios na deteção de vulnerabilidades do tipo SQL *Injection*, quando comparada com outras ferramentas: *Web Application Protection* (WAP) e *SonarPHP*.

Palavras-chave: Algoritmos Genéticos, *Artificial Bee Colony*, *Ant Colony Optimization*, SQL *Injection*, Segurança *Web*.

ABSTRACT

This report describes all the work done within the scope of the master's project of Cybersecurity and Digital Forensics, School of Technology and Management of the Polytechnic of Leiria, with the subject *Exploring vulnerabilities using Artificial Intelligence Algorithms.*

Attacks to information systems are becoming every year more sophisticated and catastrophic, resulting many times in loss of personal data and, in extreme cases, human lives. Criminals often take advantage of flaws in systems to trigger attacks. OWASP (non-profit foundation that works to improve the security of software) considers "Injection"one of the most common Web Application Security Risks. There are many automated tools to help professionals in the field in order to identify this vulnerability. However, keeping these tools up to date has proven to be a challenge, thus, there has been some interest in applying Artificial Intelligence (AI) in this field.

In this research, we propose and develop an approach to detect SQL injection vulnerabilities in the source code, using AI algorithms. This approach is divided in two steps. Firstly, the search for SQL queries in the source code of a PHP web application and, afterwards, the use of AI algorithms. In fact, three algorithms were implemented and tested: Genetic Algorithm, ABC, ACO. For each, a study was conduct in order to find the best parameters.

Two models were developed, in which the individual representation is different. In one model, each gene of an individual is a complete attack vector, while in the other model, only the all individual is a complete attack vector.

To test this approach empirically we used web applications purposefully vulnerable as Bricks, bWAPP, and Twitterlike.

This approach got very satisfying results to detect SQL Injection vulnerabilities, when comparing with other static tools: WAP and *SonarPHP*.

Keywords: Genetic Algorithms, Artificial Bee Colony, Ant Colony Optimization, SQL Injection, web security.

AGRADECIMENTOS

Apresento, de seguida, os meus agradecimentos a todos os que contribuíram, sob as mais diversas formas, para o desenvolvimento deste projeto:

Às professoras orientadoras, Professora Anabela Bernardino e Professora Eugénia Bernardino, por todo o tempo, auxílio e encorajamento durante o processo de desenvolvimento deste trabalho. O conhecimento principalmente na área da inteligência artificial foi fulcral para o êxito deste projeto. Quero ainda deixar um agradecimento pelo incentivo e colaboração nos artigos desenvolvidos e publicados nas conferências *ICS 2021: 15. International Conference on Security*, *18th International Conference Applied Computing* e *ICITS22 - The 2022 International Conference on Information Technology & Systems*.

À Fundação para a Ciência e a Tecnologia (FCT), pelo suporte na participação nas várias conferências. Sem esta, as publicações e contribuições para a comunidade científica, não teriam sido possíveis.

Ao Instituto Politécnico de Leiria, pelas condições e incentivo à realização deste projeto.

Por fim, reservo os agradecimentos finais para a minha família que esteve presente em todos os momentos.

ÍNDICE

Resumo i
Abstract iii
Agradecimentos iv
Índice v
Lista de Figuras x
Lista de Tabelas xii
Lista de Listagens xiii
Lista de Abreviaturas xvi

1 INTRODUÇÃO 1
1.1 Motivação e Objetivos . . . 1
1.2 Cronograma . . . 2
1.3 Estrutura do documento . . . 5

2 ENQUADRAMENTO 7
2.1 Vulnerabilidades de Segurança . . . 7
2.1.1 Injeção . . . 8
2.1.2 Falhas de Autenticação . . . 8
2.1.3 Expor dados sensíveis . . . 8
2.1.4 *XML External Entities* . . . 9
2.1.5 Falha nos controlos de acesso . . . 9
2.1.6 Incorretas configurações de segurança . . . 9
2.1.7 *Cross-Site Scripting* . . . 9
2.1.8 Desserialização insegura . . . 10
2.1.9 Usar componentes com vulnerabilidades conhecidas . . . 10
2.1.10 Insuficiente monitorização . . . 10
2.2 *SQL Injection* . . . 10
2.2.1 Exemplo de um ataque simples . . . 11
2.2.2 Obter o esquema da base dados . . . 12
2.2.3 Prevenir ataques de *SQL Injection* . . . 13
2.3 Algoritmos de Inteligência Artificial . . . 14
2.3.1 Algoritmos Genéticos . . . 14

2.3.2 *Artificial Bee Colony* 20
2.3.3 *Ant Colony Optimization* 22
2.4 Estado da Arte 23
2.5 Estudo de Problemas 26
2.5.1 Instalação local 27
2.5.2 *Bricks* 28
2.5.3 *bWAPP* 34
2.5.4 *Twitterlike* 36
2.6 Ferramentas automáticas 36
2.6.1 *Web Application Protection* 37
2.6.2 *SonarPHP* 37

3 METODOLOGIA 39
3.1 Quadro *Kanban* 39
3.2 Cartões *Kanban* 39
3.3 *Work In Progress* 40
3.4 *backlogs* 40
3.5 *Continuous releases* 41
3.6 Sumário 41

4 DESENVOLVIMENTO 42
4.1 Modelação do problema 42
4.1.1 Fase preliminar 43
4.1.2 Fase de deteção 44
4.2 Abordagem Técnica — Algoritmos Genéticos 45
4.2.1 Parâmetros de entrada 45
4.2.2 Representação do Indivíduo 46
4.2.3 Geração da população inicial 47
4.2.4 Operadores Genéticos 47
4.2.5 *Fitness* 48
4.3 Abordagem Técnica — *Artificial Bee Colony* 50
4.3.1 Parâmetros de entrada 51
4.3.2 Fase das abelhas empregadas 51
4.3.3 Fase das abelhas espetadoras 52
4.3.4 Fase das abelhas exploradoras 53
4.4 Abordagem Técnica — *Ant Colony Optimization* 53
4.4.1 Parâmetros de entrada 54
4.4.2 Inicialização de valores de feromonas 55

4.4.3 Modificação da solução . . . 55
4.4.4 Mecanismo de intensificação . . . 56
4.4.5 Atualização do rastro de feromonas . . . 56
4.5 *Cache* . . . 56
4.6 *Interface GUI* . . . 59
4.7 Linha de comandos . . . 60
4.8 Configurações . . . 60
4.9 Tecnologias usadas . . . 62
4.10 Diagrama da aplicação . . . 64

5 EXPERIÊNCIAS E RESULTADOS 68
5.1 Algoritmo Genético - Versão 1 . . . 69
5.1.1 Influência do número de gerações . . . 70
5.1.2 Influência do tamanho da população . . . 70
5.1.3 Influência do Elitismo . . . 72
5.1.4 Influência dos métodos de seleção . . . 72
5.1.5 Influência do tamanho do torneio . . . 73
5.1.6 Influência dos métodos de recombinação . . . 73
5.1.7 Influência das probabilidades de recombinação e mutação . . . 73
5.1.8 Melhores parâmetros e resultados . . . 74
5.2 Algoritmo Genético - Versão 2 . . . 75
5.2.1 Influência do número de gerações . . . 75
5.2.2 Influência do tamanho da população . . . 75
5.2.3 Influência do elitismo . . . 76
5.2.4 Influência dos métodos de seleção e tamanho do torneio . . . 77
5.2.5 Influência dos métodos de recombinação . . . 77
5.2.6 Influência das probabilidades de recombinação e mutação . . . 77
5.2.7 Melhores parâmetros e resultados . . . 78
5.3 *Artificial Bee Colony* - Versão 1 . . . 78
5.3.1 Influência do número máximo de iterações . . . 79
5.3.2 Influência do número de abelhas empregadas . . . 79
5.3.3 Influência do número de abelhas espetadoras . . . 80
5.3.4 Influência do número de modificações . . . 81
5.3.5 Melhores parâmetros e resultados . . . 81
5.4 *Artificial Bee Colony* - Versão 2 . . . 82
5.4.1 Influência do número máximo de iterações . . . 82
5.4.2 Influência do número de abelhas empregadas . . . 82
5.4.3 Influência do número de abelhas espetadoras . . . 83

5.4.4 Influência do número de modificações . . . 83
5.4.5 Melhores parâmetros e resultados . . . 84
5.5 *Ant Colony Optimization* - Versão 1 . . . 84
5.5.1 Influência do número máximo de iterações . . . 85
5.5.2 Influência do número de iterações máximo sem melhorias . . . 85
5.5.3 Influência do tamanho da colónia . . . 86
5.5.4 Influência dos parâmetros de probabilidades . . . 86
5.5.5 Melhores parâmetros e resultados . . . 87
5.6 *Ant Colony Optimization* - Versão 2 . . . 88
5.6.1 Influência do número de iterações máximo sem melhorias . . . 88
5.6.2 Influência do tamanho da colónia . . . 88
5.6.3 Influência dos parâmetros de probabilidades . . . 89
5.6.4 Melhores parâmetros e resultados . . . 90
5.7 Resultado de outras ferramentas . . . 90
5.7.1 *Web Application Protection* . . . 90
5.7.2 *Sonar PHP* . . . 91
5.8 Resultados e análise . . . 92
5.8.1 Estudo comparativo entre versões e algoritmos desenvolvidos 93
5.8.2 Melhores indivíduos . . . 94
5.8.3 Estudo comparativo com outras ferramentas . . . 96

6 CONCLUSÃO 98
6.1 Sumário e Análise Crítica . . . 98
6.2 Limitações . . . 99
6.3 Trabalho Futuro . . . 100
6.4 Contribuições . . . 101

BIBLIOGRAFIA 102

Apêndices

A CONFIGURAÇÕES 108
A.1 Ficheiro *Docker* para instalação dos projetos estudados numa máquina local . . . 108
B INICIALIZAÇÃO DOS PROJETOS ESTUDADOS 109
B.1 Inicializar *Bricks* . . . 109
B.2 Inicializar *bWAPP* . . . 110
B.3 Inicializar *Twitterlike* . . . 112

DECLARAÇÃO 115

LISTA DE FIGURAS

Figura 1 Cronograma do trabalho realizado 4
Figura 2 Mecanismo de distribuição de seleção usando *Roulette Wheel* 16
Figura 3 Método de recombinação com um corte 18
Figura 4 Método de recombinação com dois cortes 19
Figura 5 Método de recombinação uniforme 19
Figura 6 Exemplo de mutação 20
Figura 7 Pedido intercetado pelo BurpSuite 29
Figura 8 Alteração do *User Agent* de um pedido 33
Figura 9 Campo de pesquisa no *bWAPP* 35
Figura 10 Obtenção do nome das tabelas do *bWAPP* 36
Figura 11 Exemplo de um quadro de *Kanban* 40
Figura 12 Fase preliminar 43
Figura 13 Fase de deteção de vulnerabilidades 44
Figura 14 Etapas do AG aplicado a SQL *Injection* 45
Figura 15 Representação de um indivíduo - versão 1 46
Figura 16 Representação de um indivíduo - versão 2 46
Figura 17 Exemplo de uma população 47
Figura 18 Exemplo de uma recombinação aplicada ao problema SQL *Injection* 47
Figura 19 Exemplo de uma mutação aplicada ao problema SQL *Injection* 48
Figura 20 Etapas para calcular o *fitness* 48
Figura 21 Exemplo do indivíduo, versão 2, aplicado a uma *query* . . . 49
Figura 22 *Interface* gráfica da ferramenta desenvolvida 59
Figura 23 Classes definidas para os Operadores Genéticos 64
Figura 24 Classes utilizadas pelo algoritmo *Artificial Bee Colony* para a representação da pesquisa local 64
Figura 25 Classes definidas para a representação dos Algoritmos 65
Figura 26 Classes definidas para a representação do indivíduo 66
Figura 27 Classes definidas para a representação do problema 67
Figura 28 Classes definidas para a representação da população 67
Figura 29 Efeito do número máximo de gerações no *fitness* - versão 1 . 70
Figura 30 Efeito do tamanho da população no valor de *fitness* - versão 1 71

Figura 31 Influência do tamanho da população no valor de *fitness* com limite de 30 segundos - versão 1 71
Figura 32 Influência de métodos de seleção no valor de *fitness* - versão 1 72
Figura 33 Influência do tamanho do torneio no valor de *fitness* - versão 1 73
Figura 34 Influência dos métodos de recombinação no valor de *fitness* - versão 1 73
Figura 35 Influência das probabilidades de mutação e recombinação no valor de *fitness* - versão 1 74
Figura 36 Efeito do número máximo de gerações no *fitness* - versão 2 . 75
Figura 37 Efeito do tamanho da população no valor de *fitness* - versão 2 76
Figura 38 Efeito do elitismo no valor de *fitness* - versão 2 76
Figura 39 Influência das probabilidades de mutação e recombinação no valor de *fitness* - versão 2 77
Figura 40 Influência do número máximo de iterações no valor de *fitness* - versão 1 79
Figura 41 Efeito do número de abelhas empregadas no valor de *fitness* - versão 1 79
Figura 42 Influência do número de abelhas empregadas no valor de *fitness* com limite de 30 segundos - versão 1 80
Figura 43 Influência do número de abelhas espetadoras no valor de *fitness* - versão 1 80
Figura 44 Influência do número de modificações no valor de *fitness* - versão 1 81
Figura 45 Influência do número máximo de iterações no valor de *fitness* - versão 2 82
Figura 46 Efeito do número de abelhas empregadas no valor de *fitness* - versão 2 82
Figura 47 Influência do número de abelhas espetadoras no valor de *fitness* - versão 2 83
Figura 48 Influência do número de modificações no valor de *fitness* - versão 2 83
Figura 49 Efeito do número de iterações no valor de *fitness* - versão 1 . 85
Figura 50 Efeito do número de iterações sem melhorias no valor de *fitness* - versão 1 86
Figura 51 Efeito do tamanho da colónia no valor de *fitness* - versão 1 . 86
Figura 52 Efeito das várias probabilidades no valor de *fitness* - versão 1 87
Figura 53 Efeito do número de iterações sem melhorias no valor de *fitness* - versão 2 88

Figura 54 Efeito do tamanho da colónia no valor de *fitness* - versão 2 . 89
Figura 55 Efeito do número de iterações sem melhorias no valor de *fitness* - versão 2 89
Figura 56 Resultados da análise do *Twitterlike* com a ferramenta WAP 91
Figura 57 *Interface* do *Sonar PHP* com os resultados da análise do *Bricks* 92
Figura 58 Comparação do número de vulnerabilidades detetadas por cada algoritmo e versão 93
Figura 59 Comparação do desvio padrão entre os vários algoritmos e aplicações web estudadas 94
Figura 60 Comparação da eficácia das várias ferramentas com a abordagem usando algoritmos genéticos 97

LISTA DE TABELAS

Tabela 1 Resultado de um esquema de uma Base de Dados 13
Tabela 2 Valores *fitness* e percentagem total de uma população 17
Tabela 3 Exemplo de uma seleção por torneio de tamanho 2 17
Tabela 4 Número de vulnerabilidades identificadas manualmente . . . 68
Tabela 5 Efeito do elitismo no valor de *fitness* - versão 1 72
Tabela 6 Parâmetros que obtiveram melhores resultados - versão 1 . . 74
Tabela 7 Vulnerabilidades detetadas pelo AG - versão 1 75
Tabela 8 Parâmetros que obtiveram melhores resultados - versão 2 . . 78
Tabela 9 Vulnerabilidades detetadas pelo AG - versão 2 78
Tabela 10 Parâmetros do algoritmo ABC que obtiveram melhores resultados - versão 1 81
Tabela 11 Vulnerabilidades detetadas pelo algoritmo ABC - versão 1 . 81
Tabela 12 Parâmetros do algoritmo ABC que obtiveram melhores resultados - versão 2 84
Tabela 13 Vulnerabilidades detetadas pelo algoritmo ABC - versão 2 . 84
Tabela 14 Parâmetros do algoritmo ACO que obtiveram melhores resultados - versão 1 87
Tabela 15 Vulnerabilidades detetadas pelo algoritmo ACO - versão 1 . 88
Tabela 16 Parâmetros do algoritmo ACO que obtiveram melhores resultados - versão 2 90
Tabela 17 Vulnerabilidades detetadas pelo algoritmo ACO - versão 2 . 90
Tabela 18 Vulnerabilidades detetadas pelo *Web Application Protection* 91
Tabela 19 Vulnerabilidades detetadas pelo *Sonar PHP* 92
Tabela 20 Comparação entre os algoritmos analisados 93
Tabela 21 Comparação entre o AG desenvolvido e outras ferramentas de análise estática 96

LISTA DE LISTAGENS

Listagem 1 Exemplo de uma *query* para autenticar um utilizador 11
Listagem 2 Exemplo de uma *query* para autenticar um utilizador com valores introduzidos 11
Listagem 3 Exemplo de uma *query* com parâmetros maliciosos injetados 12
Listagem 4 *Query* base para obter o nome da tabela 12
Listagem 5 Exemplo de um vetor de ataque 12
Listagem 6 Pseudocódigo das etapas do algoritmo genético 15
Listagem 7 Pseudocódigo das etapas do *Artificial Bee Colony* 22
Listagem 8 Pseudocódigo das etapas do *Ant Colony Optimization* 23
Listagem 9 Extrato de código da página *Login-1* do *Bricks* 28
Listagem 10 Possíveis vetores de ataque para *Login-1* do *Bricks* 28
Listagem 11 Extrato de código da página *Login-3* do *Bricks* 30
Listagem 12 Possíveis vetores de ataque para *Login-3* do *Bricks* 30
Listagem 13 Extrato de código da página *Login-4* do *Bricks* 30
Listagem 14 Possíveis vetores de ataque para *Login-4* do *Bricks* 31
Listagem 15 Extrato de código da página *Login-5* do *Bricks* 31
Listagem 16 Possíveis vetores de ataque para *Login-5* do *Bricks* 31
Listagem 17 Extrato de código da página *Login-6* do *Bricks* 31
Listagem 18 Extrato de código da página *Content-1* do *Bricks* 32
Listagem 19 Possíveis vetores de ataque para *Content-1* do *Bricks* . . . 32
Listagem 20 Extrato de código da página *Content-2* do *Bricks* 32
Listagem 21 Extrato de código da página *Content-3* do *Bricks* 33
Listagem 22 Extrato de código da página *Content-4* do *Bricks* 33
Listagem 23 Extrato de código da página *Content-6* do *Bricks* 34
Listagem 24 Código *SQL* executado durante uma pesquisa no *bWAPP* . 35
Listagem 25 Possível vetor de ataque para o campo de pesquisa do *bWAPP* 35
Listagem 26 Extrato de código do algoritmo *Artificial Bee Colony* 51
Listagem 27 Extrato de código do algoritmo *Ant Colony Optimization* . . 54
Listagem 28 Extrato de código para *Cache* 58
Listagem 29 Extrato de código para execução do sistema de *cache* 59
Listagem 30 Execução através da linha de comandos 60
Listagem 31 Exemplo de um ficheiro de configurações 61

Listagem 32 Ficheiro para criar e popular uma base de dados 63
Listagem 33 Ficheiro para criar e popular dados para o projeto *Bricks* . . 64
Listagem 34 Ficheiro para instalação de projetos 108
Listagem 35 Ficheiro para criar e popular o projeto *Bricks* 109
Listagem 36 Ficheiro para criar e popular o projeto *bWAPP* 112
Listagem 37 Ficheiro para criar e popular o projeto *Twitterlike* 114

LISTA DE ABREVIATURAS

ABC	*Artificial Bee Colony*
ACO	*Ant Colony Optimization*
AG	Algoritmos Genéticos
API	*Application Programming Interface*
bWAPP	*Buggy Web Application*
DML	*Data Manipulation Language*
DOM	*Document Object Model*
DoS	*Denial of Service*
ESTG	Escola Superior de Tecnologia e Gestão
FCT	Fundação para a Ciência e a Tecnologia
FDD	*Feature Driven Development*
GUI	*Graphical User Interface*
IA	Inteligência Artificial
IPLeiria	Instituto Politécnico de Leiria
JAR	*Java Archive*
JIT	*Just In Time*
LDAP	*Lightweight Directory Access Protocol*
LFI	*Local File Inclusion*
MAC	*Message Authentication Algorithm*
ML	*Machine Learning*
MVP	*Minimum Viable Product*
NoSQL	*Non Structured Query Language*
ORM	*Object Relational Mapper*
OSCI	*Operating System Command Injection*
OSI	*Open System Interconnection*
OWASP	*Open Web Application Security Project*

PHP	*Hypertext Preprocessor*
RAM	*Random Access Memory*
RCE	*Remote Code Execution*
RFI	*Remote File Inclusion*
SGBD	Sistema de Gestão de Base de Dados
SO	Sistema Operativo
SQL	*Structured Query Language*
WAP	*Web Application Protection*
WIP	*Work In Progress*
XML	*Extensible Markup Language*
XP	*eXtreme Programming*
XSS	*Cross-Site Scripting*
XXE	*XML External Entities*

1

INTRODUÇÃO

O presente relatório foi elaborado no âmbito do projeto do Mestrado em Cibersegurança e Informática Forense, da ESTG do IPLeiria, e tem como tema: *Exploração de Vulnerabilidades com Algoritmos de Inteligência Artificial*.

No primeiro capítulo deste documento, pretende-se dar a conhecer a motivação para o desenvolvimento do projeto elaborado, assim como os objetivos do mesmo (Secção 1.1). De seguida, é apresentado um cronograma do projeto desenvolvido (Secção 1.2) e, por fim, é apresentada a estrutura e organização do documento (Secção 1.3).

1.1 MOTIVAÇÃO E OBJETIVOS

Nesta nova era da informação, a segurança da informação revela ter um papel cada vez mais importante. A complexidade dos sistemas informáticos faz com que seja difícil afirmar que um sistema está seguro, portanto, é necessário ter mecanismos para aumentar a confiança de um sistema (Stiawan et al., 2017).

O trabalho manual na deteção de vulnerabilidades é fulcral, no entanto, trata-se de um processo lento, propenso ao erro humano e, caro, visto que depende de trabalhadores altamente qualificados.

Nesta medida, tal como observaremos no Capítulo 2, grande percentagem das vulnerabilidades na *web* deve-se a negligência por parte de quem desenvolve e gere sistemas informáticos. Assim, e com base no estudo efetuado, percebeu-se que SQL *Injection* apesar de se tratar de uma vulnerabilidade facilmente corrigida, tem uma grande insurgência na *web*, sendo que o seu impacto é bastante nefasto. Neste sentido, pretende-se desenvolver uma estratégia capaz de auxiliar profissionais da área a prevenir vulnerabilidades a partir do código-fonte.

Neste trabalho, apenas foram estudadas aplicações *web* desenvolvidas em PHP, propositadamente vulneráveis (*Bricks*, *Twitterlike* e *bWAPP*). O interesse na deteção de vulnerabilidades em projetos PHP, deve-se ao facto desta linguagem apresentar uma percentagem bastante significativa das aplicações desenvolvidas na *web*, repre-

sentando uma percentagem de 41%, seguindo-se aplicações desenvolvidas em *ASP .NET* com 22% (Rexhepi et al., 2017).

O principal objetivo desta investigação é desenvolver uma ferramenta capaz de detetar vulnerabilidades de SQL *Injection* o mais rapidamente possível, para auxiliar os profissionais na área. Para o efeito, foi elaborada uma lista de objetivos secundários que devem ser efetuados:

- Identificar vulnerabilidades mais comuns na *web*;
- Efetuar uma revisão literária na área;
- Analisar ferramentas na área para perceber problemas atuais;
- Aplicar vários algoritmos no campo da Inteligência Artificial (IA);
- Avaliar as soluções e compará-las com outras ferramentas;
- Efetuar uma análise crítica da aplicação implementada e da sua viabilidade.

1.2 CRONOGRAMA

O presente relatório de projeto foi realizado no ano letivo de 2020/21, iniciado em outubro de 2020 e com término em novembro de 2021. A Figura 1 ilustra o cronograma de atividades durante o ano letivo.

No planeamento do projeto, em outubro de 2020, foram definidos os objetivos a atingir e delineadas as primeiras etapas a serem percorridas. Tratando-se de um projeto que requeria uma investigação inicial, o mês de novembro de 2020 foi dedicado à investigação de projetos na área. O objetivo primordial era encontrar artigos na área da IA que tivessem sido aplicados na área da cibersegurança para deteção de vulnerabilidades.

Após a análise de um conjunto de artigos e dissertações na área, procedeu-se à filtragem e análise mais profunda de alguns desses artigos. Em dezembro de 2020, o problema foi modelado num problema de otimização. Desta forma, foi desenvolvido um *Minimum Viable Product* (MVP) da ferramenta. Nesta primeira abordagem, apenas eram detetadas vulnerabilidades em *queries* de pesquisa (*SELECT*) e em ficheiros isolados.

Em janeiro de 2021, a ferramenta foi estendida para ser usada em projetos completos, isto é, dada uma diretoria, são detetados e analisados todos os ficheiros recursivamente. Nesta altura, investigaram-se várias aplicações de código-aberto,

desenvolvidas em PHP, que pudessem ser usadas para testar a eficácia do primeiro algoritmo implementado, Algoritmo Genético (AG). O primeiro projeto a ser usado foi o *Bricks* (Secção 2.5.2), visto que se tratava de um projeto propositadamente vulnerável e de dimensões reduzidas. Isto permitiu uma análise manual do projeto para validar de forma empírica as vulnerabilidades reais existentes. Esta secção demonstra, em pormenor, as vulnerabilidades encontradas, bem como as técnicas aplicadas.

Em fevereiro de 2021, tendo-se obtido resultados satisfatórios no projeto *Bricks*, estendeu-se a ferramenta a outras operações *Data Manipulation Language* (DML). Uma operação DML bem-sucedida leva à alteração do estado da base de dados, pelo que foi implementado um mecanismo que garantisse o estado original da base de dados em cada iteração. Surgiu, desta forma, a versão 1.1. Ainda em fevereiro de 2021, foi modelada uma segunda versão da possível solução desenvolvida, onde, a possível solução é construída de forma diferente.

Em março de 2021, o AG é aplicado a projetos de maiores dimensões. Nesta altura, surgem alguns problemas de desempenho, principalmente se um projeto tiver muitas *queries* do tipo *DELETE*, *INSERT* e *UPDATE*. A Secção 4.5 ilustra o problema e a solução implementada, surgindo desta forma a versão 1.3.

Em abril de 2021, para além da *interface* gráfica desenvolvida anteriormente, foi adicionada a possibilidade de interagir com a linha de comandos. Nesta altura, todos os parâmetros variáveis foram extraídos para ficheiros de configurações, de modo a que fosse possível executar o projeto em qualquer dispositivo. Surgindo, desta forma, os primeiros testes realizados num *Raspberry PI*.

Em maio de 2021, foi implementado o algoritmo *Artificial Bee Colony* (ABC) para a resolução do mesmo problema. Nesta fase, começaram a ser realizados alguns testes comparativos entre ambos os algoritmos. Foi ainda melhorada a obtenção de estatísticas, com a extração de parâmetros variáveis com base no algoritmo escolhido, bem como valores médios, desvio padrão, máximos e mínimos de variáveis como *fitness*, tempo de execução e quantidade de vulnerabilidades encontradas.

Em junho de 2021, foram efetuadas várias melhorias ao AG e à ferramenta, tendo sido recuperada a primeira versão do indivíduo desenvolvida (todos os algoritmos implementados trabalham com um conjunto de indivíduos, população, sendo que um indivíduo corresponde a uma possível solução para o problema a otimizar - um indivíduo na primeira versão correspondia a um possível vetor de ataque). Assim, é dada a possibilidade ao utilizador de escolher entre a versão um e dois para a representação do indivíduo. Estas versões são analisadas no Capítulo 4. Nesta

altura, várias iterações na função de *fitness* são efetuadas, para avaliar quais são os resultados que obtêm melhores valores. É nesta altura que se obtém 100% de eficácia na deteção de vulnerabilidades no projeto *Bricks* sem nenhum falso positivo, algo que não fora possível nos vários artigos estudados, tal como constatado na Secção 2.4.

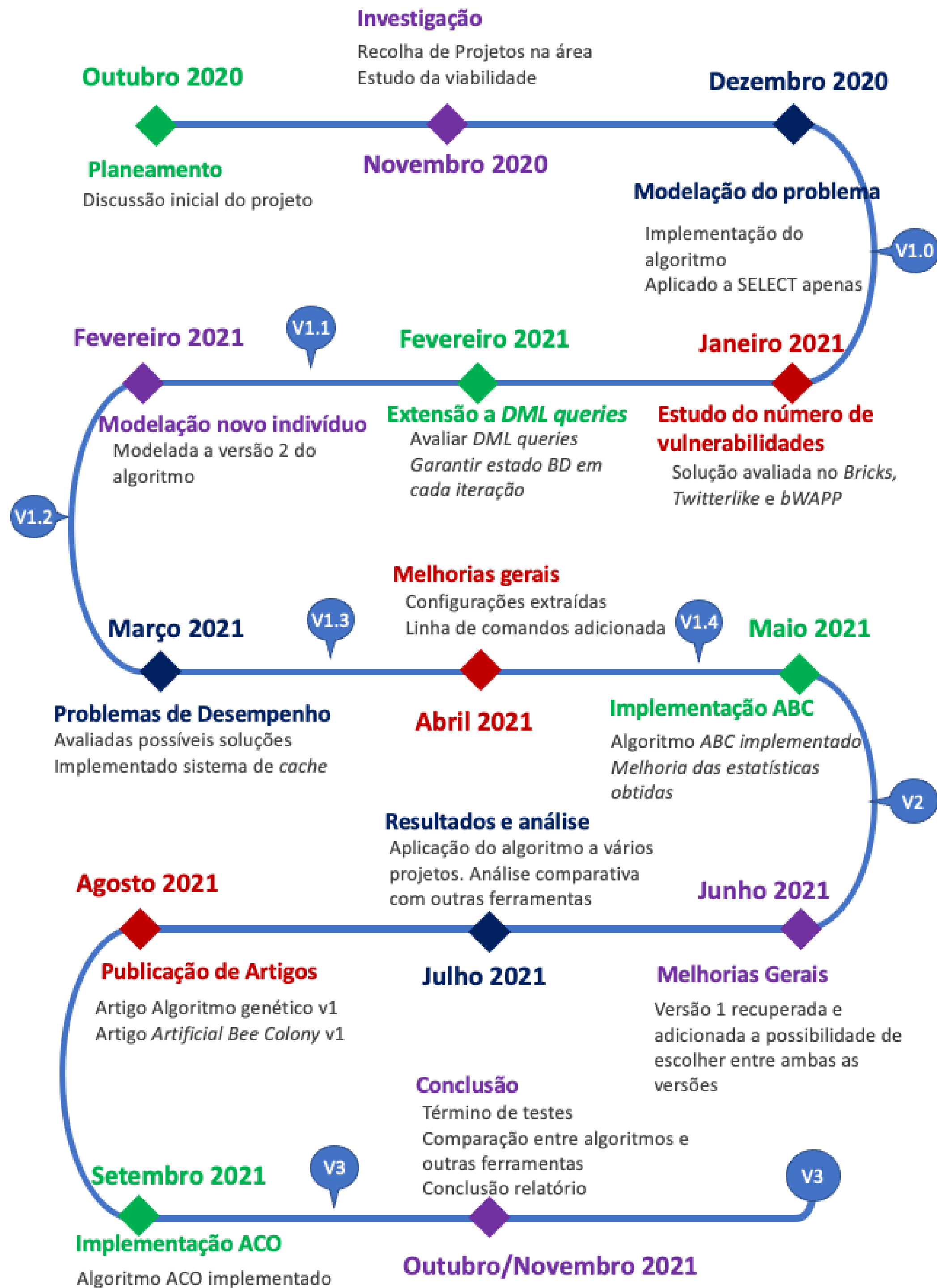

Figura 1: Cronograma do trabalho realizado

Em julho de 2021, são retirados os vários resultados comparativos dos algoritmos desenvolvidos para ambas as versões (representações do indivíduo) desenvolvidas. Nesta altura, foram adicionados novos parâmetros, como tempo máximo de execução, permitindo comparar todos os algoritmos para um tempo pré-definido de execução. A partir desta altura, o foco foi obter os vários resultados necessários, bem como a escrita do presente relatório.

Em agosto de 2021, foi aceite o artigo *Exploring SQL Injection Vulnerabilities Using Genetic Algorithms* na conferência *ICS 2021: 15. International Conference on Security*, em Lisboa. Ainda no mês de agosto, foi publicado outro artigo, *Exploring sql injection vulnerabilities using artificial bee colony* na *18th International Conference on Applied Computing*. Ambos os artigos focam-se na segunda versão desenvolvida e têm como base o AG e o ABC, respetivamente.

No mês de setembro de 2021, foi implementado um terceiro algoritmo, *Ant Colony Optimization* (ACO), bem como realizados os vários testes para encontrar os parâmetros ótimos.

No mês de outubro de 2021, foi aceite a publicação de um terceiro artigo baseado no AG, versão 1 do indivíduo, na conferência *ICITS22 - The 2022 International Conference on Information Technology & Systems*.

Por fim, no mês novembro de 2021, procedeu-se à conclusão da escrita deste documento, com o término das comparações entre os algoritmos estudados e desenvolvidos, bem como à submissão de um quarto artigo com a comparação entre os algoritmos ABC e ACO, que se encontra ainda em processo de revisão.

1.3 ESTRUTURA DO DOCUMENTO

O presente documento está estruturado em seis capítulos, sendo o primeiro capítulo reservado à introdução.

No Capítulo 2 é feito um enquadramento do projeto, com foco no estado da arte e conceitos de segurança informática e IA. São ainda introduzidos os projetos analisados, bem como ferramentas desenvolvidas por terceiros que serão utilizadas para comparação com a abordagem implementada.

No Capítulo 3 é apresentada a metodologia de desenvolvimento utilizada ao longo da realização deste projeto.

O Capítulo 4 é dedicado ao desenvolvimento do projeto. Este inclui a descrição do problema, bem como a proposta de modelação. De seguida, é descrita para

cada algoritmo de IA estudado, a sua aplicação ao problema inicial. No restante capítulo são apresentados pontos mais técnicos, tais como otimizações efetuadas, configurações existentes e tecnologias utilizadas.

No Capítulo 5 são apresentadas as experiências realizadas e os resultados obtidos. São ainda apresentadas comparações entre os algoritmos estudados e, posteriormente, com ferramentas desenvolvidas por terceiros.

Por fim, o Capítulo 6 finaliza o documento com um breve sumário das experiências e resultados. As contribuições deste projeto, na comunidade científica, a importância da aplicação de algoritmos IA na área da cibersegurança, as limitações atuais, bem como trabalho futuro são outros tópicos deste capítulo.

2

ENQUADRAMENTO

As organizações têm vindo, ao longo dos anos, a digitalizar os seus processos de negócio. Com esta nova era da informação, a segurança torna-se um pilar cada vez mais fundamental. Contudo, a segurança informática continua a ser um tópico abstrato, dado que é bastante difícil afirmar que um sistema está completamente seguro. Os sistemas atuais estão muitas vezes assentes em diferentes camadas de abstração, dificultando a identificação de vulnerabilidades.

Neste capítulo é efetuado um enquadramento a vulnerabilidades de segurança (Secção 2.1), bem como às que representam um maior risco na *web*. De seguida, são descritas algumas vulnerabilidades de SQL *Injection* (Secção 2.2), assim como, formas de colmatar estas vulnerabilidades.

Na Secção 2.3, são introduzidos os algoritmos de IA, dando foco aos AG, ABC e ACO. Uma vez introduzidas as áreas de IA e segurança, o foco da Secção 2.4 é descrever o estado da arte atual que interliga estas duas áreas disciplinares.

As Secções 2.5 e 2.6 contextualizam o cenário montado para o desenvolvimento e análise do projeto realizado.

2.1 VULNERABILIDADES DE SEGURANÇA

A OWASP[1] é uma organização internacional de fundos não lucrativos, dedicada à segurança em aplicações *web*. Um dos princípios fundamentais da OWASP é disponibilizar conteúdo gratuitamente e de fácil acesso *online*, para que qualquer pessoa possa melhorar a segurança das suas aplicações *web*. Segundo Mishra (2021), uma vulnerabilidade pode ser definida como:

> *A vulnerability is a hole or a weakness in the application, which can be a design flaw or an implementation bug, that allows an attacker to cause harm to the stakeholders of an application. Stakeholders include the application owner, application users, and other entities that rely on the application.*

1 https://owasp.org/

Com base nesta definição foi desenvolvido um projeto designado *OWASP Top 10*, cujo intuito é consciencializar as pessoas para riscos em aplicações *web* (*OWASP Top Ten* 2017). Este documento representa um consenso global, sobre os maiores riscos para aplicações *web* atualmente e, recomenda que todas as empresas incorporem este documento nos seus processos para minimizar ou mitigar riscos de segurança.

Nas próximas secções são resumidas as 10 vulnerabilidades com maior risco na *web* (*OWASP Top Ten* 2017).

2.1.1 *Injeção*

Em primeiro lugar, considerada a vulnerabilidade com maior risco (*OWASP Top Ten* 2017) encontra-se "ataques de injeção" que podem estar relacionados com SQL, Sistema Operativo (SO) e *Lightweight Directory Access Protocol* (LDAP). Estas vulnerabilidades encontram-se no topo da lista, visto que para além de serem ataques recorrentes, o impacto é devastador. Durante estes ataques é enviado texto malicioso ao sistema. Quando não existe uma sanitização correta dos valores recebidos, o atacante consegue executar comandos não autorizados no sistema. As consequências da exploração com sucesso de uma vulnerabilidade deste tipo são devastadoras, podendo resultar na alteração, remoção e visualização de dados sensíveis, em *Denial of Service* (DoS), ou em casos extremos, na aquisição completa da aplicação.

2.1.2 *Falhas de Autenticação*

As aplicações *web* usam sessões para gerir a autenticação dos utilizadores. Por vezes, incorretas configurações permitem a um atacante comprometer *passwords*, chaves ou *tokens* de sessão, conseguindo, desta forma, assumir a identidade de outros utilizadores de forma temporária ou permanente. Esta vulnerabilidade pode, portanto, expor dados sensíveis e executar ações não permitidas.

2.1.3 *Expor dados sensíveis*

Muitas aplicações *web* e *Application Programming Interface* (API) não protegem devidamente dados sensíveis. É importante definir-se uma política de controlos de acesso, isto é, definir quem e a quais os recursos podem aceder. Caso não seja bem

definida e, posteriormente implementada, os atacantes podem explorar estas falhas para obter dados sigilosos.

2.1.4 *XML External Entities*

Os atacantes podem explorar esta vulnerabilidade se conseguirem efetuar *upload* de um *Extensible Markup Language* (XML) ou incluir conteúdo malicioso num documento XML. Estas falhas podem ser exploradas para extrair dados, executar pedidos remotos ao servidor, analisar o sistema interno, ou executar um ataque DoS.

2.1.5 *Falha nos controlos de acesso*

As restrições dadas aos utilizadores, não são, muitas vezes, garantidas pelas aplicações. Os atacantes podem explorar estas falhas no controlo de acessos, para obter dados confidenciais.

2.1.6 *Incorretas configurações de segurança*

As aplicações *web* são complexas, pois assentam muitas vezes em várias camadas, desde *frameworks*, bibliotecas, servidores de base de dados, *email*, entre outras. Estas camadas devem ser configuradas e atualizadas para versões mais recentes. Caso isto não aconteça, o atacante pode aproveitar-se deste facto e explorar estas falhas.

2.1.7 *Cross-Site Scripting*

As vulnerabilidades *Cross-Site Scripting* (XSS) ocorrem quando uma aplicação inclui dados de origem não fidedigna numa página *web*, sem a devida validação ou sanitização. Estas falhas permitem ao atacante executar código nos navegadores dos utilizadores que usam a aplicação. As consequências podem ser: conseguir obter a sessão de um utilizador, redirecioná-lo para um *site* malicioso, entre outros.

2.1.8 *Desserialização insegura*

Esta vulnerabilidade, muitas vezes, permite ao atacante executar código remotamente, tipicamente designado como *Remote Code Execution* (RCE). Mesmo que por vezes não se consiga executar um RCE, estas falhas muitas vezes permitem outras categorias de ataques, nomeadamente ataques de injeção, ou escalamento de privilégios.

2.1.9 *Usar componentes com vulnerabilidades conhecidas*

Um componente pode ser definido como qualquer biblioteca, *framework*, ou outro *software* que seja necessário para a aplicação *web*. Muitas vezes determinadas versões ficam conhecidas por estarem vulneráveis a alguma categoria de ataque e é lançada uma nova versão para corrigir a situação. No entanto, é necessário estar atento a estas situações, pois se uma aplicação depender de um componente desatualizado e vulnerável a alguma categoria de ataque, este pode ser aproveitado para fins maliciosos. As consequências serão tão graves quanto a gravidade da vulnerabilidade em questão.

2.1.10 *Insuficiente monitorização*

A maioria dos ataques bem-sucedidos começa com uma fase de exploração. Permitir que estas investigações iniciais aconteçam, aumenta substancialmente a probabilidade de um ataque bem-sucedido no futuro. Por exemplo, um utilizador enganar-se nas suas credenciais aquando de uma autenticação, pode ser visto como algo normal. O mesmo utilizador enganar-se, por exemplo, 200 vezes por minuto, é muito provavelmente, trabalho de um ataque de força bruta executado por um *script*.

2.2 SQL INJECTION

A maioria das aplicações *web* possui formulários para o utilizador preencher e interagir, quer seja para criar uma conta, autenticar-se ou pesquisar. No que lhe concerne, o sistema irá receber os dados preenchidos e interpretá-los com a devida lógica de negócio envolvida. Independentemente do cenário, é expectável que existam dados e, que estes sejam guardados numa base de dados, quer seja SQL ou *Non*

Structured Query Language (NoSQL). Haverá, portanto, um acesso a uma base de dados, para recolher os dados necessários para a operação a ser realizada. Esta operação, caso o programador não tenha alguns cuidados pode ser explorada para obter dados não intencionados.

Tal como vimos na Secção 2.1.1, vulnerabilidades da categoria injeção foram, em 2017, consideradas pela OWASP, como as vulnerabilidades de maior risco para aplicações *web*. Portanto, é importante conhecer esta vulnerabilidade com mais detalhe. As próximas secções detalham o funcionamento de SQL *Injection* (Secções 2.2.1 e 2.2.2), bem como mecanismos de prevenção (Secção 2.2.3).

2.2.1 *Exemplo de um ataque simples*

Como um possível ataque, consideremos um processo de autenticação no sistema. Para simplificar o processo, podemos considerar que a *password* é guardada em *Plain Text*, isto é, guardada exatamente como o utilizador a escreveu, sem qualquer algoritmo de *hashing*.

Nesta situação, o utilizador introduz o seu *username* e *password* num formulário. O sistema por sua vez, verifica se existe um *username* com determinada *password*. A Listagem 1 representa um exemplo de uma *query* que poderia ser encontrada no código-fonte de uma aplicação em PHP.

```
1   SELECT *
2   FROM users
3   WHERE name='$username' and password='$pwd';
```

Listagem 1: Exemplo de uma *query* para autenticar um utilizador

Neste caso, as variáveis *$username* e *$pwd* serão substituídas pelos valores dados pelo utilizador. Consideremos que um utilizador bem-intencionado, introduz *Maria* e *123* como *username* e *password*, respetivamente. A *query* executada pelo sistema será:

```
1   SELECT *
2   FROM users
3   WHERE name='Maria' and password='123';
```

Listagem 2: Exemplo de uma *query* para autenticar um utilizador com valores introduzidos

Se o utilizador existir, o sistema irá retornar o mesmo, e o sistema valida que de facto existe um utilizador com a *password* introduzida. Agora, consideremos,

um utilizador mal-intencionado, que introduz como *$username*, o valor `'OR 1=1 --` Desta forma a *query* executada pode ser observada na Listagem seguinte:

```
1    SELECT *
2    FROM users
3    WHERE name='' OR 1=1 -- ' and password='';
```

Listagem 3: Exemplo de uma *query* com parâmetros maliciosos injetados

Sendo `--` um comentário em alguns Sistema de Gestão de Base de Dados (SGBD), tudo o que ficar depois deste é ignorado. Nesta situação, esta *query*, que inicialmente deveria apenas devolver o utilizador, caso exista, devolve todos os utilizadores da base de dados, isto porque a segunda condição será sempre verdadeira (1=1). Portanto, o valor de entrada da *password* perde qualquer relevância, pois é ignorado.

2.2.2 *Obter o esquema da base dados*

Anteriormente, vimos como ultrapassar um mecanismo de autenticação que seja vulnerável a SQL *Injection*. Nesta secção, pretende-se demonstrar como alterar a *query* de modo a deduzir o mapeamento dos campos de uma tabela na base de dados. Na situação anterior, apesar da autenticação realizada com sucesso, apenas se consegue obter aquilo que se encontra na tabela dos utilizadores.

A primeira etapa neste processo é tentar adivinhar alguns campos que possam ser mais óbvios existirem numa tabela. Tendo como exemplo a tabela de utilizadores, é expectável, que possa ter campos como morada, *email*, número de telefone, entre outros. O ideal seria executar a *query* `SHOW TABLE <nome_tabela>`, no entanto, neste ponto ainda não se sabe o nome da tabela. Consideremos a *query* apresentada na Listagem 4 e o vetor de ataque apresentado na Listagem 5.

```
1    SELECT name, description, price
2    FROM products
3    WHERE category = ?
```

Listagem 4: *Query* base para obter o nome da tabela

```
1    1 AND 1=2 UNION SELECT table_schema, table_name, 1 FROM
     ↪  information_schema.table
```

Listagem 5: Exemplo de um vetor de ataque

O resultado deste *output* pode ser algo muito semelhante àquilo que podemos observar na Tabela 1. Tal como podemos observar, é possível obter dados do esquema e tabela a que pertencem.

Tabela 1: Resultados da execução de uma *query* para obter o esquema da Base de Dados

NAME	DESCRIPTION	PRICE
twitterlike	follow	1
twitterlike	tweet	1
twitterlike	user	1

Regra geral, conseguir um ataque de SQL *Injection* com sucesso requer várias tentativas e, o uso de várias técnicas, para além daquelas demonstradas anteriormente. É necessário ter em conta que o atacante, à partida, não tem acesso ao código-fonte e, portanto, não sabe exatamente como a *query* está construída. No cenário anterior, convenientemente o seletor tinha 3 parâmetros, sendo que para o ataque demonstrado, dois deles eram bastante úteis, pois tratavam-se de duas colunas que eram *strings*. Como não precisávamos da terceira, colocou-se o valor 1. Chegar a este valor, envolve efetuar muitas tentativas, quando não se tem acesso ao servidor. Todavia, o servidor pode-nos dar pistas relativamente ao erro durante a construção de uma *query* válida. Com base nesta resposta, é possível fazer ligeiras alterações nos parâmetros e, com alguma dedução, obter uma *query* válida.

2.2.3 *Prevenir ataques de SQL Injection*

Vulnerabilidades do tipo SQL *Injection* são introduzidas quando os programadores criam *queries* dinâmicas que dependem de dados vindos de fontes não fidedignas. Desta forma, para evitar estas falhas, segundo Mishra (2021), existem duas hipóteses: (1) não usar *queries* dinâmicas ou (2) prevenir o utilizador de providenciar *inputs* que possam ser maliciosos. A primeira solução não é comum no desenvolvimento de *software*, pois dificilmente existirão aplicações que não interagem com o utilizador. Tendo em foco a segunda hipótese, serão indicadas algumas defesas primárias, agnósticas à linguagem de programação ou *framework* utilizada.

Uma opção para mitigar este problema é o uso de *Prepared Statements* (Mishra, 2021). Isto garante que o atacante não é capaz de alterar a intenção de uma *query*, mesmo que este envie um *input* malicioso, pois os parâmetros serão tratados literalmente e não executados como código.

Outra hipótese, é sanitizar todos os *inputs* providenciados por fontes não fidedignas. No entanto, segundo a OWASP, esta técnica apenas deve ser usada como último recurso. A ideia é tratar os dados introduzidos pelos utilizadores antes de executar uma *query*. O problema desta abordagem é não garantir a prevenção de todos os ataques de SQL *Injection* em todas as situações. Por este motivo, a OWASP aconselha o uso de *Prepared Statements*, Procedimentos ou mesmo um *Object Relational Mapper* (ORM) que construa as *queries*.

2.3 ALGORITMOS DE INTELIGÊNCIA ARTIFICIAL

A IA é uma área com diversas aplicabilidades. Vários algoritmos têm sido utilizados para resolver diversos problemas, desde problemas bem definidos como jogos, até problemas de classificação, como identificação de *spam* em *emails*. Nos últimos anos tem-se tentado integrar IA e *Machine Learning* (ML) tanto numa vertente mais defensiva de cibersegurança, como ofensiva (Mckinnel et al., 2019).

Nas próximas secções são introduzidos os vários algoritmos de IA utilizados para a resolução do problema apresentado anteriormente (AG na Secção 2.3.1, ABC na Secção 2.3.2 e ACO na Secção 2.3.3). O principal objetivo deste trabalho é usar estes algoritmos para detetar vulnerabilidades do tipo SQL *Injection* em código-fonte.

2.3.1 *Algoritmos Genéticos*

Os AG são um dos primeiros algoritmos estocásticos propostos baseados em populações de indivíduos (Mirjalili, 2018). Estes foram inspirados na teoria da evolução de *Darwin* (D. Goldberg, 1988), segundo a qual, existe uma luta constante pela sobrevivência. Contudo, aquele que sobrevive não é necessariamente o mais forte, mas aquele que melhor se adapta às condições do ambiente em que vive.

Segundo Holland (1992), este algoritmo foi desenvolvido para simular alguns dos processos observáveis na evolução da natureza. Este algoritmo difere de outros métodos de procura, no sentido em que a procura acontece dentro de uma população medida por pontos e com um conjunto de parâmetros.

Uma solução (indivíduo) corresponde a um cromossoma e cada posição do indivíduo representa um gene. A aptidão de um indivíduo na população é avaliada através de uma função de *fitness*. Para melhorar as soluções, vários mecanismos podem ser aplicados. Segundo Mirjalili (2018), quando uma solução é muita fraca, a melhor

solução é usar métodos de seleção aleatórios, tal como a *Roulette Wheel*. É provável obter melhores resultados com este mecanismo, pois a probabilidade é proporcional ao valor de *fitness*. Holland (1992) ainda sugere que este mecanismo ajuda a que as soluções não fiquem presas em máximos/mínimos locais. Ou seja, a aleatoriedade, tal como na natureza, é muito importante, para introduzir variabilidade que permita a uma espécie se adaptar às mudanças do ambiente.

Segundo Gen e Cheng (2007), apesar de o AG ser estocástico, este pode ser considerado fidedigno e capaz de encontrar uma solução ótima para um dado problema, devido à sua capacidade de manter as melhores soluções em cada geração e, usando-as para melhorar outras soluções. A recombinação entre indivíduos resulta na exploração da zona entre duas soluções. O algoritmo também beneficia da mutação que adiciona aleatoriedade na solução, alterando os genes de um indivíduo. Desta forma, a diversidade do indivíduo é incrementada na população, abrindo novas zonas de procura para o algoritmo explorar.

Para aplicar os AG a um problema é necessário seguir determinadas etapas. A Listagem 6 mostra resumidamente as etapas necessárias durante a execução do AG. Vários operadores genéticos são aplicados durante a execução do algoritmo e dentro de cada um, existem variações. Nas próximas secções este processo é descrito em pormenor.

```
1   Inicializar Parâmetros
2   Criar População Inicial
3   Avaliar População
4   ENQUANTO condição de término não atingida:
5       Criar Nova População
6           Selecionar indivíduos com base num método de seleção
7
8           Aplicar Operadores Genéticos
9               Recombinação
10              Mutação
11
12      Calcular fitness e ordenar a população
13  Fim do ciclo
```

Listagem 6: Pseudocódigo das etapas do algoritmo genético

2.3.1.1 *População inicial*

O algoritmo começa com a criação de uma população de forma aleatória. O principal objetivo nesta etapa é conseguir uma população inicial distribuída uniformemente no espaço de procura. Quanto melhor a distribuição inicial, maior a probabilidade de encontrar áreas promissoras (Mirjalili, 2018).

2.3.1.2 *Métodos de Seleção*

O método de seleção, ou designado por vários autores como reprodução, é um dos operadores com mais inspiração na teoria de *Darwin* (Goldberg e Deb, 1991), na medida em que na natureza, os indivíduos mais aptos a determinado ambiente têm mais hipóteses de obter comida e reproduzir-se e, portanto, os seus genes têm maior probabilidade de contribuírem para gerações futuras.

Inspirado por esta ideia, este operador atribui uma probabilidade de um indivíduo ser escolhido para a próxima geração. Para atribuir esta probabilidade, diferentes métodos de seleção podem ser utilizados, desde *Boltzmann Selection*, *Rank Selection*, *Steady State Selection*, *Truncation Selection*, *Roulette Wheel* ou *Tournament Selection* (Torneio) (Goldberg e Deb, 1991). Durante a realização deste projeto foram implementados e analisados os métodos de seleção: *Roulette Wheel*, Torneio e Elitismo.

No método de seleção *Roulette Wheel*, a probabilidade de escolher um indivíduo para a geração seguinte é proporcional ao seu *fitness* (Bernardino et al., 2008). A Figura 2 representa uma população com cinco indivíduos. Os dados foram construídos com base na Tabela 2. Ao ser utilizado este método de seleção, o indivíduo 1 é o indivíduo com maior probabilidade de passar às gerações seguintes, no entanto, apesar de o indivíduo 4 ser o que tem menor probabilidade, existe, contudo, a possibilidade de que este indivíduo seja selecionado. Isto pode ser visto como um benefício, pois ao contrário de descartar por completo as piores soluções, este mecanismo ajuda a manter a diversidade na população. Tal como referido anteriormente, a diversidade na população ajuda a que uma população não fique presa numa área de procura.

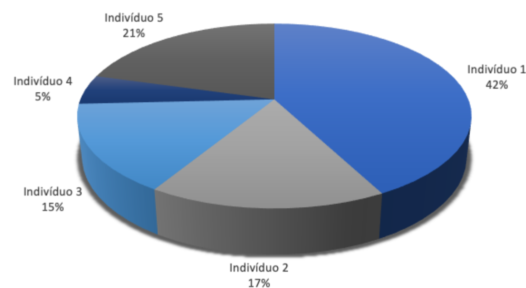

Figura 2: Mecanismo de distribuição de seleção usando *Roulette Wheel*

Tabela 2: Valores *fitness* e percentagem total de uma população

INDIVÍDUO	VALOR *FITNESS*	VALOR EM PERCENTAGEM
1	8.2	42%
2	3.2	17%
3	3	15%
4	1	5%
5	4	21%

Os métodos de seleção por Torneio providenciam uma seleção, tal como o nome indica, criando torneios de tamanho variável de indivíduos para competirem entre si. O vencedor de um torneio é o indivíduo com maior *fitness*. Indivíduos com menor *fitness*, poderão eventualmente ganhar, se dentro do seu torneio forem os que têm maior *fitness*. No método de seleção por Torneio, a ideia é executar vários torneios entre os indivíduos. Em cada torneio, são selecionados N indivíduos (sendo N o tamanho do torneio). No fim do torneio, é devolvido o melhor indivíduo encontrado (Nadarajah, 2018).

No caso de o torneio ter um tamanho baixo, este método de seleção dá também a hipótese de qualquer indivíduo ser escolhido, preservando desta forma a diversidade. No entanto, ao contrário do método de seleção *Roulette Wheel*, se o tamanho do torneio for grande, os indivíduos com menor *fitness* muito dificilmente serão escolhidos para as gerações seguintes (Nadarajah, 2008). Por exemplo, para o pior indivíduo ser escolhido num torneio com tamanho 2, é necessário que o mesmo indivíduo seja escolhido duas vezes. À medida que incrementamos o tamanho do torneio, a probabilidade de isto acontecer diminui drasticamente.

A Tabela 3 mostra um exemplo de competições entre indivíduos por torneio de tamanho 2.

Tabela 3: Exemplo de uma seleção por torneio de tamanho 2

TENTATIVA	INDIVÍDUOS SELECIONADOS	VENCEDOR
1	1 vs 2	Indivíduo 1
2	1 vs 3	Indivíduo 1
3	2 vs 4	Indivíduo 2
4	3 vs 5	Indivíduo 5

Para medirmos a eficácia destes métodos de seleção, para além de compararmos os resultados obtidos entre ambos, é efetuado um estudo exploratório para perceber

que valores de probabilidade geram melhores resultados. No caso do método de seleção por Torneio, ainda é conduzido um estudo exploratório para perceber qual o tamanho de torneio que obtém melhor *fitness*.

Para além dos métodos de seleção apresentados anteriormente, foi ainda utilizado um terceiro, designado de Elitismo. O Elitismo garante que alguns dos melhores indivíduos passem para as próximas gerações (Ahn e Ramakrishna, 2003). A ideia deste conceito é evitar que soluções de elite (com *fitness* ótimo) sejam prejudicadas quando aplicados outros operadores. O Elitismo pode ser aplicado em percentagem ou número absoluto. Por exemplo, podemos garantir que 10% de uma população não seja prejudicada, como podemos garantir que apenas 5 dos melhores indivíduos passem à geração seguinte.

O Elitismo nem sempre garante melhores resultados, pois garantir que determinados elementos da população não sejam afetados, pode fazer com que a curva de aprendizagem da população seja muito prejudicada e, que a variabilidade genética não seja o suficiente, estagnando a evolução.

2.3.1.3 *Métodos de Recombinação*

Após a seleção dos indivíduos para as próximas gerações, é necessário que alguns desses indivíduos se cruzem para gerar novos. A ideia é combinar os genes de dois indivíduos para produzir um terceiro. Tal como existem vários métodos de seleção, também existem várias formas de recombinação. Na literatura podem-se encontrar vários métodos de recombinação possíveis, que dependem da representação dos indivíduos: uniforme, *Three Parents Crossover*, *Cycle Crossover*, *Heuristic Crossover* (Hu e Paolo, 2009; Raghuwanshi e Kakde, 2006). No entanto, vamos focar-nos nos métodos de recombinação de um corte, dois cortes e uniforme.

Tal como podemos observar pela Figura 3, na recombinação com um corte, existe apenas um ponto de corte a partir do qual o gene dos progenitores é trocado um com outro.

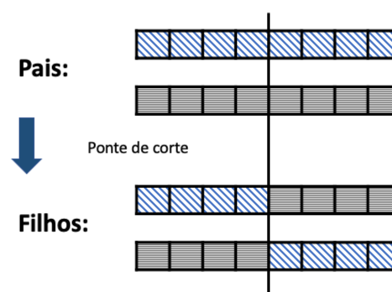

Figura 3: Método de recombinação com um corte

O método de recombinação de dois cortes é muito semelhante ao método anterior. A diferença, tal como se pode observar pela Figura 4, é a existência de dois pontos de corte. Existem ainda recombinações com múltiplos cortes (Shang e Li, 1991).

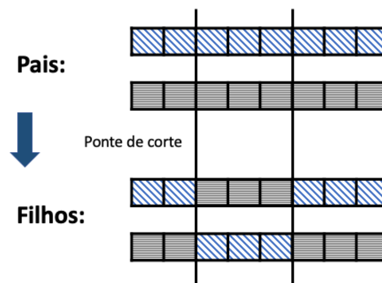

Figura 4: Método de recombinação com dois cortes

O método de recombinação uniforme é conhecido como o operador mais eficaz em AG convencionais (Spears e Jong, 1995). Tal como podemos constatar pela Figura 5, o algoritmo utiliza uma máscara auxiliar (Semenkin e Semenkina, 2012). Esta máscara é construída aleatoriamente com 1 e 0 (ou verdadeiro e falso). Com base no valor da máscara haverá ou não uma troca de genes entre os indivíduos pais. No caso da Figura 5, a máscara apresenta 1 como primeiro gene, pelo que haverá troca entre os genes dos pais. No terceiro gene da máscara foi gerado o valor 0, pelo que não haverá troca de genes.

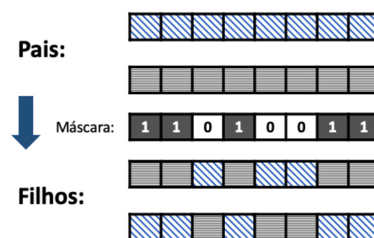

Figura 5: Método de recombinação uniforme

2.3.1.4 *Mutação*

A mutação é o último operador a ser executado, tipicamente após a seleção e a recombinação. O objetivo deste operador é introduzir variabilidade genética ao nível do gene, ou seja, dada uma determinada probabilidade é calculado se um gene deve ou não ser alterado. Valores muito altos de mutação não são recomendados, pois, induzem a uma pesquisa aleatória (Michalewicz, 2018).

A Figura 6 mostra um exemplo de mutação que poderia ser aplicada aos cromossomas filhos. Basicamente, este processo pode ser visto como pequenas mudanças aleatórias em alguns genes, de modo a introduzir mutação num indivíduo. Esta técnica pode ajudar uma população a sair de *maus* genes dos progenitores.

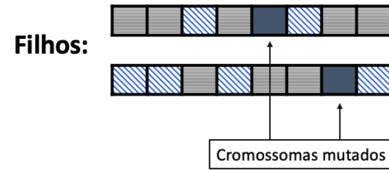

Figura 6: Exemplo de mutações

2.3.1.5 *Função de Fitness*

Como mencionado anteriormente, os AG imitam o comportamento na natureza e têm como base a lei de evolução de *Darwin*, em que o indivíduo mais apto a um dado ambiente, vive mais tempo, reproduz-se mais e, portanto, os genes que o favorecem no seu ambiente, tendem a surgir mais vezes na população. Trata-se da sobrevivência do mais apto, portanto, naturalmente, os AG são recomendados para resolver problemas de otimização (Nakama, 2008).

O valor de *fitness* é descrito como a qualidade que determinado indivíduo tem no seu ambiente, sendo, portanto, o medidor de qualidade de um indivíduo. Num problema de maximização, o objetivo é maximizar o valor de *fitness*. A definição de uma função de *fitness* adequada ao problema é fulcral para o sucesso da aplicação destes algoritmos. Tal como veremos no Capítulo 3, o objetivo é maximizar o valor de *fitness*. Um maior valor será indicativo de mais vulnerabilidades, mas tal como veremos, existem outros parâmetros importantes a avaliar.

Na Secção 4.2 será apresentado o AG aplicado ao problema desenvolvido e, como cada uma das etapas foi aplicada ao problema apresentado.

2.3.2 *Artificial Bee Colony*

O ABC é um algoritmo de *Swarm Intelligence* proposto por Karaboga (2005), que se baseia no comportamento de colónias de abelhas e foi aplicado com sucesso em vários problemas de otimização numérica (Akay e Karaboga, 2010; Akay e Karaboga, 2020).

Uma colónia de abelhas procura por alimentos com mais nutrientes para maximizar a quantidade de néctar produzido na colmeia. Embora existam vários processos para classificar o trabalho de cada abelha dentro de uma colmeia, o algoritmo utiliza um modelo simplificado composto por abelhas empregadas (*employed*), espetadoras (*onlooker*) e exploradoras (*scouts*).

Uma abelha empregada tem como finalidade explorar fontes de alimentação na sua vizinhança e transmitir essa informação à colmeia. Tem, portanto, como principal objetivo explorar o espaço de solução conhecido.

A abelha espetadora fica na colmeia e tem como principal objetivo decidir qual a fonte de alimento a explorar. A sua decisão é baseada na informação que as abelhas empregadas partilham. Nesta fase, a abelha espetadora escolhe a melhor fonte com base na qualidade da solução, ou seja, com base no *fitness*.

Por fim, a abelha exploradora, sai do espaço de solução conhecido para explorar alimentos em novas zonas. Esta exploração é aleatória e, visa introduzir variabilidade na solução e, evitar que a qualidade da solução fique comprometida pela zona conhecida, evitando máximos locais.

Segundo Akay e Karaboga (2020), este processo consiste em quatro etapas essenciais:

1. Abelhas empregadas exploram a vizinhança e partilham com abelhas espetadoras a qualidade da solução;
2. Abelhas espetadoras com base na informação recebida avaliam a melhor solução;
3. Abelhas exploradoras introduzem variabilidade na solução, visto que exploram espaços de solução aleatórios.

A Listagem 7 resume as etapas principais do algoritmo ABC aplicadas neste projeto. Neste pseudocódigo, é possível distinguir vivamente as três etapas do algoritmo, bem como funciona cada etapa.

À semelhança do AG, este algoritmo também trabalha com uma população de indivíduos. Cada abelha corresponde a um indivíduo na população. Para avaliar a qualidade de cada abelha na população, é também usada uma função de *fitness* que determina a qualidade da potencial solução.

Na Secção 4.3, cada etapa associada ao algoritmo ABC é descrita no contexto do problema de SQL *Injection*. Desta forma, é possível perceber como cada etapa foi desenvolvida, bem como todas as equações utilizadas.

```
Inicializar parâmetros
Inicializar abelhas empregadas
Avaliar População

ENQUANTO condição de término não atingida:
    Fase Abelhas empregadas
        PARA CADA abelha NA colónia de abelhas empregadas
            avaliar abelha
        Calcular probabilidades

    Fase Abelhas espetadoras
        PARA CADA abelha_empregada NA colónia de abelhas empregadas
            calcular número de abelhas espetadoras
            PARA CADA abelha_espetadora NA colónia de abelhas espetadoras
                Aplicar busca local
                Avaliar abelha_espetadora
                SE fitness_abelha_espetadora < fitness_abelha_empregada
                    abelha_espetadora = abelha_empregada

    Fase Abelhas exploradoras
        Inicializar abelhas exploradoras
        Substituir piores abelhas empregadas por abelhas exploradoras


Fim do ciclo
```

Listagem 7: Pseudocódigo das etapas do *Artificial Bee Colony*

2.3.3 *Ant Colony Optimization*

Na natureza, as formigas comunicam indiretamente, entre si, depositando feromonas no chão, influenciando, desta forma, as decisões de outras formigas. Este algoritmo de otimização explora este comportamento das formigas. Desde o início dos anos 90, quando o primeiro algoritmo ACO foi proposto, por Dorigo et al. (1996), o algoritmo atraiu a atenção de vários investigadores.

Segundo Bernardino et al. (2008), o comportamento real das formigas transforma-se num algoritmo que estabelece um mapeamento entre: (1) a busca da formiga e o conjunto de soluções viáveis; (2) a quantidade de alimentos numa fonte e a sua relação com a função de *fitness*; e (3) rastro de feromonas e capacidade de memória.

À semelhança do AG e do ABC, este algoritmo também trabalha com uma população de indivíduos. Cada formiga corresponde a um indivíduo na população.

Para avaliar a qualidade de cada formiga na população é também usada uma função de *fitness* que determina a qualidade da potencial solução.

A Listagem 8 resume as etapas principais do algoritmo ACO aplicadas neste projeto. Mais informação relativamente à aplicação do algoritmo ACO ao problema de SQL *Injection* é apresentada na Secção 4.4. Nesta secção, é apresentado um extrato do código desenvolvido, para complementar a descrição.

```
1  Inicializar parâmetros
2  Inicializar colónia de formigas
3  Avaliar colónia
4
5  Inicializar rastro de feromonas
6  ENQUANTO condição de término não atingida:
7      PARA CADA formiga na colónia
8          Modificar a formiga
9          Avaliar formiga
10         Aplicar mecanismos de intensificação
11     Atualizar rastro de feromonas
12     Aplicar Mecanismos de Diversificação
13
14
15 Fim do ciclo
```

Listagem 8: Pseudocódigo das etapas do *Ant Colony Optimization*

2.4 ESTADO DA ARTE

As aplicações *web* ganharam popularidade no final do século XX e, nos últimos anos várias investigações foram efetuadas com o intuito de detetar e prevenir vulnerabilidades nestas aplicações. Vários projetos surgiram, com o intuito de ajudar entidades a proteger, testar, identificar e corrigir estas vulnerabilidades.

O trabalho manual na deteção de vulnerabilidades é fulcral, no entanto, tem vários problemas, desde o próprio erro humano, o custo, o tempo e até à própria dimensão da tecnologia. Tal como vimos anteriormente, uma aplicação *web* assenta em várias camadas. O modelo *Open System Interconnection* (OSI) surgiu como intuito de providenciar uma clara distinção entre camadas: física, enlace, rede, transporte, sessão, de apresentação e aplicacional. Mesmo se nos focarmos apenas na camada aplicacional, existem várias linguagens, várias *frameworks*, várias versões e várias bibliotecas, cada uma com as suas diferenças. Para além de termos muitas ferramentas diferentes, cada uma individualmente relaciona-se com a outra num ecossistema muito complexo. Uma pequena falha neste ecossistema abre a porta a várias vulnerabilidades que podem ser exploradas. Estudos como os de Huizinga e

Kolawa (2007) e Stefinko et al. (2016), sugerem formas automatizadas para explorar estas vulnerabilidades.

Num esforço para reduzir o custo das aplicações e aumentar a qualidade das aplicações, várias técnicas foram utilizadas para automatizar estes processos. Huizinga e Kolawa (2007), para automatizar este processo, usaram um modelo baseado em máquinas de estado. Estes modelos são ideais quando o problema tem várias sequências de eventos. Esta abordagem revelou ser bastante bem-sucedida, no entanto, tem como desvantagem o seu custo de execução, visto que durante a geração dos modelos é necessária a intervenção humana. O autor sugere que para melhorar o benefício custo-desempenho é necessário reduzir o número de possibilidades. O problema das máquinas de estado finito é a escalabilidade. As aplicações *web* modernas lidam com um grande número de transações, várias camadas de abstrações e muitas linhas de código, o que revelou ser bastante problemático usando esta abordagem.

O estudo de Stefinko et al. (2016) mostra os benefícios e desvantagens de testes automatizados e manuais. Segundo o autor, testes de penetração manuais, embora eficazes, dificilmente conseguem ir ao encontro de todos os requisitos de segurança que estão em constante mudança e evolução. Além disso, requerem conhecimento especializado que apresenta um custo elevado e é tipicamente mais lento. A alternativa são ferramentas automatizadas, que embora mais rápidas, muitas vezes não se adaptam ao contexto e unicidade de cada aplicação.

Nos últimos anos tem-se tentado integrar IA e ML, tanto numa vertente mais defensiva de cibersegurança, como ofensiva. Mckinnel et al. (2019), conduziu um estudo comparativo de vários algoritmos de IA na exploração de vulnerabilidades. No seu estudo, os autores compilaram vários trabalhos na área, para comparar vários algoritmos, desde algoritmos não supervisionados, *Reinforcement Learning*, AG, entre outros. Os autores concluem que o AG tem um melhor desempenho temporalmente devido à natureza da evolução das gerações. Sugerem que a sua aplicabilidade precisa ter em conta uma boa definição de *fitness* para obter melhores resultados. Como trabalho futuro, recomendam soluções desenhadas para sistemas específicos, em vez de soluções genéricas, pois, estas soluções mais genéricas apenas exploram vulnerabilidades mais superficiais e simples.

Em Niculae (2018), o autor desenvolveu uma estratégia de aprendizagem reforçada capaz de comprometer um sistema mais rapidamente do que uma abordagem de força bruta e aleatória. Este concluiu ser possível construir um agente capaz de aprender e evoluir temporalmente, para que consiga penetrar uma rede. A sua eficácia ficou igualável ao da capacidade humana. Por fim, concluiu também, que

embora o objetivo inicial tenha sido cumprido, existem ainda várias direções a serem exploradas. Uma delas seria a utilização de diferentes algoritmos para uma perspetiva tanto de segurança ofensiva (*Red Team*), como defensiva (*Blue Team*). Sugere também, a aplicação de conceitos de teoria de jogos (Casey et al., 2014), especialmente para tratar o problema como um *Stackelber Security Game*. Estas técnicas foram aplicadas com sucesso em vários domínios de segurança. Permite, por exemplo, encontrar a alocação ótima para segurança de um aeroporto, dado o conhecimento dos atacantes.

Alenezi e Javed (2016) analisaram vários projetos de código aberto para identificar vulnerabilidades. Concluíram que grande parte destes erros se devem a negligência por parte das pessoas que desenvolveram as aplicações, assim como o uso de más práticas. Para colmatar este problema, os autores sugerem o desenvolvimento de uma *framework* que incentive programadores a seguir as boas práticas e detetar possíveis falhas no código.

Em Liu et al. (2019), os autores propõem uma solução para detetar vulnerabilidade XSS, baseado no uso de AG, assim como uma proposta para remover as vulnerabilidades encontradas durante a fase de deteção. O objetivo era, portanto, encontrar o máximo de vulnerabilidades com o mínimo número de testes possível. O resultado obtido com o AG foi comparado com outras ferramentas de análise estática. Como trabalho futuro, sugerem a aplicação desta técnica em aplicações de outra área tecnológica, assim como, aos três tipos de XSS: persistente, refletido e baseado em *Document Object Model* (DOM).

Em Elderman et al. (2017), o foco está na segurança de ficheiros e dados numa rede de computadores. Após modelarem o problema, o treino ocorre com agentes a atacar e a defender-se entre si. O jogo é simulado com informação incompleta por parte de cada agente e, é treinado a usar técnicas de aprendizagem reforçada. Neste estudo concluíram que algoritmos como *Q-Learning* usando o algoritmo *epsilon-greedy* para exploração, mostram ser mais efetivos durante a defesa, enquanto os algoritmos baseados em aprendizagem *Monte Carlo* foram mais efetivos durante ataques.

Medeiros et al. (2014), propuseram uma técnica para deteção de vulnerabilidades em aplicações *web*, com o mínimo número de falsos positivos como objetivo principal, usando a combinação de dois métodos: *taint analysis* e *data mining*. A *taint analysis* é um processo usado para identificar fluxos de execução, no código-fonte. A análise consiste na construção de possíveis caminhos, com grafos, da execução de código. O objetivo é observar o caminho percorrido por variáveis que provêm de *inputs*

de utilizadores. Portanto, esta análise providencia potenciais vulnerabilidades e o processo de *data mining*, prevê a existência de falsos positivos nas vulnerabilidades identificadas.

Em Kumar e Chatterjee (2014) é modelado um cenário para deteção de SQL *Injection*, usando o conceito de teoria da informação. A proposta baseia-se na ideia de que o *input* de uma *query* maliciosa, altera a complexidade da *query*. Portanto, medindo esta complexidade estaticamente, conseguem observar em tempo real, qualquer desvio da ocorrência de um ataque de SQL *Injection*. A ferramenta desenvolvida funciona tanto do lado do cliente, como do lado do servidor. A ideia é implementar um algoritmo do lado do cliente para verificar a categoria de dados e o tamanho dos dados submetidos ao servidor. No servidor, o modelo funciona em duas fases: treino e deteção. A complexidade da *query* é calculada durante a fase de treino. Durante a fase de deteção é calculada a entropia da *query* gerada. Após isso, é aplicado um *Message Authentication Algorithm* (MAC) a ambos. Se a *query* foi alterada, isso implica que a estrutura original foi alterada e, portanto, a entropia também, o que irá alterar o correspondente MAC. Estas alterações são indicativas de SQL *Injection*. Uma das melhorias mencionadas é aplicar esta abordagem a procedimentos da base de dados, o que envolve ser estendido para *scripts* de *stored procedures*.

2.5 ESTUDO DE PROBLEMAS

De forma a avaliar a qualidade das soluções, foram estudadas várias aplicações *web* desenvolvidas em PHP com o intuito de verificar manualmente a quantidade de vulnerabilidades existentes. Na Secção 2.5.1 é descrito como as aplicações foram instaladas numa máquina local. Na Secção 2.5.2 é descrito o projeto *Bricks*. A Secção 2.5.3 é dedicada ao projeto *bWAPP* e, por fim, na Secção 2.5.4 é introduzido o *Twitterlike*. Posteriormente, é descrito e analisado o procedimento efetuado, assim como as vulnerabilidades identificadas e o modo de executá-las.

2.5.1 *Instalação local*

Para analisar as aplicações estudadas, foi necessário instalá-las numa máquina local. O *Bricks* e o *Twitterlike* não apresentam versões *online* e, embora o *Buggy Web Application* (bWAPP)[2] apresente, esta revela ser muito instável.

Docker[3] é uma plataforma *open source* para construir e gerir aplicações em contentores (*containers*). Ao contrário de máquinas virtuais que emulam um sistema operativo por completo, com *Docker* os processos são virtualizados e isolados sem esta necessidade. Desta forma, obtemos todos os benefícios e funcionalidades de uma máquina virtual, incluindo isolamento, escalabilidade e volatilidade. Além disso, são em termos computacionais, muito mais leves que as máquinas virtuais, visto que não necessitam da sobrecarga de correr um sistema operativo por completo e, permitem correr várias cópias de uma aplicação em paralelo.

Esta tecnologia foi aplicada não só para a preparação das várias aplicações estudadas, mas também para a ferramenta desenvolvida durante este processo. Em conjunto com *Docker Compose*, sendo uma ferramenta que auxilia a gestão de múltiplos *containers*, é possível gerir facilmente as várias aplicações em paralelo.

As principais vantagens retiradas desta tecnologia neste projeto são, tal como mencionado, o facto de conseguirmos ter as várias aplicações a correr em simultâneo na mesma máquina sem conflitos, visto que cada uma delas vive no seu contentor e não sabe da existência das outras aplicações.

Outra vantagem, é o facto destes contentores serem agnósticos ao sistema operativo e/ou tecnologia. Por exemplo, tal como veremos, as aplicações estudadas foram desenvolvidas em PHP, a ferramenta desenvolvida em *Java* e foram utilizadas várias bases de dados. Com o *Docker* não existe necessidade de instalar nenhuma destas tecnologias diretamente na máquina com as versões requeridas. Estas são definidas num ficheiro de configurações do *Docker* e apenas são instaladas nos contentores. Isto permite que possamos facilmente executar este processo noutra máquina física, sem problemas de versões ou diferentes sistemas. Tal como veremos de seguida, isto permitiu que os testes fossem realizados num *Raspberry PI* sem qualquer problema.

No Apêndice A.1 é possível observar o ficheiro de configurações desenvolvido para executar as aplicações testadas.

2 http://dotcomsecure.com/portal.php
3 https://www.docker.com/

2.5.2 *Bricks*

O *Bricks* é uma aplicação *web* propositadamente vulnerável com o intuito de perceber como é que algumas vulnerabilidades podem ser exploradas[4]. A aplicação foi desenvolvida em PHP usando uma base de dados *MySQL*[5]. O foco desta aplicação é conter diversas vulnerabilidades que podem ser exploradas, tanto através de uma abordagem manual, como automatizada. Sendo que não existe uma fonte oficial com o número de vulnerabilidades disponível, foi feita uma análise manual para perceber quantas vulnerabilidades do tipo SQL *Injection* existem. As secções seguintes descrevem a análise deste estudo.

A aplicação está divida em várias páginas (*Login-1.php*, *Login-2-php*, entre outras), pelo que as secções seguintes seguem esta divisão de páginas.

2.5.2.1 *Login-1*

Quando o *Login-1* é submetido ao servidor, o servidor executa uma *query* SQL para verificar se determinado utilizador e *password* existem na base de dados. A Listagem 9 mostra a porção de código executada.

```
1 <?php
2 $username=$_POST['username'];
3 $pwd=$_POST['passwd'];
4 $sql="SELECT * FROM users WHERE name='$username' and password='$pwd'";
5 ?>
```

Listagem 9: Extrato de código da página *Login-1* do *Bricks*

Para ultrapassar este mecanismo de autenticação, é necessário injetar campos válidos. A alteração deve permitir que o resultado gerado pela execução da *query* SQL seja válido. Existem várias possibilidades para explorar esta vulnerabilidade.

```
1     -- usando ' OR '1'='1 como input
2     SELECT *
3     FROM users
4     WHERE name='' OR '1'='1' and password='' or '1'='1'
5 
6     -- usando 1' OR 1=1 -- para o username and qualquer coisa para password
7     SELECT * FROM users
8     WHERE name='1' or 1=1 -- -'and password='QualquerCoisasServe'
```

Listagem 10: Possíveis vetores de ataque para *Login-1* do *Bricks*

4 https://sechow.com/bricks/
5 https://www.mysql.com/

2.5.2.2 *Login-2*

O código do lado do servidor é exatamente igual ao mencionado na Secção 2.5.2.1. O que difere no *Login-2*, é que é feita uma validação do lado do cliente que não permite ao utilizador enviar no formulário caracteres especiais. No entanto, as validações do lado do cliente não garantem proteção nenhuma contra SQL *Injection*.

De forma a contornar a validação do cliente, apenas é necessária uma ferramenta como o *BurpSuite*[6]. Esta ferramenta pode ser configurada com um *Proxy* e desta forma intercetar todos os pedidos. A Figura 7 mostra um pedido intercetado pelo *BurpSuite*. Como o cliente não permitia enviar caracteres especiais, enviamos username=aa e passwd=bb. Depois intercetamos o pedido e alteramos o *username* para um vetor de ataque igual ao utilizado na Secção 2.5.2.1. Como o servidor não tem mecanismos de proteção, é igualmente vulnerável a estes ataques e conseguimos fazer o *login* com sucesso.

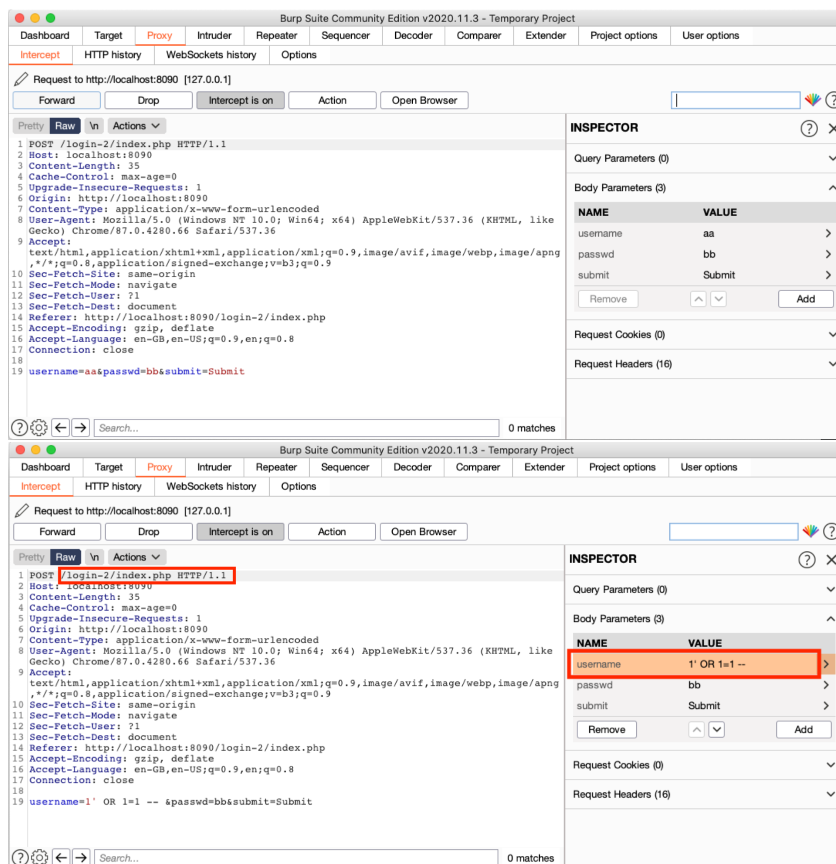

Figura 7: Pedido intercetado pelo *BurpSuite* antes e após ser alterado

6 https://portswigger.net/burp

2.5.2.3 *Login-3*

A *query* usada no *Login-3* é ligeiramente diferente das anteriores, as variáveis foram colocadas entre parênteses e foi acrescentado um limite aos resultados obtidos, tal como pode ser observado na Listagem 11.

Estas alterações, contudo, não facultam nenhuma proteção contra-ataques de SQL *Injection*. Alguns exemplos de vetores podem ser encontrados na Listagem 12.

```
<?php
$username=$_POST['username'];
$pwd=$_POST['passwd'];
$sql="SELECT * FROM users WHERE name=('$username') and password=('$pwd') LIMIT
↪  0,1";
?>
```

Listagem 11: Extrato de código da página *Login-3* do *Bricks*

```
    -- usando ') or 1=('1 como input
    SELECT * FROM users
    WHERE name=('') OR 1=('1')
    AND password=('') OR 1=('1')
    LIMIT 0,1

    -- usando ') or 1=('1')--  para o username and qualquer coisa para password
    SELECT * FROM users
    WHERE name=('') OR 1=('1')-- ') and password=('QualquerCoisa')
    LIMIT 0,1
```

Listagem 12: Possíveis vetores de ataque para *Login-3* do *Bricks*

2.5.2.4 *Login-4*

O *Login-4* difere do primeiro, na medida em que a *query* foi construída usando aspas (") em vez de plicas ('), tal como pode ser observado na Listagem 13. Todavia, isto não faculta nenhum tipo de proteção contra SQL *Injection*. Os novos vetores de ataque apenas necessitam de conter as devidas alterações, tal como apresentado na Listagem 14.

```
<?php
$username=$_POST['username'];
$pwd=$_POST['passwd'];
$username = '"' . $username . '"';
$pwd = '"' . $pwd . '"';
$sql="SELECT * FROM users WHERE name=($username) and password=($pwd)";
?>
```

Listagem 13: Extrato de código da página *Login-4* do *Bricks*

```
1       -- usando ") or 1=("1 como input
2       SELECT * FROM users
3       WHERE name=("") OR 1=("1")
4       AND password=("") OR 1=("1")
5
6       -- usando ") or 1=("1")--  para o username e qualquer coisa para password
7       SELECT * FROM users
8       WHERE name=("") OR 1=("1") -- -") and password=("Qualquercoisa")
```

Listagem 14: Possíveis vetores de ataque para *Login-4* do *Bricks*

2.5.2.5 *Login-5*

O *Login-5* tem como particularidade, o uso da função *MD5* para a *password*. Apesar de isto complicar bastante a injeção de um vetor de ataque no segundo parâmetro, uma forma de ultrapassar este mecanismo de segurança é usar um comentário no primeiro vetor, para que o segundo parâmetro seja ignorado, tal como evidenciado na Listagem 16.

```
1  <?php
2  $username=$_POST['username'];
3  $pwd=md5($_POST['passwd']);
4  $sql="SELECT * FROM users WHERE name='$username' and password='$pwd'";
5  ?>
```

Listagem 15: Extrato de código da página *Login-5* do *Bricks*

```
1       -- usando 1' OR 1=1 -- para o username and qualquer coisa para password
2       SELECT * FROM users
3       WHERE name='1' or 1=1 -- -'and password='QualquerCoisasServe'
```

Listagem 16: Possíveis vetores de ataque para *Login-5* do *Bricks*

2.5.2.6 *Login-6*

Quando um utilizador abre este formulário, está automaticamente autenticado. O código vulnerável é semelhante ao da Secção 2.5.2.1. A única diferença é que é necessário satisfazer mais uma condição para chegar à vulnerabilidade em questão, tal como podemos constatar na Listagem 17. Para isso, basta apenas enviar na *query* uma variável "p" com o valor *login*.

```
1  <?php
2  if (isset($_GET["p"])) { }
3  ?>
```

Listagem 17: Extrato de código da página *Login-6* do *Bricks*

2.5.2.7 *Content-1*

A Listagem 18 mostra uma porção do código do *Content-1* do *Bricks*. Tal como podemos observar, esta está vulnerável a ataques de SQL *Injection*. Sendo muito semelhante às *queries* anteriores, a Listagem 19 mostra possíveis vetores que poderiam ser utilizados.

```
1  <?php
2  $id=$_GET['id'];
3  if(isset($_GET['id'])) {
4          $sql = "SELECT * FROM users WHERE idusers=$id LIMIT 1";
5          $result=mysql_query($sql);
6  }
7  ?>
```

Listagem 18: Extrato de código da página *Content-1* do *Bricks*

```
1      -- usando 1 OR 1=1 -- para o username and qualquer coisa para password
2      SELECT * FROM users WHERE idusers=1 OR 1=1 LIMIT 1
3
4      -- 0 and 1=2 UNION SELECT user(),2,3,4,5,6,7,8
5      SELECT * FROM users WHERE idusers=0 and 1=2 UNION SELECT
       ↪  user(),2,3,4,5,6,7,8 LIMIT 1
6
7      -- 0 and 1=2 UNION SELECT version(),2,3,4,5,6,7,8
8      SELECT * FROM users WHERE idusers=0 and 1=2 UNION SELECT
       ↪  version(),2,3,4,5,6,7,8 LIMIT 1
```

Listagem 19: Possíveis vetores de ataque para *Content-1* do *Bricks*

2.5.2.8 *Content-2*

A Listagem 20 mostra o código respetivo do *Content-2* do *Bricks*. Tal como podemos ver é muito semelhante ao do *Content-1*, a exceção é que usamos a variável *user* em vez da variável *id*.

```
1  <?php
2  $user=$_GET['user'];
3  if(isset($_GET['user'])) {
4          $sql = "SELECT * FROM users WHERE name='$user'";
5          $result=mysql_query($sql);
6  }
7
8  ?>
```

Listagem 20: Extrato de código da página *Content-2* do *Bricks*

2.5.2.9 *Content-3*

O *Content-3* apresenta do lado do cliente um seletor para escolher o utilizador, contudo tem o mesmo problema dos anteriores. Para ultrapassar este mecanismo de segurança, apenas precisamos de utilizar a mesma técnica apresentada na Secção 2.5.2.2.

```
<?php
if(isset($_POST['submit'])) {

        $username=$_POST['username'];
        $sql="SELECT * FROM users WHERE name='$username'";
?>
```

Listagem 21: Extrato de código da página *Content-3* do *Bricks*

2.5.2.10 *Content-4*

O *Content-4* tem a particularidade de usar a variável $_SERVER com o HTTP_USER_AGENT. Mais uma vez é possível alterar este valor com uma ferramenta como o *BurpSuite*, tal como podemos observar na Figura 8.

```
<?php
        $uagent = $_SERVER['HTTP_USER_AGENT'];
        $sql= "SELECT * FROM users WHERE ua='$uagent' ";
?>
```

Listagem 22: Extrato de código da página *Content-4* do *Bricks*

2.5.2.11 *Content-5*

O *Content-5* apresenta uma *query* vulnerável a SQL *Injection* e outra onde o *input* é sanitizado, ou seja, temos uma query vulnerável e outra que não é vulnerável.

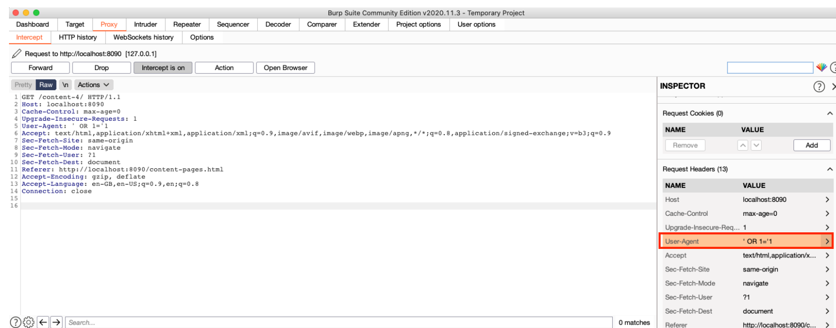

Figura 8: Pedido intercetado pelo *BurpSuite* com a alteração do *User Agent*

2.5.2.12 *Content-6*

A situação da página do *Content-6*, apesar de parecer diferente dos outros casos, é apenas uma variação. Tal como podemos observar na Listagem 23, é usada uma função para descodificar o *id* que é recebido por parâmetro em base 64. Se, por exemplo, enviarmos o vetor `JycgT1IgMTOxICOtIA==`, este ao ser descodificado tornar-se-á `'' OR 1=1 --` e, portanto, será vulnerável, tal como vimos anteriormente.

```
1 <?php
2         $id=base64_decode($_GET['id'],true);
3     $sql = "SELECT * FROM users WHERE idusers=$id LIMIT 1";
4 ?>
```

Listagem 23: Extrato de código da página *Content-6* do *Bricks*

2.5.2.13 *Bricks Sumário*

Pela análise descrita nas secções anteriores podemos concluir que o *Bricks* apresenta 12 vulnerabilidades do tipo SQL *Injection*. Este valor será importante, pois servirá como base de comparação para a análise comparativa entre as diversas ferramentas automatizadas e a ferramenta desenvolvida.

2.5.3 *bWAPP*

bWAPP[7] é uma aplicação web gratuita, de código aberto, propositadamente vulnerável. O principal objetivo é ajudar entusiastas, estudantes ou profissionais da área a descobrir e prevenir vulnerabilidades *web*. A aplicação tem mais de cem vulnerabilidades, cobrindo as vulnerabilidades mais conhecidos e incluídas na OWASP.

A aplicação é desenvolvida em PHP com uma ligação a uma base de dados *MySQL*[8].

Visto que não existe uma fonte oficial relativamente ao número de vulnerabilidades de SQL *Injection* existentes nesta aplicação, procedeu-se a uma primeira análise manual.

Desta análise, conclui-se que existem 56 vulnerabilidades do tipo SQL *Injection*. Devido à extensão de vulnerabilidades encontradas, apenas serão apresentadas algumas que não tenham sido mencionados na Secção 2.5.2.

7 http://itsecgames.com/
8 https://www.mysql.com/

2.5.3.1 *Obter o esquema da Base de dados*

Utilizando a técnica demonstrada na Secção 2.2.2, com ligeiras variações, conseguimos obter as várias tabelas que compõem a base de dados associada a este projeto. A Figura 9 mostra um campo de pesquisa possível de encontrar nesta aplicação. Dado que temos acesso ao código-fonte, a análise não é uma análise *black box*, em que apenas temos o contexto externo. Com uma análise *black box* seria possível chegar à mesma conclusão, quer seja por métodos de dedução ou de força bruta, sendo que apenas demoraria mais tempo.

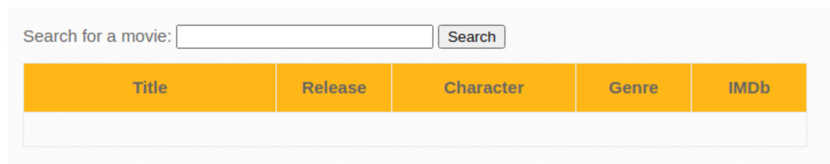

Figura 9: Campo de pesquisa no *bWAPP*

A Listagem 24 demonstra a *query* executada quando uma pesquisa é efetuada no campo de pesquisa da Figura 9.

```
1  <?php
2
3  $sql = "SELECT * FROM movies WHERE title LIKE '%" . $title . "%'";
4
5  ?>
```

Listagem 24: Código *SQL* executado durante uma pesquisa no *bWAPP*

Com base nestes dados e, se procedermos a um vetor de ataque mais complexo como o apresentado na Listagem 25, é possível obter dados explícitos do esquema da base de dados.

```
1      iron' union select 1,user(),database(),(select
   ↪  GROUP_CONCAT(column_name,'\n') from information_schema.columns where
   ↪  table_name='users'),version(),6,7-- - %'
```

Listagem 25: Possível vetor de ataque para o campo de pesquisa do *bWAPP*

A Figura 10 mostra o nome das tabelas existentes na aplicação, através do uso da injeção do vetor de ataque mencionado na Listagem 25.

Search for a movie: iron' union select 1, user(), dat Search

Title	Release	Character	Genre	IMDb
root@172.31.0.1	bwapp	8.0.22	email ,host ,idusers ,lang ,name ,password ,ref ,ua ,activated ,activation_code ,admin ,email ,id ,login ,password ,reset_code ,secret ,CURRENT_CONNECTIONS ,TOTAL_CONNECTIONS ,USER ,password ,uid ,username	Link

Figura 10: Obtenção do nome de tabelas do *bWAPP* com SQL *Injection*

2.5.3.2 *bWAPP Sumário*

O bWAPP é uma ferramenta muito interessante que permite aprender bastante relativamente a vulnerabilidades *web* e, ao contrário das outras aplicações estudadas, contém um número elevado de vulnerabilidades, o que irá permitir estudar a ferramenta num ambiente com mais linhas de código e com mais entropia. Serão consideradas 56 vulnerabilidades existentes neste projeto do tipo SQL *Injection*.

2.5.4 *Twitterlike*

Twitterlike é outra aplicação de código aberto[9] com uma funcionalidade muito básica semelhante à rede social *Twitter*, mas vulnerável a ataques de SQL *Injection*.

A aplicação foi desenvolvida em PHP e tem como suporte uma base de dados relacional *MySQL*.

Após uma análise manual concluiu-se a existência de 17 *queries* vulneráveis a estes ataques.

2.6 FERRAMENTAS AUTOMÁTICAS

Para verificar a eficácia da abordagem sugerida com o uso de algoritmos de IA, é importante fazer-se a comparação não só com um método manual, mas também com métodos automáticos. Existem várias ferramentas automatizadas para a deteção de vulnerabilidades. Nas secções seguintes são apresentadas as ferramentas utilizadas,

9 https://github.com/kevinbaptist/twitterlike

sendo que na Secção 2.6.1 é introduzido o WAP, uma ferramenta *open-source* e, na Secção 2.6.2, o *SonarPHP*, uma ferramenta comercial.

2.6.1 *Web Application Protection*

WAP[10] é uma ferramenta de análise estática de código-fonte, cujo intuito é detetar e corrigir vulnerabilidades encontradas em aplicações PHP, que segundo os autores tem uma baixa percentagem de falsos positivos. Esta ferramenta deteta vulnerabilidades:

- SQL *Injection*;
- XSS;
- *Remote File Inclusion* (RFI);
- *Local File Inclusion* (LFI);
- *Operating System Command Injection* (OSCI).

De forma geral, esta ferramenta analisa o fluxo e semântica do código-fonte e faz uma análise comparativa para detetar vulnerabilidades em termos de validação de dados. O objetivo da análise é seguir o fluxo de dados de entrada através de variáveis como $_GET e $_POST e, no final, a ferramenta usa *data mining* para confirmar se a vulnerabilidade é real ou se é um falso positivo.

A ferramenta foi desenvolvida em *Java* e é constituída por 3 módulos principais: análise de código, detetor de falsos positivos e um corretor de código. Apenas foram estudados os dois primeiros módulos.

2.6.2 *SonarPHP*

SonarPHP[11] é uma ferramenta desenvolvida por *SonarSource* que usa padrões e análise de fluxo, para detetar vulnerabilidades em código PHP. Trata-se uma análise estática com base em regras bem definidas. Esta ferramenta disponibiliza mais de 140 regras, suportando regras personalizadas.

O módulo *SonarPHP* é específico para projetos em PHP, no entanto, a *SonarSource* tem módulos para mais de 20 linguagens, sendo *Java*, *C#*, *Go*, *Python* apenas alguns dos exemplos. Esta ferramenta pode ser integrada diretamente num

10 http://awap.sourceforge.net/
11 https://www.sonarsource.com/php/

editor como *IntelliJDEA*[12] ou *Visual Studio Code*[13] ou num sistema de integração contínua (*continuous integration*) tal como *Gitlab*[14].

A ferramenta é composta por 248 regras, capaz de identificar 38 tipos de vulnerabilidades[15].

12 https://www.jetbrains.com/idea/
13 https://code.visualstudio.com/
14 https://about.gitlab.com/
15 https://rules.sonarsource.com/php

3 METODOLOGIA

Nos últimos anos surgiram metodologias ágeis, como alternativa às metodologias tradicionais de desenvolvimento de *software* (Ilieva et al., 2004). Alguns dos exemplos de metodologias mais populares são *Scrum*, *eXtreme Programming* (XP) e *Feature Driven Development* (FDD) e *Kanban*.

Neste capítulo será descrita a metodologia de desenvolvimento utilizada no desenvolvimento do projeto.

3.1 QUADRO *KANBAN*

Kanban é uma metodologia ágil com um fluxo visual muito popular no desenvolvimento de *software* (Kile, 2007). Desenvolvido originalmente por Toyota em 1940, este modelo foi inspirado em lojas de mercado, que apenas acumulavam produtos em *stock* consoante a necessidade. As versões iniciais eram uma abordagem *Just In Time* (JIT), isto é, as fábricas apenas poderiam criar as peças necessárias no momento para não desperdiçar recursos desnecessários.

Como qualquer outra metodologia ágil, o *Kanban* foi pensado para melhorar o trabalho de uma equipa de desenvolvimento e, é composto por vários componentes essenciais: quadro *Kanban* (Secção 3.1), os cartões (Secção 3.2), *Work In Progress* (WIP) (Secção 3.3), *backlogs* (Secção 3.4) e *Continuous releases* (Secção 3.5).

O quadro é uma forma de visualização do projeto da equipa que adiciona transparência no projeto. No contexto deste projeto foi utilizada uma versão simplificada com apenas três colunas: *To Do* (por fazer), *In Progress* (em progresso), *Done* (terminado).

3.2 CARTÕES *KANBAN*

Um cartão no quadro de *Kanban* é uma representação de uma tarefa a ser desenvolvida.

A Figura 11 mostra um exemplo esquemático do quadro de *Kanban*, utilizado durante a realização deste projeto. Neste exemplo existem três colunas (*selected for development*, *in progress* e *done*), sendo que os retângulos representam um cartão de *Kanban*, isto é uma tarefa. Os números junto a cada coluna representam o número de tarefas nessa coluna. Neste exemplo podemos observar que há duas tarefas prontas a serem desenvolvidas (*selected for development*), uma que está em progresso e, por fim, seis que já foram terminadas (na versão atual).

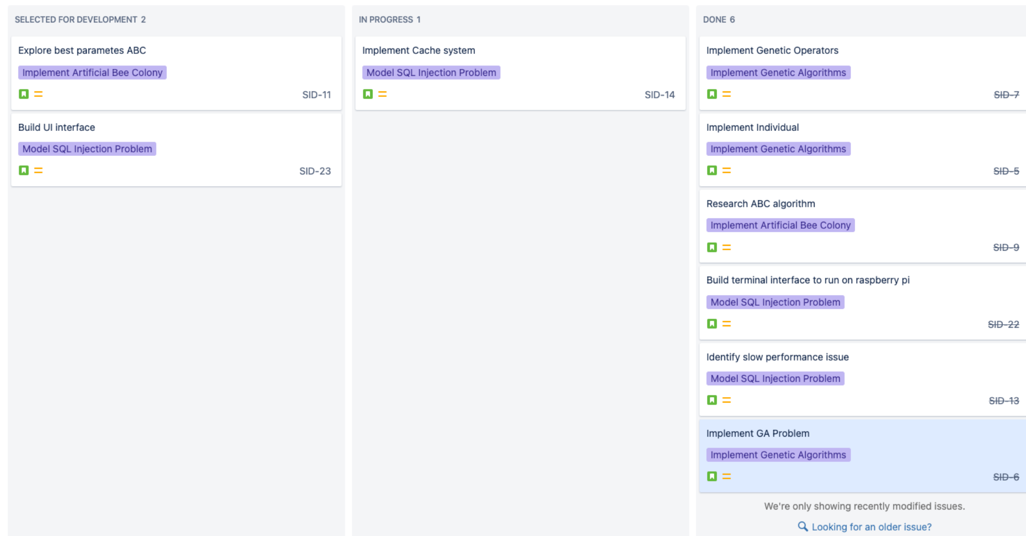

Figura 11: Exemplo de um quadro de *Kanban*

3.3 WORK IN PROGRESS

Tendo em conta a formulação original desta metodologia, é necessário definir algumas regras para que não haja desperdício de recursos. A ideia é limitar o trabalho na coluna do *em progresso*. Para tal, é necessário terminar tarefas em desenvolvimento antes de começar novas tarefas. Isto ajuda as equipas a não terem demasiado trabalho em paralelo, mas no contexto do projeto atual, beneficia a entrega de valor no fim de cada iteração de trabalho.

3.4 BACKLOGS

Sempre que existem novas tarefas que devem ser desenvolvidas, estas são adicionadas ao *backlog* para serem analisadas e desenvolvidas no futuro. Sendo o projeto

atual autoproposto, esta tarefa resultou da combinação do autor do projeto e das orientadoras.

3.5 *CONTINUOUS RELEASES*

Ao contrário de outras metodologias ágeis como o *Scrum* que se focam em torno de sucessivas iterações designadas como *sprints*, o *Kanban* não tem iterações bem definidas. Isto, contudo, não significa que não exista planeamento, apenas não se está limitado a limitações de iterações predeterminadas.

3.6 SUMÁRIO

O projeto envolveu uma fase de investigação, de implementação, de recolha e análise de resultados, portanto, uma metodologia ágil permitia uma adaptação em tempo real das novas necessidades do projeto. Por este motivo, as abordagens *waterfall*, em que todo o desenvolvimento é delineado ao início, não seriam viáveis devido à enorme quantidade de variáveis desconhecidas.

Dentro das metodologias ágeis existem vários modelos e *frameworks*. Dentro das mesmas, decidiu-se prosseguir com o modelo *Kanban* por se tratar de uma abordagem flexível. Esta flexibilidade oferece liberdade para desenvolver o projeto com tempo bastante limitado, tal como a capacidade de adaptar um projeto de investigação e tomar uma nova direção a qualquer altura, se necessário.

4

DESENVOLVIMENTO

Este Capítulo começa com modelação do problema (Secção 4.1), isto é, como o processo foi dividido. Nas secções seguintes são apresentados alguns aspetos técnicos relativamente à implementação de cada algoritmo, nomeadamente AG (Secção 4.2), ABC (Secção 4.3) e ACO (Secção 4.4).

Na Secção 4.5 é apresentada uma melhoria efetuada que beneficia todos os algoritmos implementados, por colocar alguns valores em *cache*, diminuindo assim o tempo de execução total.

Durante a realização deste projeto foi desenvolvida uma versão gráfica (Secção 4.6) e uma versão a partir do terminal (Secção 4.7). As configurações inerentes a cada versão, bem como parâmetros das mesmas, são demonstrados na Secção 4.8.

Por fim, as tecnologias utilizadas e o diagrama da aplicação são apresentados nas Secções 4.9 e 4.10.

4.1 MODELAÇÃO DO PROBLEMA

Para aplicar algoritmos de IA para a deteção de vulnerabilidades SQL *Injection*, é necessário modelar o problema num problema de otimização. A solução apresentada é dividida em duas etapas principais:

- Fase preliminar com o objetivo de encontrar todas as *queries* SQL;
- Fase de deteção, com a aplicação de algoritmos de IA, com o intuito de detetar vulnerabilidades.

Este trata-se de um problema de maximização, ou seja, o objetivo é encontrar o máximo número de vulnerabilidades no código.

Tal como veremos, a fase de deteção teve duas implementações diferentes, cujos resultados são apresentados no Capítulo 5. Para diferenciar estas implementações, estas serão designadas como versão 1 e versão 2. As diferenças de implementação são evidenciadas na Secção 4.2.

4.1.1 *Fase preliminar*

Esta fase é responsável por detetar todas as *queries* SQL existentes num projeto. A Figura 12 apresenta, de uma forma esquemática, a fase preliminar executada inicialmente.

De uma forma resumida, após a identificação de todos os ficheiros recursivamente de uma pasta, é feita a indexação de todas as variáveis bem como do seu histórico. Isto é efetuado para se perceber quais foram as validações efetuadas antes de as *queries* SQL serem executadas. Todas as *queries* não parametrizadas são adicionadas a uma lista que será utilizada pelos algoritmos de IA implementados, numa segunda fase. Podem existir, contudo, *queries* que foram parametrizadas, mas ainda assim estão vulneráveis. Para estas, é verificado se o valor de entrada sofreu alguma mutação antes de ter sido executado. Caso se verifique esta situação, a *query* também será adicionada à lista mencionada anteriormente.

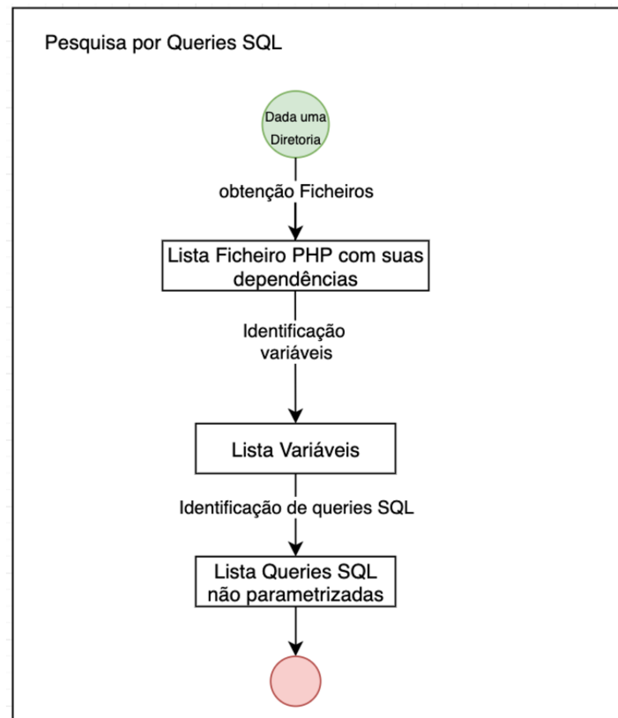

Figura 12: Fase preliminar no processo de deteção de vulnerabilidades

De forma a identificar se determinada linha de código contém uma *query* SQL é utilizada a seguinte expressão regular:

`((SELECT|UPDATE|INSERT|DELETE).*WHERE.*?([;"]))$` /mi,

com as *flags* *m* (múltiplas linhas) e *i* (insensível às maiúsculas).

4.1.2 *Fase de deteção*

A Figura 13 representa uma visão de alto nível do processo, em que é utilizado um AG. A fase de deteção começa com a lista de *queries* identificadas previamente e, está representada na figura como “Lista de *Queries*”. Na versão 1, a primeira etapa é gerar uma nova população com base num conjunto de vetores de ataque. Na versão 2, na primeira etapa, é também gerada uma nova população, mas com base num conjunto de *strings* (que podem constituir vetores de ataques simples ou formar, pela sua concatenação, um vetor de ataque mais complexo). De seguida, é necessário calcular o *fitness* para cada indivíduo da população (o *fitness* é um valor que determina a qualidade do indivíduo, ou seja, a qualidade da possível solução). Este processo continua até que uma das condições de paragem seja verdadeira, tal como demonstrado na Listagem 6.

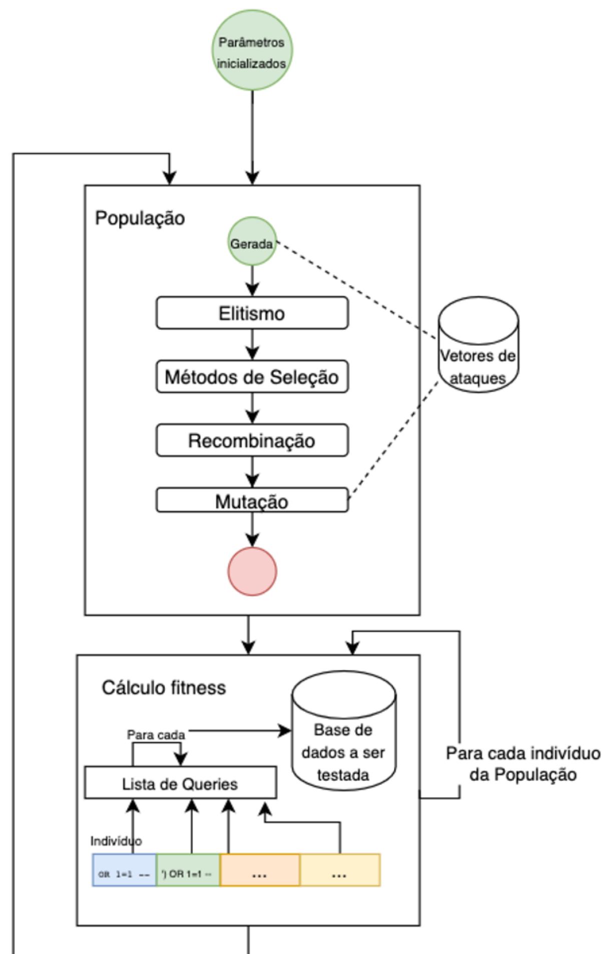

Figura 13: Fase de deteção de vulnerabilidades com algoritmos genéticos

4.2 ABORDAGEM TÉCNICA — ALGORITMOS GENÉTICOS

Após recolher todas as *queries* de uma dada pasta, a etapa seguinte é detetar as vulnerabilidades. As próximas secções descrevem a adaptação dos AG para gerar vetores de SQL *Injection*. De forma a aplicar estes algoritmos ao problema mencionado é necessário:

- Definir como o indivíduo é representado;
- Definir uma população de indivíduos;
- Definir como são aplicados os operadores;
- Definir a função de *fitness*.

A Figura 14 mostra as etapas do AG quando aplicado ao problema SQL *Injection*. Nas secções seguintes cada etapa é explicada detalhadamente.

Valor de Entrada: Q - Conjunto de ***Queries***

Processo:

1. Iniciar População (P, tamanho)
2. Avaliar a populaçao
3. Enquanto n_geração_atual < max_gerações
    - 3.1 Executar Operadores para criar uma nova população
        - *3.1.1. Elitismo*
        - *3.1.2. Seleção*
        - *3.1.3. Recombinação*
        - *3.1.4. Mutação*
    - 3.2. Avaliar a Solução
    - 3.3. Incremento n_geração_atual

Figura 14: Etapas do AG aplicado a SQL *Injection*

4.2.1 *Parâmetros de entrada*

De seguida são apresentados os parâmetros de entrada, para o AG:

- ***population_size*** define o tamanho da população. Valor numérico e positivo.
- ***elitism_size*** define o valor de elitismo. Valor de 0 significa que não é aplicado elitismo, outro valor garante que um determinado número de indivíduos passa às gerações seguintes.
- ***selection*** define a seleção escolhida. Valores possíveis: *tournament* e *roulette_wheel*.

- ***tournament_size*** define o tamanho do torneio. Apenas é obrigatório se a opção anterior for *tournament*. Valor numérico positivo.
- ***recombination*** define a recombinação a ser usada. Valores aceites são *one_cut*, *two_cuts* e *uniform*.
- ***recombination_probability*** define o valor de probabilidade para uma recombinação ocorrer. Valor tem de estar entre 0 e 1. Sendo que com o valor 0 não ocorre recombinação e 1 ocorre sempre.
- ***mutation*** define a mutação a ser usada. Valores aceites são *string*.
- ***mutation_probability*** define o valor de probabilidade para uma mutação ocorrer. Valor tem de estar entre 0 e 1. Sendo que 0 não ocorre mutação e 1 ocorre sempre.

4.2.2 *Representação do Indivíduo*

Tal como mencionado anteriormente, foram implementadas duas versões da fase de deteção. Estas implementações estão diretamente correlacionadas com a representação do indivíduo.

A Figura 15 demonstra a representação de um indivíduo usando a versão 1. Neste exemplo, temos um indivíduo com cinco genes, onde cada gene é um vetor de ataque completo. Seguindo esta abordagem, um indivíduo pode gerar até N vetores de ataque diferentes, sendo N o número de genes do indivíduo.

Figura 15: Representação de um indivíduo — versão 1

A Figura 16 mostra um exemplo de um indivíduo que poderia ser usado na versão 2. Nesta versão um indivíduo apresenta quatro genes, sendo que cada gene é parte do vetor de ataque. Para obtermos um vetor de ataque completo é necessário concatenar todos os genes do indivíduo, embora que cada gene também pode ser visto como um vetor de ataque simples. Seguindo esta abordagem, um indivíduo pode gerar até $N+1$ vetores, sendo N o número de genes do indivíduo.

Figura 16: Representação de um indivíduo — versão 2

4.2.3 *Geração da população inicial*

A população inicial é construída com base num conjunto de indivíduos. Estes indivíduos são preenchidos inicialmente com base num conjunto de valores obtidos previamente. Estes foram recolhidos a partir de vários *datasets*[1,2]. A Figura 17 mostra um exemplo de uma população inicial usando a versão 2.

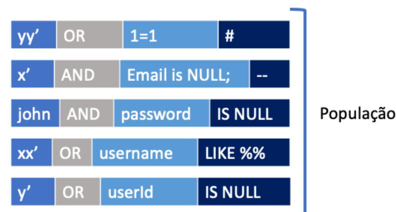

Figura 17: Exemplo de uma população

4.2.4 *Operadores Genéticos*

A recombinação de indivíduos de uma população acontece tal como descrito na Secção 2.3.1.3. A Figura 18 mostra um exemplo de uma recombinação de um corte, em que com base em dois indivíduos, existe um corte a partir do qual os genes de ambos os indivíduos serão trocados.

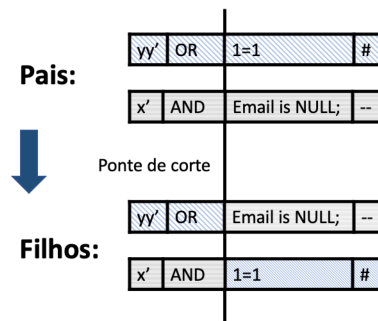

Figura 18: Exemplo de uma recombinação aplicada ao problema SQL *Injection*

A mutação de um indivíduo, tal como observado na Secção 2.3.1.4, pode ser vista como uma alteração aleatória de um gene por outro. A Figura 19 representa uma situação em que 2 indivíduos têm 1 gene alterado. No primeiro indivíduo, o segundo

1 http://www.unixwiz.net/techtips/sql-injection.html
2 https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/

gene é trocado para *OR*, no caso do segundo indivíduo o seu terceiro gene é trocado para --.

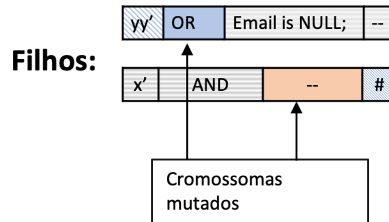

Figura 19: Exemplo de uma mutação aplicada ao problema SQL *Injection*

4.2.5 *Fitness*

A Figura 20 resume as etapas necessárias para avaliar cada indivíduo de uma população. Tal como se pode observar na Secção 4.2.2, um indivíduo é composto por um conjunto de vetores que representam os seus genes. Portanto, dado um conjunto de *queries* inicial, encontradas na fase preliminar, cada gene do indivíduo irá substituir a variável mutável existente na *query* e é verificado se essa substituição se reproduz numa *query* válida e vulnerável. Por fim, e apenas na versão 2, é feito o mesmo para o valor (*string*) que resulta da concatenação dos genes.

Valor de Entrada: Q - Conjunto de *Queries*

Processo:
Para cada **indivíduo** na População
Para cada **gene** no Indivíduo
Para cada ***query*** em Q
Substituição da variável mútavel na **query** pelo **gene**
Concatenação dos Genes do indivíduo
Para cada ***query*** em Q
Substituição da variável mútavel na **query** pela **concatenação**
Apenas versão 2
Cálculo do fitness do indivíduo.

Figura 20: Etapas para calcular o *fitness*

Para melhor ilustrar a situação é apresentada a Figura 21. A *query* analisada é possível ser observada no projeto *bWAPP*. Basicamente são filtrados todos os *movies* (ou filmes), cuja coluna *name* seja igual ao valor associado à variável *$name*. O indivíduo estudado é constituído por quatro genes: *yy'*, *OR*, *1=1* e -- (valores possíveis de usar na versão 2). Cada um dos genes é testado individualmente como valor de *$name* e, por fim, como um conjunto *yy' OR 1=1 –*.

Figura 21: Exemplo do indivíduo, versão 2, aplicado a uma *query*

No exemplo dado, apenas o primeiro gene testado individualmente conduz a uma *query* inválida, devido à existência de dois caracteres ' seguidos. Contudo, apenas o conjunto dos genes do indivíduo se traduz numa *query*, que para além de ser válida é vulnerável, pois permite obter todos os resultados existentes nesta tabela.

Após perceber como é avaliado um indivíduo, é necessário traduzir o seu desempenho num número. Para o efeito, foi usada a função de *fitness* apresentada na Equação 1, sendo T_Genes o número total de genes do indivíduo; $Uvul$ o número único de vulnerabilidades detetadas pelo indivíduo; e $Gvul$ o número de genes diferentes que detetou pelo menos uma vulnerabilidade.

$$Fitness = \frac{U_{vul} * 5 + G_{vul} * 2}{T_Genes} \tag{1}$$

Tendo em conta o exemplo apresentado na Figura 21, o indivíduo apresenta quatro genes, pelo que $t_Genes = 4$. Para a situação apresentada foi apenas analisada uma *query*, em que o indivíduo conseguiu detetar uma vulnerabilidade, portanto o valor de $Uvul = 1$. Relativamente aos cinco vetores de ataques gerados, apenas um obteve sucesso, pelo que o valor de $Gvul = 1$.

Para a versão 2 do problema será muito comum o valor de $Uvul$ e $Gvul$ serem iguais, pois o indivíduo foi construído de modo a que a concatenação de todos os seus genes gere um ataque válido. No entanto, na versão 1, cada gene é um vetor de ataque completo, pelo que será muito comum vermos um indivíduo que gerou múltiplos ataques válidos ($Gvul$) para a mesma *query* ($Uvul = 1$).

Na função de *fitness* apresentada na Equação 1, podem-se observar pesos diferentes associados aos valores de *Uvul* e *Gvul*, sendo que o *Uvul* apresenta um peso de 5, enquanto o *Gvul* apresenta um valor de 2, tendo desta forma menos impacto no cálculo do *fitness*. O desenvolvimento desta função requereu várias tentativas. A existência da variável *Uvul* é clara, o objetivo principal de um indivíduo é encontrar vulnerabilidades. Quanto mais encontrar, melhor é considerado este indivíduo. A variável *Gvul* pode não ser tão evidente. Sem a existência da mesma, para um indivíduo da versão 1, existia uma tendência para que quando o indivíduo encontrasse um vetor válido, este o replicasse para todos os seus genes. Consideremos o indivíduo representado na Figura 15. Se o gene *– 1=1* fosse válido, notaríamos uma tendência em que o melhor indivíduo era constituído apenas por quatro genes iguais.

Para contornar esta situação, foram delineadas várias possibilidades: prejudicar indivíduos com o mesmo gene repetido, ou beneficiar indivíduos com genes diferentes. A última abordagem revelou obter melhores resultados. Com a introdução desta variável foi então necessário aplicar diferentes pesos às variáveis discutidas, caso contrário, um indivíduo com genes diferentes, mas sem vulnerabilidades encontradas, era tão beneficiado como outro que obteve vulnerabilidades. A proporção com melhores resultados foi a de 5:2.

4.3 ABORDAGEM TÉCNICA — *ARTIFICIAL BEE COLONY*

No algoritmo ABC o indivíduo é construído da mesma forma que no AG, sendo que também existem duas versões para a criação do indivíduo. De facto, o cálculo do *fitness* também é igual, sendo que as diferenças assentam nas etapas do algoritmo. O algoritmo ABC não apresenta os operadores genéticos.

Na Listagem 26 é apresentado um extrato do código executado para o algoritmo ABC. Tal como apresentado na Secção 2.3.2, o algoritmo apresenta 3 etapas principais: fase das abelhas empregadas, fase das abelhas espetadoras e fase das abelhas exploradoras. Cada fase é executada numa classe própria, sendo que estas são visíveis nas linhas 12, 14 e 16 respetivamente.

```
generationNumber = 0;
initialTime = System.currentTimeMillis();

employedBeesPhase = new EmployedBeesPhase<>(numberEmployedBees, problem);
bestInRun = employedBeesPhase.getBest();

bestInRun.setExecutionTime(System.currentTimeMillis() - initialTime);
fireGenerationEnded(new GAEvent<>(this));


while (terminalConditionIsNotMet()) {
        Population<I> employedBees = employedBeesPhase.run();

        onlookerBeesPhase.run(employedBees,
        ↪  employedBeesPhase.getProbabilities());

        scoutBeesPhase.run(problem, employedBees);

        Individual<I> bestInGen = employedBees.getBest();
        computeBestInRun(bestInGen);

        generationNumber++;
        fireGenerationEnded(new GAEvent<>(this));
}
```

Listagem 26: Extrato de código do algoritmo *Artificial Bee Colony*

4.3.1 *Parâmetros de entrada*

De seguida são apresentados os parâmetros de entrada, para o algoritmo ABC:

- ***number_employed_bees*** define o número de abelhas empregadoras. Valor tem de ser inferior ou igual ao número de abelhas espetadoras.
- ***number_onlooker_bees*** define o número de abelhas espetadoras. Valor numérico e positivo.
- ***number_modifications*** define o número de modificações a ocorrer. Número usado na fase das abelhas espetadoras para aplicação da busca local.

4.3.2 *Fase das abelhas empregadas*

As abelhas empregadas são responsáveis por: 1) tirar partido do néctar de zonas já exploradas e, 2) fornecer informação às abelhas espetadoras, sobre a qualidade do néctar que exploram. Nesta fase, cada abelha empregada produz uma nova fonte de alimento no local e explora a melhor. Nesta implementação, a posição de uma fonte de alimento representa uma possível solução para o problema, e o valor do néctar corresponde à qualidade da solução associada (Equação 1). O número de abelhas

empregadas é exatamente o número de soluções iniciais. As abelhas empregadas partilham a informação relacionada ao néctar das fontes e as suas posições com as abelhas espetadoras. A abelha espetadora escolhe uma fonte de alimento com base no valor de probabilidade associado a essa fonte de alimento *pri*, tal como se pode constatar pela Equação 2.

$$totalFitness = \sum_{n=1}^{ne} fit_n \qquad pr_i = \frac{fit_i}{totalFitness} \tag{2}$$

4.3.3 *Fase das abelhas espetadoras*

As abelhas espetadoras esperam na colmeia e decidem qual a fonte de alimento explorar, conforme as informações partilhadas pelas abelhas empregadas. Nesta fase, cada abelha seleciona uma fonte, dependendo da qualidade (com base no valor de *fitness*) da solução. De seguida, produz uma nova fonte de alimento na mesma posição e explora a melhor. Sendo neste caso a melhor, a que tiver um maior valor de *fitness*.

A abelha espetadora avalia a informação do néctar e escolhe a fonte de alimento com a probabilidade associada à quantidade de néctar. Nesta implementação, o algoritmo calcula o número de abelhas espetadoras com base na Equação 3, sendo que o valor de probabilidade *pri*, foi obtido a partir das abelhas empregadas (Eq. 2).

$$no_i = pr_i * no \tag{3}$$

Sendo no_i, o número de abelhas espetadoras enviadas para a fonte de alimento i.

A vizinhança é obtida através de um número de tentativas, com o objetivo de melhorar a solução, em que o número é definido pelo número de modificações (Secção 4.8). A ideia é o algoritmo executar N modificações para encontrar uma nova posição para a abelha espetadora. A modificação consiste na alteração do valor da posição de um indivíduo por outro valor aleatório (neste caso um novo vetor de ataque para a versão 1, ou uma *string* para a versão 2). O algoritmo repete este processo até que ocorra pelo menos uma melhoria ou o número de modificações seja atingido. Caso a quantidade de néctar da solução seja maior que a anterior, o valor antigo é removido, sendo substituído pelo novo.

4.3.4 *Fase das abelhas exploradoras*

As abelhas espetadoras procuram aleatoriamente no ambiente novas fontes de alimento. Nesta fase, a fonte de alimento abandonada, é substituída por novas. Nesta implementação, isto é simulado pela produção de novas soluções e substituindo as piores abelhas empregadas, ou seja, as fontes com néctar inferior são abandonadas.

Na implementação, considerou-se que o número de abelhas exploradoras é 10% do número de abelhas empregadas, sendo que o número de abelhas exploradoras é dado pela fórmula da Equação 4.

$$ns = 0.1 * ne \tag{4}$$

Sendo ne, o número de abelhas empregadas.

4.4 ABORDAGEM TÉCNICA — *ANT COLONY OPTIMIZATION*

O algoritmo ACO utiliza a mesma modulação do problema dos algoritmos anteriores, portanto, a constituição do indivíduo, assim como o cálculo do *fitness*, são efetuados da mesma forma. O algoritmo começa com a inicialização dos parâmetros de entrada, fornecidos pelo utilizador. Na Secção 4.8, cada parâmetro do algoritmo é detalhado.

Devido à dimensão do código implementado neste algoritmo, a Listagem 27 mostra apenas uma porção do código. A população inicial é criada de forma aleatória, tal como acontece nos outros algoritmos.

```
generationNumber = 0;

ants = new Population<>(numberAnts, problem);
ants.evaluate();
bestInRun = ants.getBest();

pheromonaTrail = new double[individualSize][totalValueForGene];
initPheromoneTrail();
currentIterationWithoutImprovement = 0;

while (terminalConditionIsNotMet()) {
        generationNumber++;
        for (I ant: ants) {
                I clone = ant.clone();
                for (int n = 0; n < numberModifications; n++) {
                        if (newRandom() < qProbability) {
                                exploitation()
                        } else {
                                explore()
                        }
                }
                if (intensification) {
                        if (clone.getFitness() >= ant.getFitness()) {
                                ants.swap(ant, clone);
                        }
                } else {
                        ants.swap(ant, clone);
                }
        }

        evaluate(ants);
        currentIterationWithoutImprovement++;

        updatePheromoneTrail(problem);

        if (maxIterationWithoutImprovement ==
        ↪  currentIterationWithoutImprovement) {
                initPheromoneTrail();
                regenerateNewColony();
                currentIterationWithoutImprovement = 0;
        }
}
```

Listagem 27: Extrato de código do algoritmo *Ant Colony Optimization*

4.4.1 *Parâmetros de entrada*

No ACO, o utilizador pode definir os seguintes parâmetros de entrada:

- ***population_size*** define o número de abelhas na colónia.
- ***number_modifications*** define o número de modificações.
- ***max_iter_without_improvement*** define o número máximo de iterações que pode ocorrer sem existir uma melhoria da colónia.
- ***Q*** é um parâmetro para inicializar os valores do rastro de feromonas.

- ***q_probability*** probabilidade de efetuar uma modificação. Caso seja gerado um valor aleatório superior a esta probabilidade, uma exploração é efetuada.
- ***pheromone_evaporation_rate*** define a probabilidade de evaporação da feromona.
- ***pheromone_influence_rate*** define a influência que a feromona tem nas restantes formigas.

4.4.2 *Inicialização de valores de feromonas*

Tal como podemos observar na Listagem 27, apresentada anteriormente, o rastro de feromonas é representado como uma matriz, onde, as linhas representam os genes do indivíduo e as colunas todos os valores possíveis para um gene. Estes valores diferem consoante a versão da representação do indivíduo utilizada. Todos os valores da matriz são inicializados com base na Fórmula 5, sendo que G é a melhor solução até ao momento e Q um parâmetro.

$$t_{ij} = \frac{Fitness(G)}{Q} \tag{5}$$

4.4.3 *Modificação da solução*

Esta etapa consiste em consecutivas modificações da solução. O número de modificações é um valor de entrada.

De início é gerado um valor aleatório entre 0 e 1, sendo que se o valor gerado for menor que a probabilidade q, é desencadeado um processo de obtenção do melhor rastro conhecido, ou seja, o vetor de ataque é escolhido para maximizar o rastro conhecido. Por outro lado, se o valor de probabilidade obtido for maior que a probabilidade q, o vetor é escolhido proporcionalmente ao valor contido no rastro de feromonas. Este mecanismo tem como propósito, aumentar o espaço de solução explorado.

4.4.4 *Mecanismo de intensificação*

Se o mecanismo de intensificação estiver ativo e a solução no início da iteração for melhor, a formiga volta ao estado inicial. O mecanismo é ativado quando a melhor solução encontrada melhorou e continua ativa. No fim da iteração, se não houver melhoria, a intensificação é desativada. Este mecanismo tem como principal objetivo impedir que a colónia fique retida num máximo local.

4.4.5 *Atualização do rastro de feromonas*

Para melhorar a convergência, os rastros são atualizados tendo apenas em conta a melhor solução encontrada (Bernardino et al., 2008).

Os valores são atualizados com base nas Equações 6 e 7, sendo *evaporationRate* um parâmetro percentual que controla a evaporação do rastro de feromonas; *influenceRate* um parâmetro, também em percentagem, que controla a influência que a melhor solução tem no rastro de feromonas.

$$t_{ij} = t_{ij} + (1 - evaporationRate) \tag{6}$$

$$t_{iGi} = t_{iGi} + (influenceRate * (1 - \frac{1}{Fitness(G)})) \tag{7}$$

4.5 CACHE

Devido à natureza dos algoritmos e à quantidade de experiências realizadas, os resultados são bastante demorados a obter. No Capítulo 5 são dadas a conhecer as várias experiências realizadas. Tendo como exemplo a utilização do AG, com uma população de tamanho 150, com um número de gerações máximo de 100 e 30 *seeds* diferentes. Neste cenário, teremos ao longo da execução um total de 150 * 100 * 30 = 450'000 indivíduos. Tendo em conta que cada indivíduo da versão 2 tem cinco genes, tal como observámos na Secção 4.2.5, isto reflete-se em cinco vetores para uma *query*. Portanto, teremos ao longo da execução do programa 450'000 * 5 = 2'250'000 genes ou vetores de ataques diferentes. Cada vetor é testado para cada *query* de um projeto. Se tivermos em conta o projeto *Bricks*, o projeto de menores dimensões estudado, estamos perante 12 *queries* identificadas.

Dadas estas condições, serão efetuadas 2’250’000 * 12 = 27’000’000 *queries* à base de dados para realizar uma experiência completa no projeto *Bricks*. O tempo de execução de uma *query* depende de muitas variáveis: máquina, complexidade da *query*, quantidade de dados, esquema da base dados, SGBD, distância, entre outros. Após uma análise de uma *query* simples em termos de *Execution plan* concluiu-se que, na máquina onde foram realizados os testes, o tempo médio para a execução de uma *query* era de 130ms. Partindo do número de *queries* que é necessário realizar, é expectável ser necessário 27’000’000 * 130ms = 3’510’000’000 ms, equivalente a pouco mais de 40 dias para uma execução completa.

Tendo em conta esta realidade, procedeu-se a um estudo com o objetivo de melhorar o tempo total de execução do algoritmo. A introdução de um sistema de *caching* revelou ser uma solução com um esforço de implementação baixo, mas com um impacto elevado no tempo total de execução. O sistema consiste em guardar em memória, o valor da execução de uma *query*. Não é necessário guardar o resultado obtido. Apenas é guardado se um dado vetor de ataque foi bem-sucedido ou não durante a injeção.

A Listagem 28 mostra uma porção de código responsável por armazenar em memória os valores obtidos. Tal como podemos observar, isto é feito através de um *HashMap*. A chave utilizada será a *query* e o valor será um *boolean*. O valor permite determinar se uma *query* é vulnerável, sendo *true* se for vulnerável, e *false*, caso contrário.

Em termos de complexidade temporal, esta estrutura de dados é conhecida por *O(1)* para obtenção de dados através de uma chave. Uma vez que a execução de uma *query* depende de vários fatores, tal como vimos anteriormente, em termos de tempo, a obtenção de um valor de um *hashmap* também depende de vários fatores, daí tipicamente ser utilizada a notação *Big O*. O valor *O(1)*, significa que independentemente do tamanho do *hashmap* a obtenção de um valor é constante em termos temporais. No entanto, procedeu-se a uma análise em milissegundos do tempo médio para obtenção de um valor no *hashmap*, na mesma máquina. Em média obteve-se o seguinte valor: $8 * 10^{-5}$ ms.

```
public abstract class Cache<K, T> {

        private final Map<K, T> store;

        protected Cache() {
                store = new HashMap<>();
        }

        public void put(K key, T element) {
                store.put(key, element);
        }

        public T get(K key) {
                return store.get(key);
        }

        public boolean contains(K key) {
                return store.containsKey(key);
        }
}
```

Listagem 28: Extrato de código para *Cache*

Tendo em conta o exemplo anterior, se todos os valores estivessem em *cache* desde o momento inicial, isto traduzir-se-ia num tempo total de 27'000'000 * $8 * 10^{-5}$ms = 2,16 segundos. Obviamente, os valores não se encontram todos em memória desde o momento inicial, mas de forma teórica, o algoritmo que para o conjunto de parâmetros apresentados demorava 37 dias, passou a demorar 2 segundos. Em termos práticos, a execução é reduzida de vários dias para apenas alguns minutos, sendo uma melhoria muito significativa.

Para a implementação é feito o uso de um *Singleton*. Um *Singleton* é um *Design Pattern* que permite ter apenas uma instância de determinada classe. Desta forma, é possível aceder sempre à mesma instância dessa classe, independentemente da zona do código.

A Listagem 29 mostra uma porção de código retirada da aplicação desenvolvida, para verificar se a *query* a ser executada existe em *cache* (linha 3). Caso exista, obtém-se o valor proveniente da *cache*. Caso não exista (linha 7), obtém-se o valor da base de dados real e coloca-se o valor em *cache*, para que da próxima vez que a mesma *query* seja executada, não seja preciso interagir com a base de dados.

```
boolean executed;
boolean isComingFromCache = false;
if (SQLInjectionCache.INSTANCE.contains(query)) {
        executed = SQLInjectionCache.INSTANCE.get(query);
        isComingFromCache = true;
} else {
        executed = databaseConnection.hasResults(query);
        SQLInjectionCache.INSTANCE.put(query, executed);
}
```

Listagem 29: Extrato de código para execução do sistema de *cache*

4.6 *INTERFACE GUI*

Para facilitar o uso da ferramenta foi desenvolvida uma *interface* gráfica. A Figura 22 mostra o painel principal desenvolvido. No canto superior esquerdo temos todas as opções que podem ser configuradas para o uso da aplicação, maioria das quais pertencem a parâmetros do AG. O campo *vulnerability*, visível no painel, apenas foi desenvolvido para SQL *Injection*, no entanto, é facilmente extensível a outros problemas como XSS. Nos campos seguintes, segue-se a configuração da base de dados para o projeto a ser analisado. No canto superior direito encontra-se um gráfico que mostra a evolução do *fitness* da população em função do número de gerações. Nos painéis inferiores são mostrados aspetos da melhor solução. Neste caso foi executada a versão 1 do algoritmo desenvolvido, daí os genes do melhor indivíduo serem vetores de ataque completos. Por fim, são mostradas as combinações de genes que obtiveram sucesso quando executadas, bem como o número total de vulnerabilidades que o melhor indivíduo conseguiu detetar.

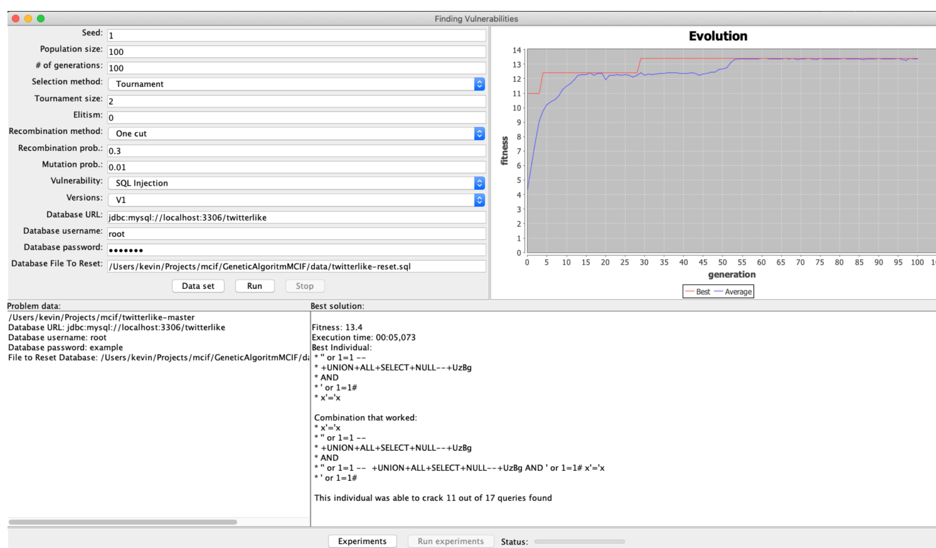

Figura 22: *Interface* gráfica da ferramenta desenvolvida

4.7 LINHA DE COMANDOS

Para além da *Graphical User Interface* (GUI) desenvolvida, foi ainda implementada a possibilidade de executar a ferramenta através da linha de comandos. Tal como irá ser descrito na Secção 4.9, a aplicação foi desenvolvida em *Java* e aceita dois parâmetros de entrada quando executada pela linha de comandos. O *Java Archive* (JAR) desenvolvido pode ser executado da seguinte forma:

```
java -jar <localização do jar> <ficheiro de parâmetros>
```

Listagem 30: Execução através da linha de comandos

4.8 CONFIGURAÇÕES

O ficheiro de parâmetros é responsável pela definição do algoritmo a usar, da versão, dos parâmetros necessários, bem como de todas as configurações necessárias para o projeto a analisar. A Listagem 31 mostra um exemplo de um ficheiro de configurações para o AG, usando a versão 1. Os campos obrigatórios e opcionais dependem do algoritmo em uso.

```
//----- Configurações gerais ---------------------------
runs: 30
algorithm: GA
version: V1

//----- Parâmetros GA --------------------------------
population_size: 150
elitism_size: 0
max_generations: 100
selection: tournament
tournament_size: 5
recombination: one_cut
recombination_probability: 0.3
mutation: string
mutation_probability: 0.5


//----- Parâmetros ABC -------------------------------
number_employed_bees: 50
number_onlooker_bees: 150
number_mutation: 2


//----- Parâmetros ACO -------------------------------
max_generations: 10
```

```
26 population_size: 80
27 number_modifications: 2
28 max_iter_without_improvement:  1
29 q: 100
30 q_probability: 0.3
31 pheromone_evaporation_rate: 0.1
32 pheromone_influence_rate: 0.5
33
34
35 //------ Configuração projeto --------------------------
36 dataset_name: bricks
37 problem_path: /Users/kevin/Projects/mcif/bricks
38 database_url: jdbc:mysql://localhost:3306/bricks
39 database_username: root
40 database_password: example
41 database_file_to_reset: /data/bricks-reset.sql
42
43 //-------- Estatística ------------------------
44 statistic: BestIndividual
45 statistic: BestAverage
46 statistic: Console
```

Listagem 31: Exemplo de um ficheiro de configurações

Na primeira secção, nas configurações gerais, podem ser definidos os seguintes parâmetros:

- ***runs*** define o número de *seeds* utilizado. Tem de ser um valor numérico positivo. No exemplo fornecido, o algoritmo é executado para as *seeds* de 1 a 30 inclusive, portanto é executado 30 vezes.
- ***algorithm*** define o algoritmo a ser usado: AG, ABC ou ACO.
- **versão** define a versão do algoritmo utilizada. Opções válidas são V1 e V2.

Para cada algoritmo é necessário preencher um número determinado de parâmetros, existindo para isso uma secção dedicada no ficheiro de configurações. Estes parâmetros foram apresentados nas Secções 4.2.1, 4.3.1 e 4.4.1.

A Secção *configuração projeto* é dedicada às configurações necessárias para analisar um projeto, sendo que:

- ***dataset_name*** define o nome do *dataset*. Este valor é um texto de qualquer dimensão e será usado para distinguir resultados produzidos.
- ***problem_path*** define o caminho completo até à localização do projeto alvo, a ser analisado.
- ***database_url*** define a ligação do projeto a ser analisado à sua respetiva base de dados.

- ***database_username*** define um utilizador válido capaz de executar comandos na base de dados. Idealmente, o utilizador deve ter acesso total à base de dados para que a análise decorra corretamente.
- ***database_password*** define a *password* do utilizador.
- ***database_file_to_reset*** define a localização do ficheiro auxiliar para repor a base de dados ao seu estado original. Este ficheiro é necessário quando um ataque de SQL *Injection* bem-sucedido altera o estado original da base de dados.

Por fim, a última parte, "Estatística" é dedicada ao *output* dos dados gerados. No exemplo apresentado na Listagem 31, é possível observar que serão produzidos dados relativos ao melhor indivíduo, bem como à média da população.

4.9 TECNOLOGIAS USADAS

Durante o desenvolvimento desta ferramenta, um aspeto fundamental tido em conta foi a exportabilidade. A ferramenta foi por esse motivo desenvolvida usando *Java*. O projeto começou com a versão 15, tendo sido migrada para a versão 16 posteriormente.

De forma a auxiliar a exportabilidade foi utilizada a ferramenta *Docker*, tal como mencionado no Capítulo 2.5.1. No contexto atual, isto permite-nos correr *containers* com a base de dados necessária, sem haver problemas de versões ou diferentes sistemas operativos. Permitiu ainda correr determinado *script* durante a criação dos *containers*. A Listagem 32 mostra o ficheiro necessário para criar uma base de dados e inicializá-la com os dados necessários. Neste caso, a linha 9 faz o mapeamento do ficheiro de inicialização da base de dados com a diretoria local designada de *data*. Todos os ficheiros nesta diretoria são executados.

```
1  version: '3.3'
2
3  services:
4    db:
5      image: mysql
6      command: --default-authentication-plugin=mysql_native_password
7      restart: always
8      volumes:
9      - ./data:/docker-entrypoint-initdb.d
10     environment:
11       MYSQL_ROOT_PASSWORD: example
12     ports:
13       - 3306:3306
```

```
14
15   adminer:
16     image: adminer
17     restart: always
18     ports:
19       - 8080:8080
```

Listagem 32: Ficheiro para criar e popular uma base de dados

A Listagem 33 mostra um exemplo do ficheiro (*script*) necessário para criar as tabelas do projeto *bricks* e para popular as mesmas com dados. O Apêndice B apresenta os ficheiros SQL para inicializar os restantes projetos.

```
1   create DATABASE bricks;
2
3   CREATE table bricks.users
4   (
5       idusers  INT           NOT NULL,
6       name     VARCHAR(45)  NOT NULL,
7       email    VARCHAR(45)  NOT NULL,
8       password VARCHAR(45)  NOT NULL,
9       ua       VARCHAR(45)  NOT NULL,
10      ref      VARCHAR(145) NOT NULL,
11      host     VARCHAR(45)  NOT NULL,
12      lang     VARCHAR(45)  NOT NULL,
13      PRIMARY KEY (idusers)
14  );
15
16  INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
17  VALUES (0, 'admin', 'admin@getmantra.com', 'admin', 'Brick_Browser',
    ↪  '$server$scriptpath/content-13/index.php',
18          '127.0.0.1', 'en');
19
20  INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
21  VALUES (1, 'tom', 'tom@getmantra.com', 'tom', 'Block_Browser', '', '8.8.8.8',
    ↪  'en');
22
23
24  INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
25  VALUES (2, 'harry', '22@getmantra.com', '5f4dcc3b5aa765d61d8327deb882cf99',
    ↪  'Mantra', '', '127.0.0.1', 'en');
26
27
28  INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
29  VALUES (3, 'harry', '33@getmantra.com', '5f4dcc3b5aa765d61d8327deb882cf99',
    ↪  'Mantra', '', '127.0.0.1', 'en');
30
31
32  INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
33  VALUES (4, 'harry', '44@getmantra.com', '5f4dcc3b5aa765d61d8327deb882cf99',
    ↪  'Mantra', '', '127.0.0.1', 'en');
```

Listagem 33: Ficheiro para criar e popular dados para o projeto *Bricks*

4.10 DIAGRAMA DA APLICAÇÃO

Para implementar a abordagem sugerida anteriormente, foi desenvolvida uma ferramenta em *Java*, como referido anteriormente. Para tal, foi utilizado um paradigma orientado a objetos e, foram implementadas várias classes para os vários processos dos algoritmos. A Figura 23 ilustra a hierarquia de classes construídas para os operadores genéticos.

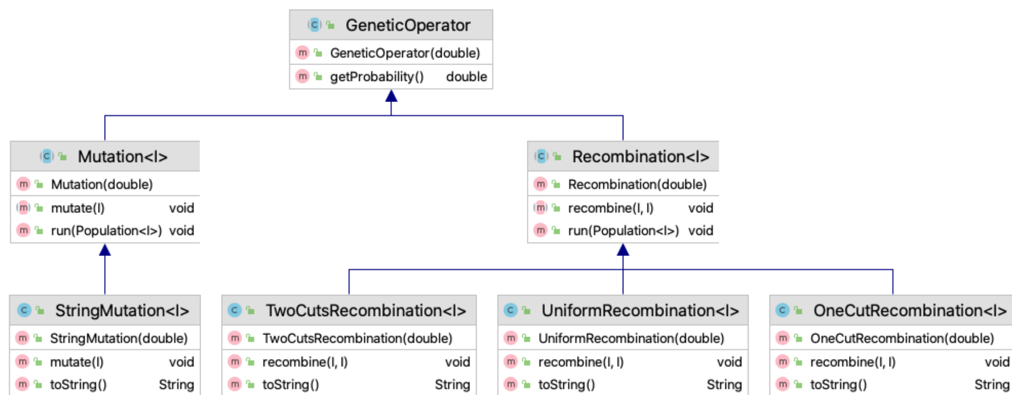

Figura 23: Classes definidas para os Operadores Genéticos

A Figura 24 apresenta o diagrama de classes relativo à pesquisa local, usada no algoritmo ABC.

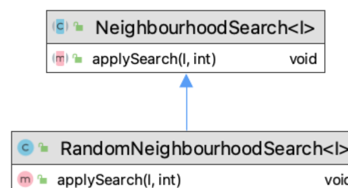

Figura 24: Classes utilizadas pelo algoritmo *Artificial Bee Colony* para a representação da pesquisa local

Na Figura 25 podemos observar a hierarquia de classes relativamente aos algoritmos implementados. Esta abordagem facilita a implementação de novos algoritmos de IA. Tal como podemos observar, cada algoritmo faz *override* do método *run* e *toString*. Estes métodos são específicos de cada algoritmo.

- ***addAlgorithmListener***: subscreve o algoritmo a um evento;
- ***fireGenerationEnded***: termina a execução de todos os eventos;
- ***fireRunEnded***: indica a todos os *listeners* que a execução de um algoritmo terminou;
- ***getAverageFitness***: devolve a média de *fitness* da população;
- ***getBestInRun***: devolve o melhor indivíduo de uma execução;
- ***getGenerationNumber***: devolve o número da geração atual;
- ***getSeedBestInRun***: devolve a *seed* que obteve melhores resultados;
- ***removeAlgorithmListener***: remove a subscrição de um algoritmo a um evento;
- ***run***: inicia a execução do algoritmo, sendo, portanto, diferente para cada algoritmo;
- ***stop***: termina a execução do algoritmo.

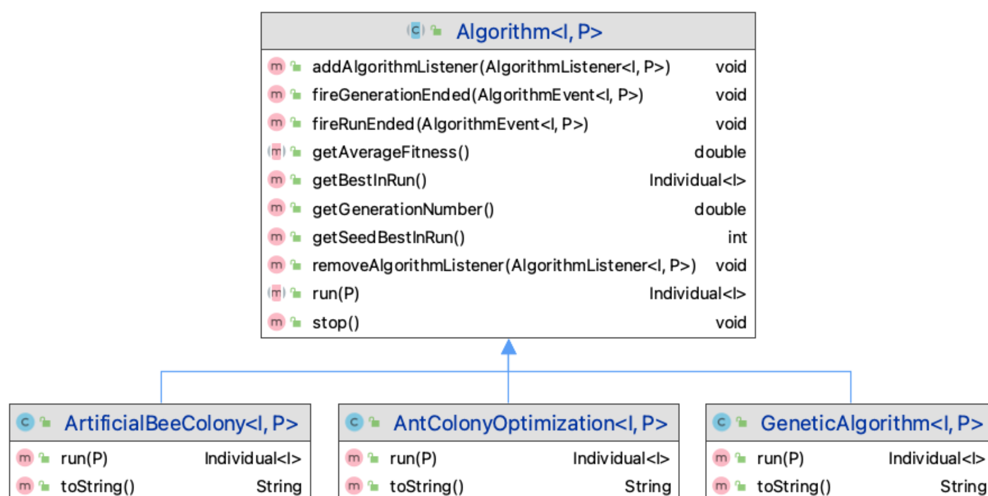

Figura 25: Classes definidas para a representação dos Algoritmos

Na Figura 26 podemos observar a hierarquia de classes relativa aos indivíduos implementados. Neste caso, ambas as versões (*SqlInjectionIndividualV1* e *SqlInjectionIndividualV2*) têm o propósito de detetar vulnerabilidades de SQL *Injection*, portanto, têm uma classe comum (*SqlInjectionIndividual*). Como estas versões têm um número de genes diferente (5 no caso do indivíduo versão 1, e 4 no indivíduo versão 2), o *getTotalGenesPossible()* é definido ao nível destas classes. O método *setGene(int)* tem como propósito alterar um gene de uma determinada posição.

Este método difere nestas duas versões porque o conjunto de valores que um gene pode assumir difere em cada versão.

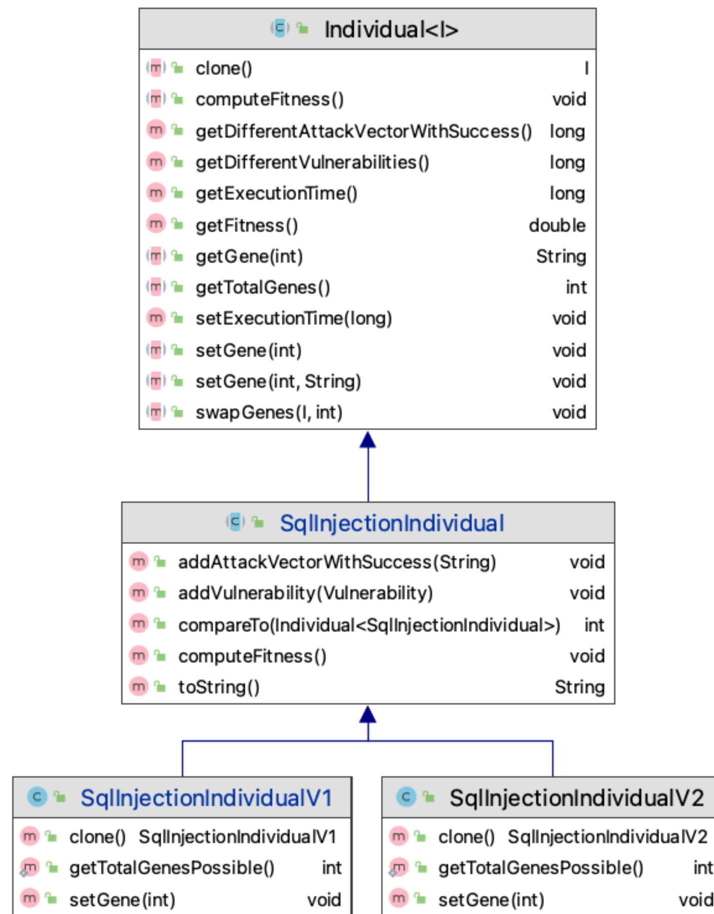

Figura 26: Classes definidas para a representação do indivíduo

Na Figura 27 podemos observar a hierarquia de classes do problema implementado. O método *score* é o método responsável por calcular o valor de *fitness* de um indivíduo. Este método é chamado pelo *computeFitness()* da classe *SqlInjectionIndividual*.

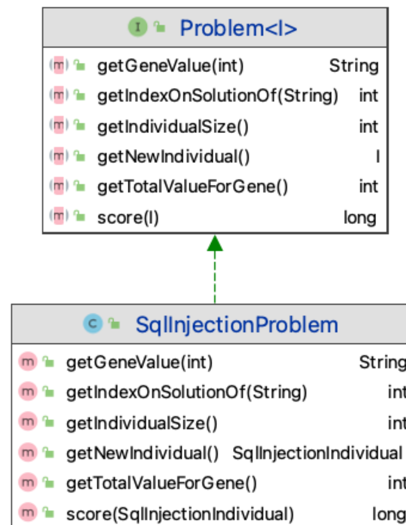

Figura 27: Classes definidas para a representação do problema

Na Figura 28 podemos observar as classes da população implementada. O método *evaluate()* é responsável por avaliar os indivíduos, chamando o método *computeFitness()* para cada um dos indivíduos da população.

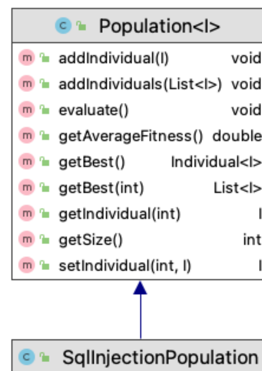

Figura 28: Classes definidas para a representação da população

5

EXPERIÊNCIAS E RESULTADOS

Neste capítulo serão apresentadas as experiências realizadas com a ferramenta desenvolvida em Java (descrita na Secção 4.7) e os resultados obtidos. Nas primeiras secções são apresentados os resultados e testes realizados para cada algoritmo em cada versão do indivíduo (da Secção 5.1 à 5.6). Posteriormente, na Secção 5.7 são apresentados os resultados obtidos de outras ferramentas, nomeadamente WAP (Secção 5.7.1) e *SonarPHP* (Secção 5.7.2). Por fim, na Secção 5.8 é mostrada uma comparação entre o melhor algoritmo e as ferramentas baseadas em análise estática.

As experiências foram realizadas com o intuito de responder às seguintes questões:

1. Que percentagem de vulnerabilidades é possível detetar?
   1.1. Para cada algoritmo, que conjunto de parâmetros apresenta melhores resultados?
   1.2. Para cada versão desenvolvida, que conjunto de parâmetros apresenta melhores resultados?
2. Que versão desenvolvida obteve melhores resultados?
3. Que algoritmo desenvolvido obteve melhores resultados?
4. Qual a eficácia desta abordagem relativamente a outras ferramentas na área em termos de vulnerabilidades identificadas?

Tal como mencionado na Secção 2.5, procedeu-se a uma deteção manual dos projetos analisados. A Tabela 4 resume o número de vulnerabilidades do tipo SQL *Injection* detetado na análise manual efetuada aos projetos.

Tabela 4: Número de vulnerabilidades identificadas manualmente

Projeto	Vulnerabilidades identificadas
Bricks	12
bWAPP	56
Twitterlike	17

Para responder à primeira pergunta (*Que percentagem de vulnerabilidades é possível detetar?*), é necessário identificar o número de vulnerabilidades detetadas pelos algoritmos desenvolvidos. Para isso, foi necessário desenvolver um conjunto de experiências para encontrar os parâmetros com melhores resultados. As secções seguintes apresentam os resultados obtidos.

Foram efetuadas 30 execuções para cada experiência (combinação de parâmetros), sendo que o valor do *fitness* diz respeito à média de valores do fitness com 30 *seeds* diferentes.

A exploração inicial do melhor conjunto de parâmetros foi realizada no projeto *Bricks*, sendo posteriormente aplicada aos restantes projetos de forma a avaliar a qualidade dos mesmos.

Todas as experiências desenvolvidas foram realizadas num *Raspberry PI 4*, modelo B, com 8GB de *Random Access Memory* (RAM) e *quad-core* 64*bits* de 1.5Ghz.

5.1 ALGORITMO GENÉTICO - VERSÃO 1

Tal como observámos na Secção 4.8, a implementação do AG requer a definição dos seguintes parâmetros:

- Tamanho da População;
- Elitismo;
- Máximo de gerações;
- Métodos de seleção;
- Tamanho do torneio;
- Métodos de recombinação;
- Probabilidade de recombinação;
- Probabilidade de mutação.

Tendo em conta estes parâmetros, foram realizados vários estudos comparativos para perceber que combinações trariam melhores resultados. É considerado um melhor resultado aquele que detetar mais vulnerabilidades, sendo que em caso de empate, o melhor resultado é aquele cujo o tempo de execução é menor.

5.1.1 *Influência do número de gerações*

Nesta secção pretendemos aferir qual o número de gerações mínimo, necessário para obter melhores valores, ou seja, a partir de quantas gerações a curva de aprendizagem estagna. Este passo é particularmente importante para não se desperdiçar recursos e tempo.

Para tal, começou-se com valores de geração de 5, 20, 50 e 100. O primeiro valor revelou ser insuficiente, pelo que a curva de aprendizagem continuava a crescer após este ponto. A Figura 29 mostra este efeito e, tal como podemos observar, após 50 gerações não obtivemos nenhum aumento significativo, pelo que para a versão 1 do algoritmo, foi considerado um valor máximo de 50 gerações. Não foram explorados números maiores de gerações, pois, nesta altura, já se tinham obtido soluções ótimas para o problema.

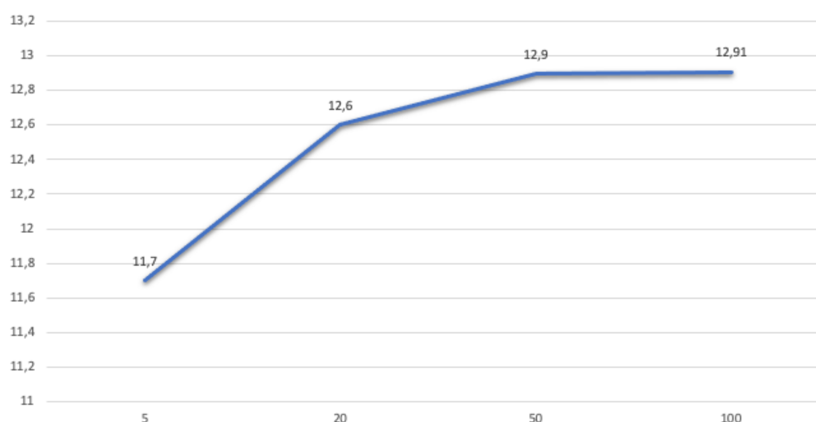

Figura 29: Efeito do número máximo de gerações no *fitness* - versão 1

5.1.2 *Influência do tamanho da população*

A Figura 30 apresenta os resultados obtidos durante a medição do *fitness*, quando são usados diferentes tamanhos para a população. À medida que a população aumenta, também o valor de *fitness* é melhor, sendo que a partir de 200 indivíduos esse acréscimo é bastante diminuto.

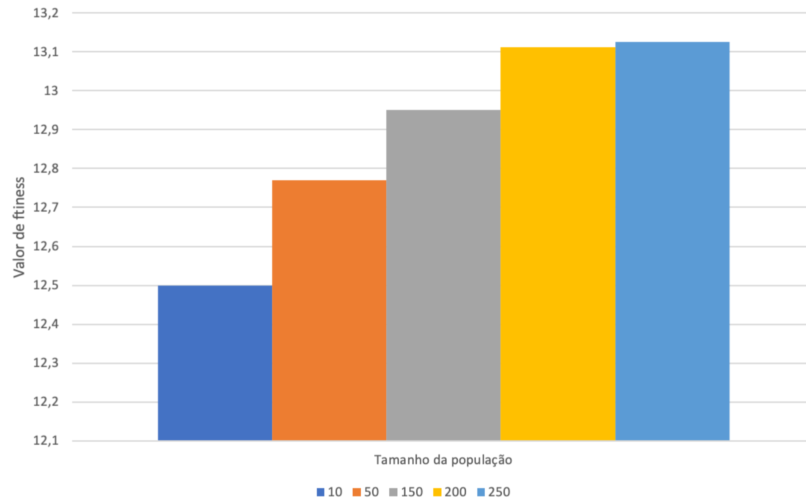

Figura 30: Efeito do tamanho da população no valor de *fitness* - versão 1

Contudo, o aumento da população envolve também o incremento do tempo de execução, pelo que é necessário verificar se o tempo que envolve avaliar uma população de dimensão superior, compensa o aumento em *fitness* subjacente. Para isso, são avaliadas de seguida as várias populações para um tempo máximo de execução, em vez de, um número máximo de gerações.

A Figura 31 mostra o efeito do tamanho da população, mas com um tempo de execução máximo de 30 segundos. Nesta situação é possível observar um decréscimo no valor de *fitness* para populações superiores a 250. Tendo em conta estes resultados, foi considerada uma população de 200 indivíduos para a análise.

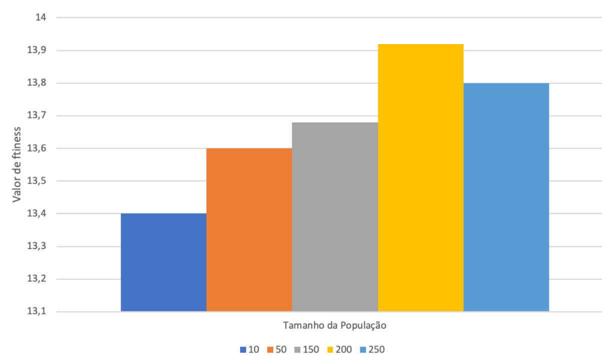

Figura 31: Influência do tamanho da população no valor de *fitness* com limite de 30 segundos - versão 1

5.1.3 *Influência do Elitismo*

Quando avaliado o impacto do Elitismo, na versão 1, do AG, os resultados apresentados na Tabela 5, sugerem que o seu impacto no valor de *fitness* é bastante reduzido. Sendo que com o valor 0, ou seja, não usando elitismo, os valores de *fitness* tendem a ser ligeiramente superiores.

Tabela 5: Efeito do elitismo no valor de *fitness* - versão 1

ELITISMO	MÉDIA DO *FITNESS*
0	12.68
2	12.62
4	12.65
6	12.63
8	12.67

5.1.4 *Influência dos métodos de seleção*

Relativamente aos métodos de seleção estudados, tal como podemos observar pela Figura 32, o claro vencedor é o método de seleção por torneio. O próximo passo é analisar qual o tamanho que obtém melhores valores.

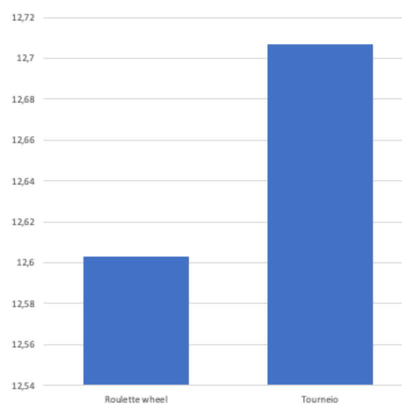

Figura 32: Influência de métodos de seleção no valor de *fitness* - versão 1

5.1.5 *Influência do tamanho do torneio*

A Figura 33 sintetiza os valores obtidos quando o AG é submetido a diferentes tamanhos de torneio. Tal como se pode observar, não existe uma relação direta entre o tamanho do torneio e o *fitness*, contudo, o torneio com tamanho 6 revela ter ligeiras melhorias quando comparado com os restantes.

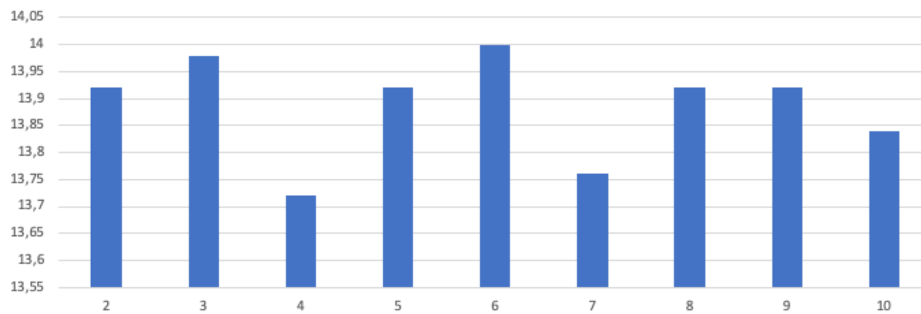

Figura 33: Influência do tamanho do torneio no valor de *fitness* - versão 1

5.1.6 *Influência dos métodos de recombinação*

A Figura 34 mostra os resultados obtidos quando comparando os vários métodos de recombinação utilizados. Tal como é possível observar, o método de um corte (*one cut*) foi o que obteve melhores resultados.

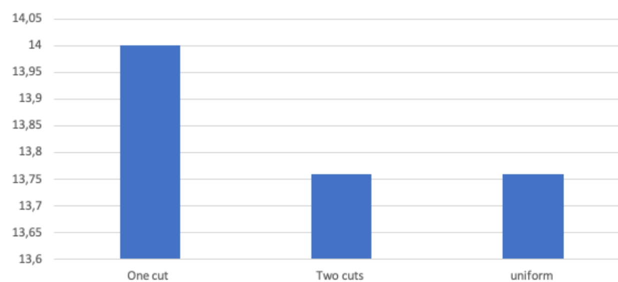

Figura 34: Influência dos métodos de recombinação no valor de *fitness* - versão 1

5.1.7 *Influência das probabilidades de recombinação e mutação*

Na Figura 35 pode ser observado um gráfico comparativo entre as probabilidades de mutação e recombinação e o seu efeito no *fitness*. No eixo do x temos as probabilidades e no eixo do y temos o valor de *fitness*. É possível constatar que, para

a mutação, valores de probabilidade altos têm um impacto bastante negativo na qualidade da solução, sendo que se considerou o melhor valor: 0.05. Por outro lado, o impacto da probabilidade de recombinação tem um efeito oposto, encontrando-se melhores valores quando são utilizados maiores valores de probabilidade, sendo o melhor valor: 0.5.

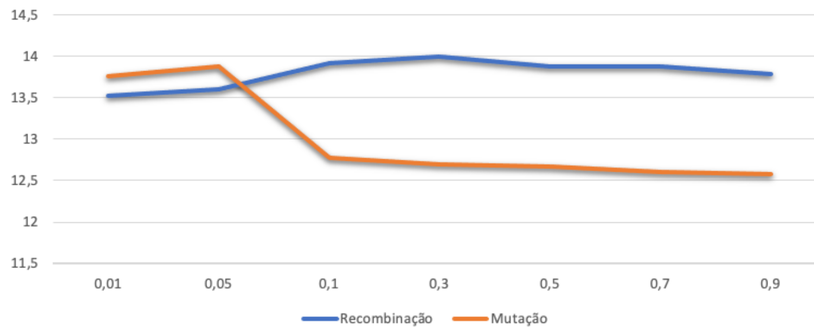

Figura 35: Influência das probabilidades de mutação e recombinação no valor de *fitness* - versão 1

Podemos ainda concluir que a probabilidade de mutação tem uma maior influência no valor do *fitness* do que a probabilidade de recombinação, pois a amplitude dos valores obtidos é maior na mutação.

5.1.8 *Melhores parâmetros e resultados*

A Tabela 6 sintetiza o conjunto de parâmetros que obteve melhores resultados para o AG, versão 1, desenvolvida.

Tabela 6: Parâmetros que obtiveram melhores resultados - versão 1

PARÂMETRO	VALOR
Tamanho da população	200
Número de gerações	50
Método de seleção	Torneio (Tamanho = 6)
Elitismo	0 (sem elitismo)
Recombinação	*One cut* (50%)
Probabilidade de mutação	5%

A Tabela 7 mostra os resultados obtidos pelo melhor individuo do AG versão 1, com os parâmetros indicados anteriormente, assim como a sua eficácia.

Tabela 7: Vulnerabilidades detetadas pelo AG - versão 1

Projeto	Vulnerabilidades	Eficácia
Bricks	12	100%
bWAPP	52	92.86%
Twitterlike	13	76.47%

5.2 ALGORITMO GENÉTICO - VERSÃO 2

De seguida, são apresentados os resultados inerentes à segunda versão modelada para um indivíduo, utilizando o AG. Os parâmetros estudados são os mesmos que os utilizados na versão 1 (Secção 5.1).

5.2.1 *Influência do número de gerações*

A Figura 36 demonstra a evolução da população ao longo de várias gerações. Apesar da obtenção de melhoramento contínuo, considerou-se que, a partir da geração 20, não existe um melhoramento significativo, pelo que os testes foram executados para um número máximo de 20 gerações.

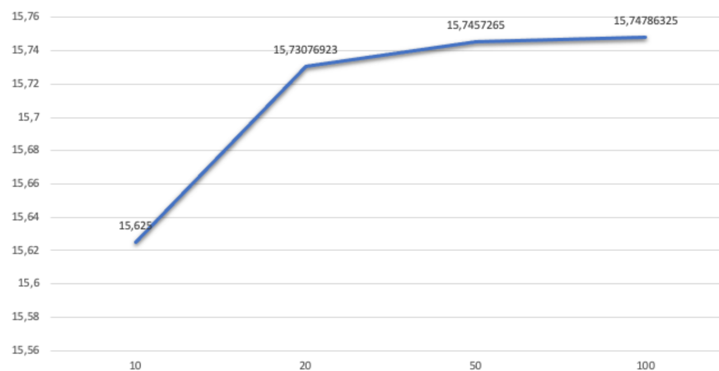

Figura 36: Efeito do número máximo de gerações no *fitness* - versão 2

5.2.2 *Influência do tamanho da população*

Relativamente à influência do tamanho da população para a versão 2 do AG, a Figura 37 demonstra o efeito desta, no valor de *fitness*. Ao contrário dos resultados

obtidos na versão 1, nesta versão não existe nenhuma melhoria quando o tamanho de população supera valores de 50 indivíduos.

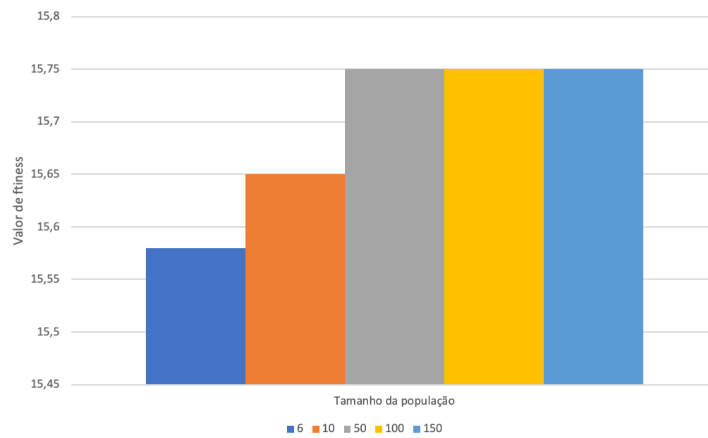

Figura 37: Efeito do tamanho da população no valor de *fitness* - versão 2

5.2.3 *Influência do elitismo*

Enquanto que na versão 1, o impacto do elitismo é reduzido, obtendo-se melhores valores de *fitness* quando não há elitismo, na versão 2, esta tendência inverteu-se. A Figura 38 demonstra que valores de elitismo superiores garantem uma melhor solução. Para esta situação escolheu-se um valor de 4, ou seja, garantimos que 4 dos melhores indivíduos de uma geração passam à geração seguinte.

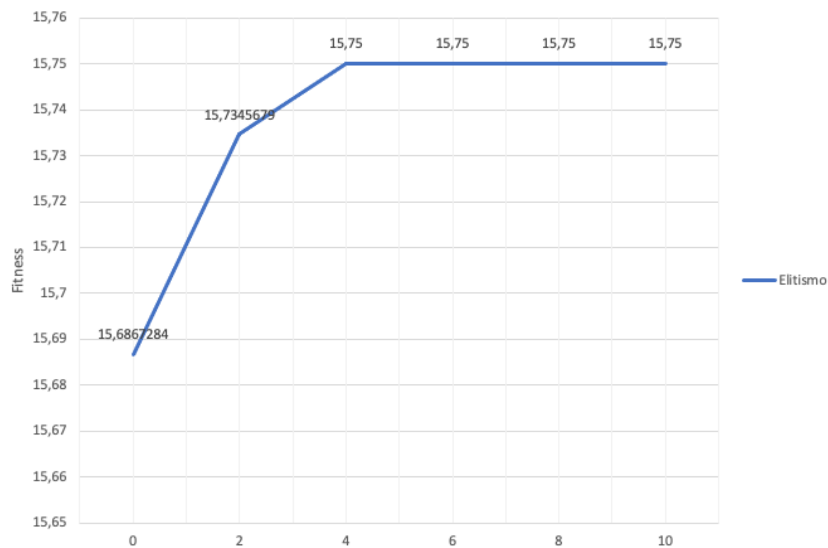

Figura 38: Efeito do elitismo no valor de *fitness* - versão 2

5.2.4 *Influência dos métodos de seleção e tamanho do torneio*

Os dados obtidos relativamente à influência dos métodos de seleção e tamanho do torneio mostram que estes parâmetros não têm influência no valor de *fitness*. Portanto, será usado o método: *Roulette Wheel*.

5.2.5 *Influência dos métodos de recombinação*

Da análise comparativa dos métodos de recombinação utilizados (um corte, dois cortes e uniforme) não existe uma forte distinção de resultados entre os métodos.

5.2.6 *Influência das probabilidades de recombinação e mutação*

Tal como observado, na versão 1, a mutação tem um efeito bastante penalizador quando testado com valores de probabilidade elevados, tendo obtido valores ideias para percentagens inferiores a 0.1.

Relativamente à recombinação, existe uma tendência oposta àquilo que foi observado na versão 1. Na versão 2, tal como podemos constatar pela Figura 39, existe um ligeiro decréscimo no valor do *fitness* para valores elevados.

Desta forma, consideramos 0.05 para o valor de mutação e 0.1 para o valor de recombinação.

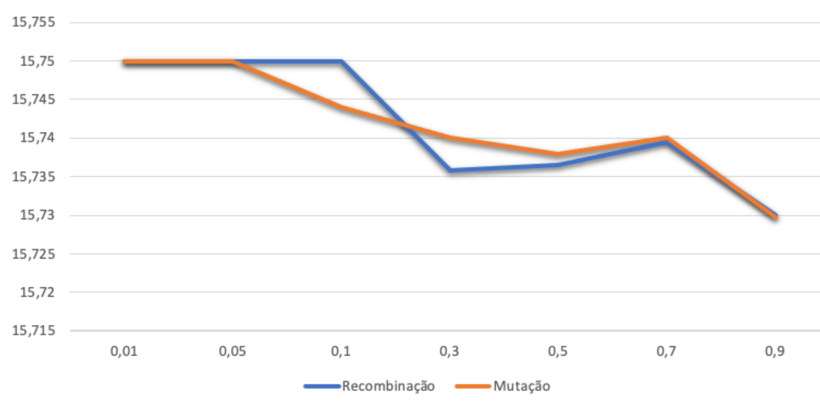

Figura 39: Influência das probabilidades de mutação e recombinação no valor de *fitness* - versão 2

5.2.7 *Melhores parâmetros e resultados*

Os melhores parâmetros para o AG, versão 2, podem ser encontrados na Tabela 8.

Tabela 8: Parâmetros que obtiveram melhores resultados - versão 2

PARÂMETRO	VALOR
Tamanho da População	50
Número de gerações	20
Método de seleção	*Roulette Wheel*
Elitismo	4 indivíduos
Recombinação	*One cut* (10%)
Probabilidade de mutação	5%

A Tabela 9 mostra os resultados obtidos pelo melhor indivíduo do AG, versão 2, com os parâmetros indicados anteriormente, assim como a sua eficácia.

Tabela 9: Vulnerabilidades detetadas pelo AG - versão 2

Projeto	Vulnerabilidades	Eficácia
Bricks	11	91.67%
bWAPP	47	83.93%
Twitterlike	10	58.82%

5.3 *ARTIFICIAL BEE COLONY* - VERSÃO 1

Nesta secção são apresentados os resultados obtidos quando analisado o algoritmo ABC, bem como o estudo realizado para obter a melhor combinação de parâmetros.

Tal como observámos na Secção 4.8, a implementação do ABC utiliza os seguintes parâmetros:

- Número de abelhas empregadas;
- Número de abelhas espetadoras;
- Número de mutações.

Tal como no algoritmo AG, foram realizados vários estudos comparativos para perceber que combinações permitem obter melhores resultados. Os resultados são apresentados de seguida.

5.3.1 *Influência do número máximo de iterações*

A Figura 40 mostra a influência que o número de iterações tem no valor de *fitness*. Tal como seria esperado, existe uma proporcionalidade direta entre ambos os fatores, ou seja, quanto maior o número de iterações maior o valor de *fitness*. No entanto, isto implica necessariamente que o tempo de execução também seja maior, portanto, é necessário avaliar a partir de que momento o crescimento é nulo ou insignificante. Tal pode ser constatado a partir de 100 iterações.

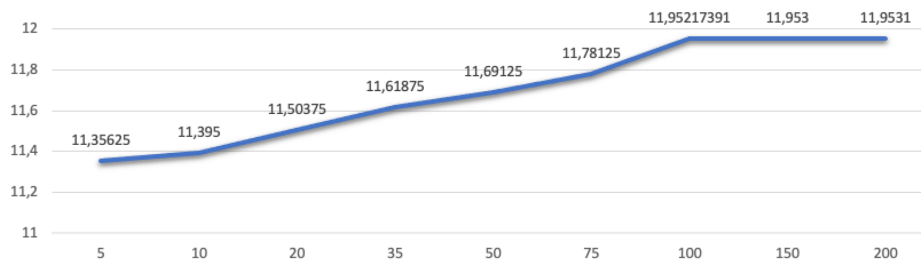

Figura 40: Influência do número máximo de iterações no valor de *fitness* - versão 1

5.3.2 *Influência do número de abelhas empregadas*

A Figura 41 apresenta os resultados obtidos durante a medição do *fitness* quando usados diferentes números de abelhas empregadas. À medida que a quantidade destas abelhas aumenta, também o valor de *fitness* é melhor, no entanto, após 75 abelhas empregadas o valor decresceu.

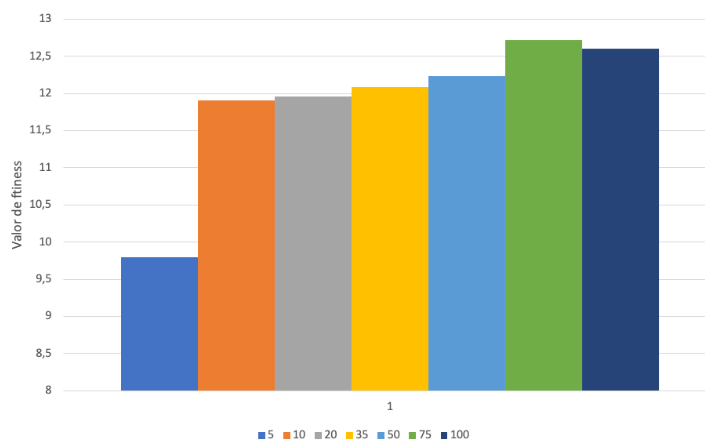

Figura 41: Efeito do número de abelhas empregadas no valor de *fitness* - versão 1

Contudo, o aumento do número de abelhas, tem como consequência o aumento do tempo de execução. Desta forma, é necessário verificar se o tempo que envolve avaliar uma colmeia de dimensão superior compensa o aumento de *fitness* subjacente. Para isso, são avaliadas, de seguida, colmeias de abelhas empregadas de várias dimensões, para um tempo máximo de execução.

A Figura 42 mostra o efeito do tamanho da população, mas com um tempo de execução máximo de 30 segundos. Nesta situação é possível observar um gráfico muito semelhante ao da Figura 41. Podemos concluir, que com 75 abelhas empregadas obtém-se o melhor valor de *fitness*.

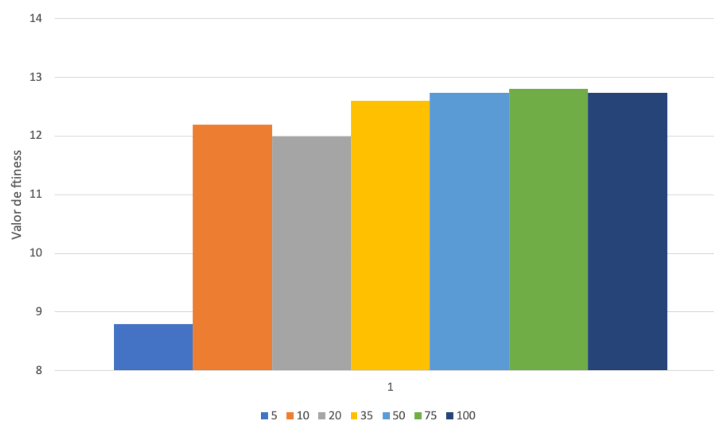

Figura 42: Influência do número de abelhas empregadas no valor de *fitness* com limite de 30 segundos - versão 1

5.3.3 *Influência do número de abelhas espetadoras*

A Figura 43 demonstra os resultados obtidos quando várias abelhas espetadoras são comparadas relativamente ao seu *fitness*. Tal como podemos observar, o valor de 100 abelhas espetadores resultou num melhor valor de *fitness*.

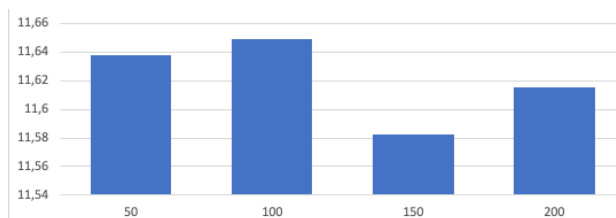

Figura 43: Influência do número de abelhas espetadoras no valor de *fitness* - versão 1

5.3.4 *Influência do número de modificações*

A Figura 44 mostra o efeito do número de modificações no valor de *fitness*. Tal como podemos constatar, não existe uma correlação direta entre o número de modificações e o *fitness*, sendo que a curva gerada é bastante inconstante e irregular. Temos, contudo, um valor ótimo, quando o número de modificações é 11.

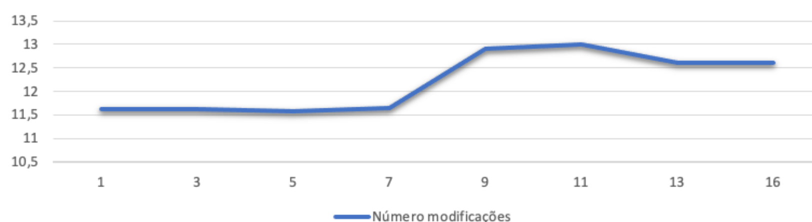

Figura 44: Influência do número de modificações no valor de *fitness* - versão 1

5.3.5 *Melhores parâmetros e resultados*

A Tabela 10 sintetiza os parâmetros do algoritmo ABC que obtiveram melhores resultados, na versão 1.

Tabela 10: Parâmetros do algoritmo ABC que obtiveram melhores resultados - versão 1

PARÂMETRO	VALOR
Número de abelhas empregadas	75
Número de abelhas espetadoras	100
Número de iterações	100
Número de modificações	11

A Tabela 11 mostra os resultados obtidos pelo melhor indivíduo do ABC, versão 1, com os parâmetros indicados anteriormente, assim como a sua eficácia.

Tabela 11: Vulnerabilidades detetadas pelo algoritmo ABC - versão 1

Projeto	Vulnerabilidades	Eficácia
Bricks	12	100%
bWAPP	33	58.93%
Twitterlike	13	76.47%

5.4 *ARTIFICIAL BEE COLONY* - VERSÃO 2

Nesta secção são apresentados os resultados das várias combinações de parâmetros para a versão 2 do algoritmo ABC. O estudo efetuado teve em conta os mesmos parâmetros apresentados na secção anterior (Secção 5.3).

5.4.1 *Influência do número máximo de iterações*

Relativamente ao efeito do número de iterações, verificou-se um aumento progressivo no valor de *fitness* até 200 iterações, tal como pode ser evidenciado pela Figura 45. A partir deste valor, o valor do *fitness* manteve-se estagnado.

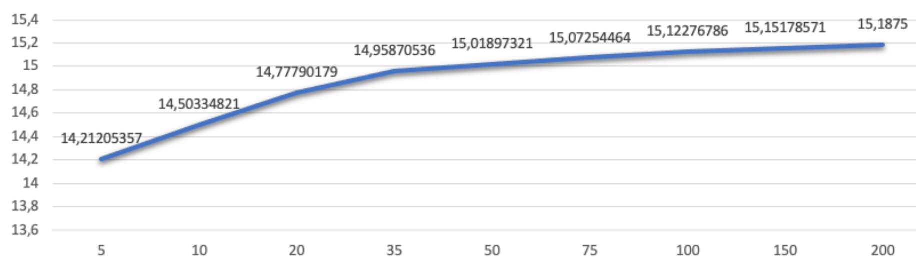

Figura 45: Influência do número máximo de iterações no valor de *fitness* - versão 2

5.4.2 *Influência do número de abelhas empregadas*

A Figura 46 mostra o efeito que o número de abelhas empregadas teve na versão 2 do algoritmo ABC. Tal como se pode constatar, a partir de 35 abelhas, o *fitness* não sofre muitas alterações, sendo que 75 abelhas é o número que oferece um melhor desempenho, tal como sucedeu com a versão 1 do mesmo algoritmo.

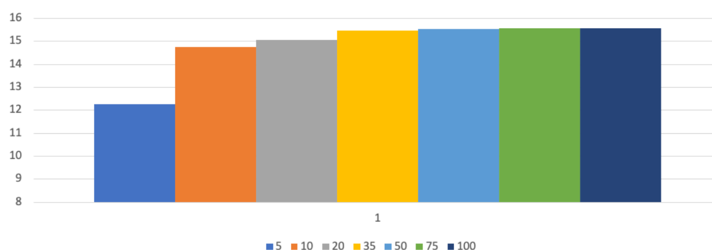

Figura 46: Efeito do número de abelhas empregadas no valor de *fitness* - versão 2

5.4.3 *Influência do número de abelhas espetadoras*

A Figura 47 mostra a influência do número de abelhas espetadoras no valor de *fitness*. Nesta situação 100 abelhas foi o melhor resultado.

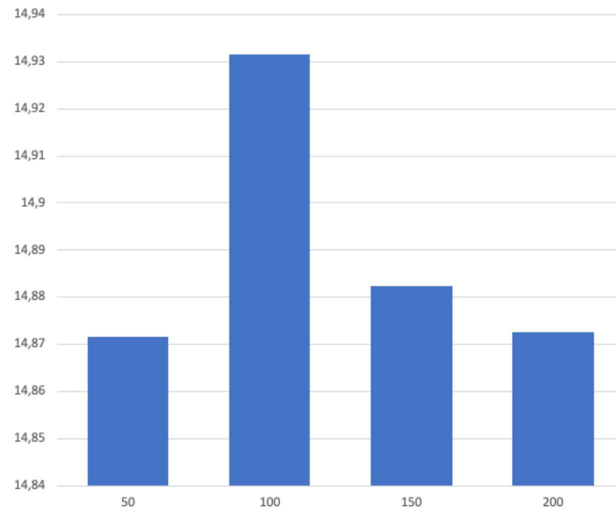

Figura 47: Influência do número de abelhas espetadoras no valor de *fitness* - versão 2

5.4.4 *Influência do número de modificações*

A Figura 48 demonstra a variação do *fitness* com base em diferentes modificações. Ao contrário do que foi verificado na versão 1, apenas com uma modificação obtiveram-se melhores valores.

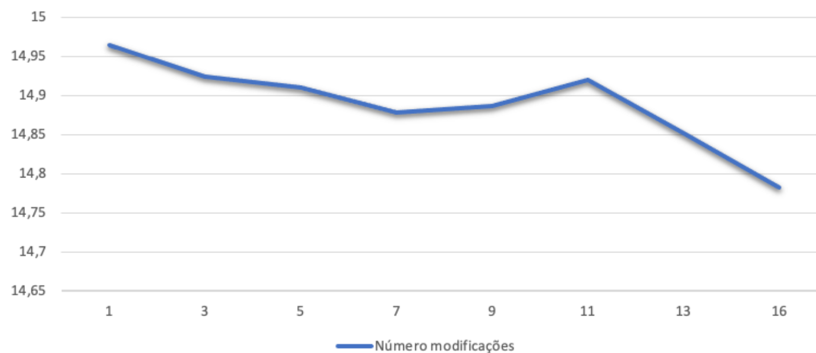

Figura 48: Influência do número de modificações no valor de *fitness* - versão 2

5.4.5 *Melhores parâmetros e resultados*

A Tabela 12 sintetiza os valores dos parâmetros do algoritmo ABC que obtiveram melhores resultados, na versão 2.

Tabela 12: Parâmetros do algoritmo ABC que obtiveram melhores resultados - versão 2

PARÂMETRO	VALOR
Número de abelhas empregadas	75
Número de abelhas espetadoras	100
Número de iterações	200
Número de modificações	1

A Tabela 13 mostra os resultados obtidos pelo melhor indivíduo do ABC, versão 2, com os parâmetros indicados anteriormente, assim como a sua eficácia.

Tabela 13: Vulnerabilidades detetadas pelo algoritmo ABC - versão 2

Projeto	Vulnerabilidades	Eficácia
Bricks	11	91.67%
bWAPP	47	83.93%
Twitterlike	10	58.82%

5.5 *ANT COLONY OPTIMIZATION* - VERSÃO 1

Tal como observámos na Secção 4.8, a implementação do ACO implica a definição dos seguintes parâmetros:

- Tamanho da colónia;
- Máximo de iterações;
- Número de modificações;
- Número de iterações sem melhorias;
- Parâmetro Q;
- Probabilidade de q;
- Probabilidade de evaporação de feromonas;
- Influência de feromonas.

Tal como nos restantes algoritmos, foram realizados vários estudos comparativos para perceber que combinações permitem obter melhores resultados. São apresentados, de seguida, os resultados obtidos.

5.5.1 *Influência do número máximo de iterações*

O objetivo é encontrar o valor mínimo de iterações que obtém o maior valor de *fitness*. Para isso, é necessário identificar a partir de que momento não existem melhorias, apesar do incremento de iterações. A Figura 49 mostra exatamente esses dados. Após 30 iterações não se verifica um aumento no valor de *fitness*, em parte porque a partir das 30 iterações já se encontrou a solução ideal.

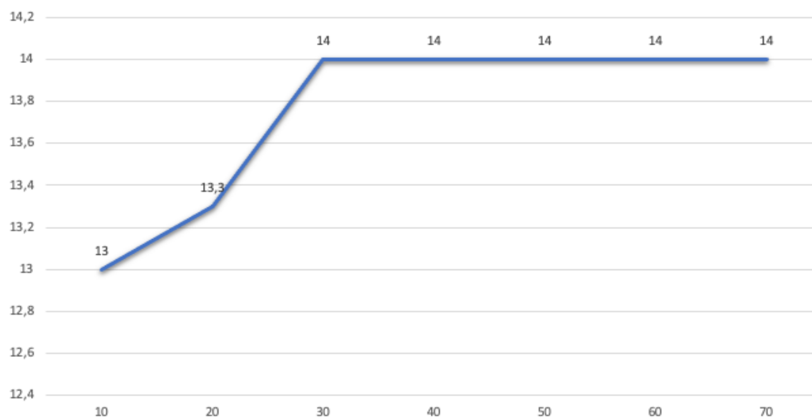

Figura 49: Efeito do número de iterações no valor de *fitness* - versão 1

5.5.2 *Influência do número de iterações máximo sem melhorias*

Este parâmetro, tal como vimos anteriormente, indica quantas iterações podem existir, sem melhorias. A Figura 50 evidencia que o número 5 de iterações obtém os melhores valores de *fitness*.

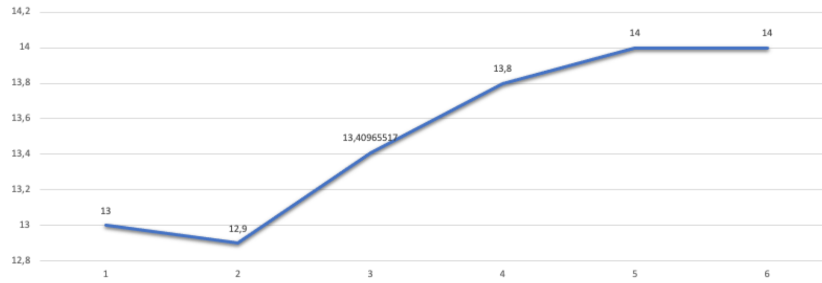

Figura 50: Efeito do número de iterações sem melhorias no valor de *fitness* - versão 1

5.5.3 *Influência do tamanho da colónia*

A Figura 30 apresenta os resultados obtidos durante a medição do *fitness* quando submetido a colónias de formigas de diferentes dimensões. É possível observar um aumento no valor do *fitness* até uma população máxima de 80 formigas, sendo que após este valor, começa-se a notar um declínio na qualidade da solução. É importante realçar a proporcionalidade direta entre o número de formigas e o tempo de execução necessário, ou seja, quanto maior a colónia, maior é o tempo de espera.

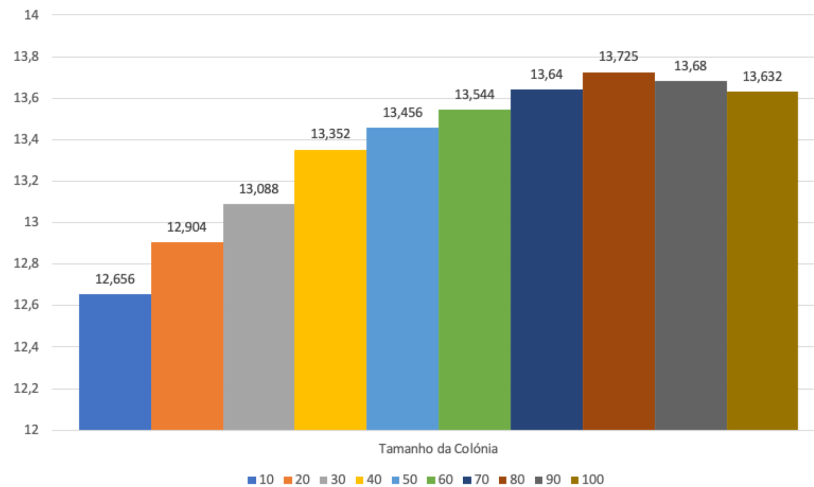

Figura 51: Efeito do tamanho da colónia no valor de *fitness* - versão 1

5.5.4 *Influência dos parâmetros de probabilidades*

A Figura 52 mostra o efeito das várias probabilidades utilizadas como parâmetros de entrada para este algoritmo. No caso da probabilidade q, nota-se um declínio no valor do *fitness* com maiores valores percentuais. Nesta situação o valor 0.1 foi o

que obteve melhores valores. No caso da probabilidade de evaporação da feromona, as oscilações de *fitness* são bastante baixas, sendo que foi considerado o valor de 0.3. Por fim, o valor de influência de feromona revelou ser aquele que obtém maiores diferenças nos valores de fitness, por probabilidade, sendo que o melhor valor esteve nos 0.3.

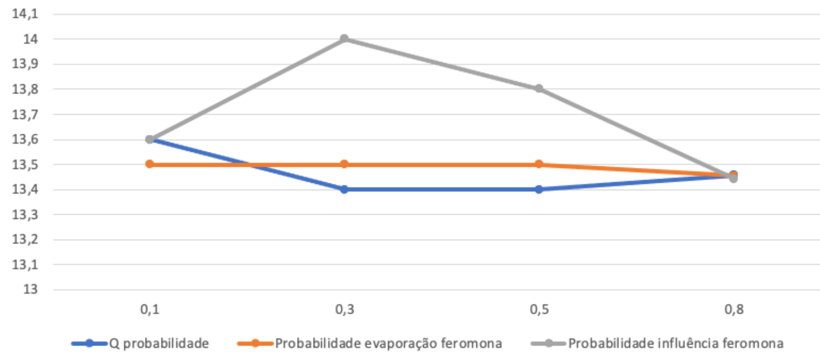

Figura 52: Efeito das várias probabilidades no valor de *fitness* - versão 1

5.5.5 *Melhores parâmetros e resultados*

A Tabela 14 sintetiza os parâmetros para o algoritmo ACO que obtiveram melhores resultados, na versão 1.

Tabela 14: Parâmetros do algoritmo ACO que obtiveram melhores resultados - versão 1

PARÂMETRO	VALOR
Número de iterações	30
Tamanho da Colónia	80
Número de modificações	2
Número de iterações sem melhorias	5
Q	100
Probabilidade (q)	0,1
Probabilidade evaporação feromona	0,3
Probabilidade influência feromona	0,3

A Tabela 15 mostra os resultados obtidos pelo melhor indivíduo do ACO na versão 1, com os parâmetros indicados anteriormente, assim como a sua eficácia, quando executado para as 30 *seeds* mencionadas anteriormente.

Tabela 15: Vulnerabilidades detetadas pelo algoritmo ACO - versão 1

Projeto	Vulnerabilidades	Eficácia
Bricks	12	100%
bWAPP	33	58.93%
Twitterlike	13	76.47%

5.6 ANT COLONY OPTIMIZATION - VERSÃO 2

5.6.1 *Influência do número de iterações máximo sem melhorias*

Na Figura 53 pode-se verificar que o número 1 de iterações sem melhorias obtém os melhores valores de *fitness*.

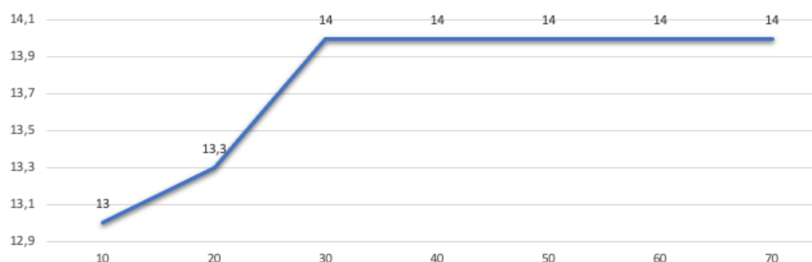

Figura 53: Efeito do número de iterações sem melhorias no valor de *fitness* - versão 2

5.6.2 *Influência do tamanho da colónia*

A Figura 54 apresenta os resultados obtidos durante a medição do *fitness* quando submetido a colónias de formigas de diferentes dimensões. É possível observar um comportamento similar ao observado na versão 1, onde, o valor do *fitness* é máximo para 80 formigas, sendo que após este valor, começa-se a notar um declínio na qualidade da solução.

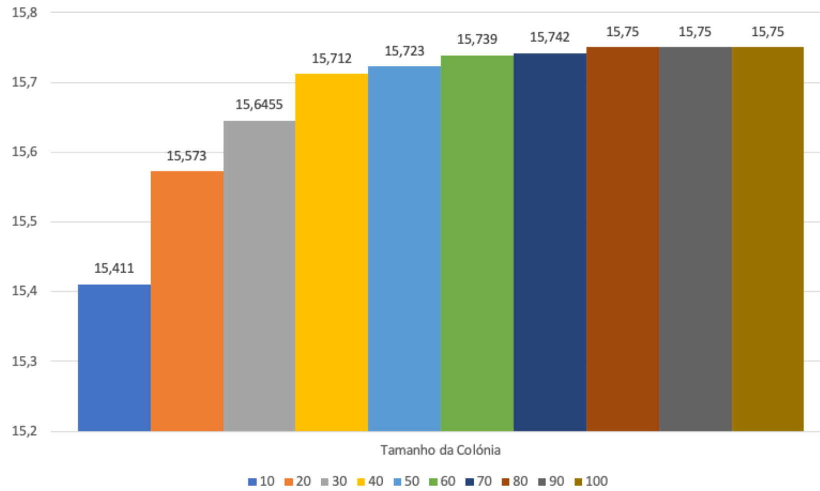

Figura 54: Efeito do tamanho da colónia no valor de *fitness* - versão 2

5.6.3 *Influência dos parâmetros de probabilidades*

A Figura 55 mostra o efeito no valor do fitness das várias probabilidades (parâmetros de entrada) do algoritmo ACO. No caso da probabilidade q, nota-se um declínio no valor do *fitness* a partir de 0.3. No caso da probabilidade de evaporação da feromona, as oscilações de *fitness* são bastante insignificantes, sendo que foi considerado o valor de 0.1. Por fim, o valor de influência de feromona, ao contrário do que aconteceu na versão 1, revelou ter também um impacto superior na qualidade do *fitness*, tendo sido considerado o valor de 0.5.

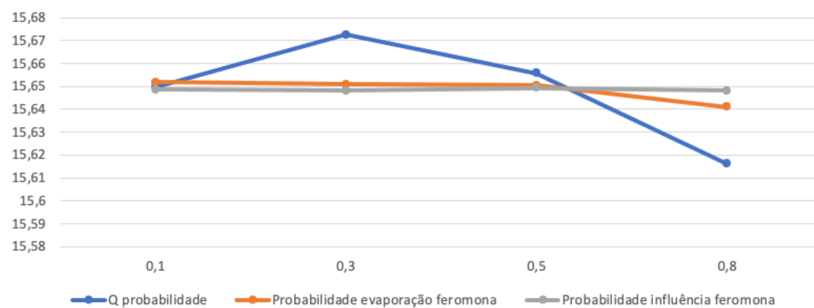

Figura 55: Efeito do número de iterações sem melhorias no valor de *fitness* - versão 2

5.6.4 *Melhores parâmetros e resultados*

A Tabela 16 sintetiza os parâmetros para o algoritmo ACO que obtiveram melhores resultados, na versão 2.

Tabela 16: Parâmetros do algoritmo ACO que obtiveram melhores resultados - versão 2

PARÂMETRO	VALOR
Número de iterações	10
Tamanho da Colónia	80
Número de modificações	2
Número de iterações sem melhorias	1
Q	100
Probabilidade (Q)	0,3
Probabilidade evaporação feromona	0,1
Probabilidade influência feromona	0,5

A Tabela 17 mostra os resultados obtidos pelo melhor indivíduo do ACO na versão 2.

Tabela 17: Vulnerabilidades detetadas pelo algoritmo ACO - versão 2

Projeto	Vulnerabilidades	Eficácia
Bricks	11	91.67%
bWAPP	51	91.07%
Twitterlike	10	58.82%

5.7 RESULTADO DE OUTRAS FERRAMENTAS

Na Secção 2.6 foram apresentadas várias ferramentas de análise estática. Nesta secção é apresentado o número de vulnerabilidades detetadas nos projetos analisados com essas ferramentas.

5.7.1 *Web Application Protection*

O WAP é uma ferramenta de análise estática que pode ser executada através da linha de comandos. Para o efeito desejado, isto é, para detetar vulnerabilidades do

tipo SQL *Injection*, a ferramenta pode ser executada da seguinte forma: *java -jar wap.jar -p <diretoria do projeto> -sqli*

A Figura 56 mostra o número de vulnerabilidades detetadas pela ferramenta WAP, quando executada no projeto *Twitterlike*.

Figura 56: Resultados da análise do *Twitterlike* com a ferramenta WAP

A Tabela 18 mostra as vulnerabilidades detetadas pelo WAP em cada projeto analisado.

Tabela 18: Vulnerabilidades detetadas pelo *Web Application Protection*

Projeto	Vulnerabilidades	Eficácia
Bricks	11	91.67%
bWAPP	15	26.79%
Twitterlike	5	29.41%

5.7.2 *Sonar PHP*

A Figura 57 demonstra a *interface* apresentada pelo *SonarPHP* após a análise do projeto *Bricks*.

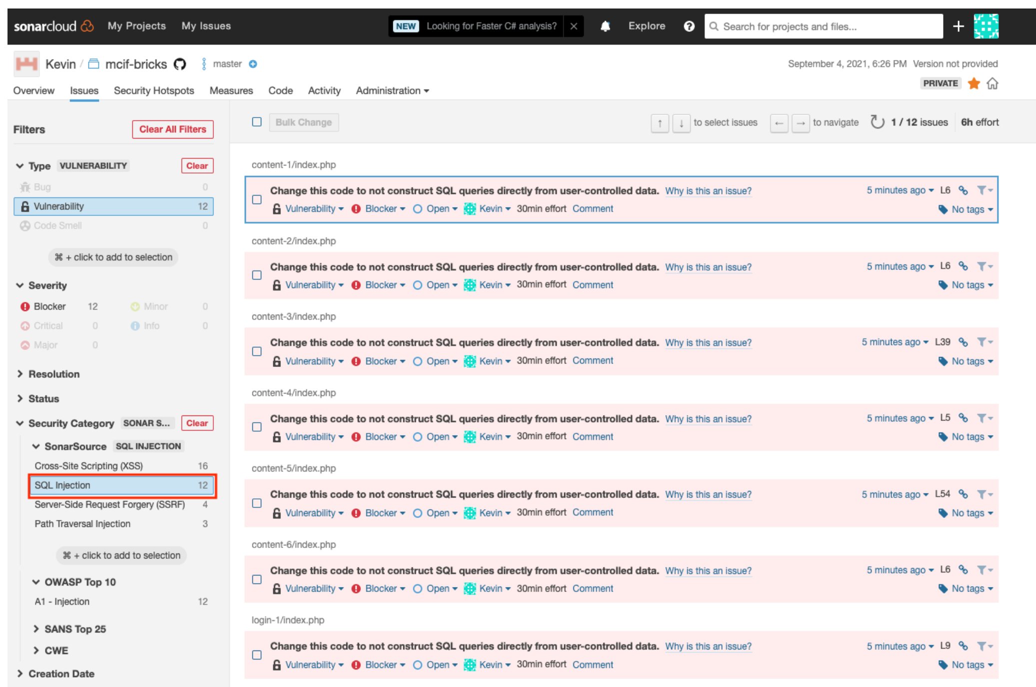

Figura 57: *Interface* do *Sonar PHP* com os resultados da análise do *Bricks*

A Tabela 19 mostra as vulnerabilidades detetadas pelo *Sonar PHP* em cada projeto analisado.

Tabela 19: Vulnerabilidades detetadas pelo *Sonar PHP*

Projeto	Vulnerabilidades	Eficácia
Bricks	12	100%
bWAPP	14	25%
Twitterlike	9	52.94%

5.8 RESULTADOS E ANÁLISE

Nesta secção é apresentada uma comparação analítica dos resultados obtidos. O intuito é responder às questões apresentadas no início do capítulo, com base nos resultados demonstrados anteriormente.

Nas Secções 5.1, 5.2, 5.3 e 5.4 foi apresentada a eficácia que cada algoritmo e versão obteve na deteção de vulnerabilidades do tipo SQL *Injection*. De seguida, é efetuada uma análise comparativa entre estes resultados, bem como, uma análise crítica dos mesmos. Posteriormente serão comparados com outras ferramentas.

5.8.1 *Estudo comparativo entre versões e algoritmos desenvolvidos*

A Tabela 20 apresenta, resumidamente, a quantidade de vulnerabilidades detetadas por cada algoritmo e versão.

Tabela 20: Comparação entre os algoritmos analisados

PROJETO	GA V1	GA V2	ABC V1	ABC V2	ACO V1	ACO V2
Bricks	12	11	12	11	12	11
bWAPP	52	47	33	47	33	51
Twitterlike	13	10	13	10	13	10

A partir dos dados apresentados na Tabela 20 e Figura 58, é possível destacar, que independentemente do algoritmo usado, a versão 1 obteve melhores resultados. Isto é, de certa forma, expectável, visto que a versão 1 gera possivelmente 5 diferentes vetores válidos, enquanto que a versão 2 constrói 1 vetor válido. É pouco provável que haja apenas uma solução a detetar todas as vulnerabilidades no código. Por outro lado, quando o indivíduo possui vários vetores (versão 1), é mais fácil encontrar o conjunto de vetores que permite detetar todas as vulnerabilidades.

A Figura 58 compara os resultados obtidos para cada algoritmo e versão para os projetos analisados. É possível verificar que, para a versão 2, à exceção do ACO, os resultados obtidos foram os mesmos. Isto provavelmente deve-se ao facto de o *dataset* inicial para a versão 2, não ser tão diversificado como no caso da versão 1.

O algoritmo ACO foi o algoritmo que melhores resultados obteve quando aplicado à versão 2.

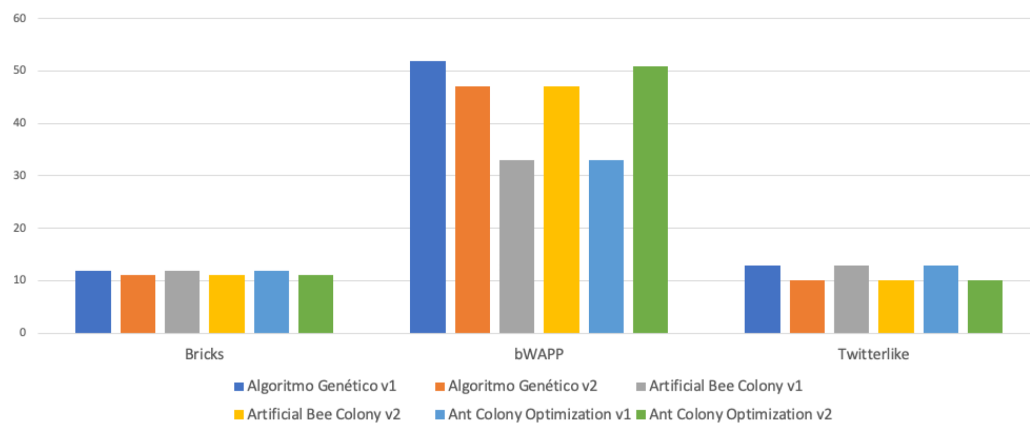

Figura 58: Comparação do número de vulnerabilidades detetadas por cada algoritmo e versão

A Figura 59 apresenta o desvio padrão que cada algoritmo obteve em cada projeto analisado. Dos três projetos, aquele que se manteve menos consistente foi o *Twitterlike*. Os elevados valores de desvio padrão, quando comparados com os outros projetos, revelam maiores amplitudes de *fitness* relativamente à média. Este comportamento tende a acontecer quando determinado vetor de ataque é utilizado para identificar várias vulnerabilidades. Neste caso, as soluções que incluam o vetor *') or 1=1 –*, tendem a detetar uma elevada quantidade de *queries* vulneráveis. Quando uma solução não tem este vetor, várias vulnerabilidades ficam por identificar, levando a esta diferença de resultados. Quando comparado o desvio padrão entre os vários algoritmos, não existem dados conclusivos. No projeto *Bricks*, o AG tem melhores valores por ter um desvio padrão mais baixo. A tendência é oposta no caso do *bWAPP*. As diferenças são, no entanto, muito residuais, e, devem-se ao facto de existirem vetores-chave que solucionam várias vulnerabilidades e, quando não são incluídos, a solução é bastante penalizada.

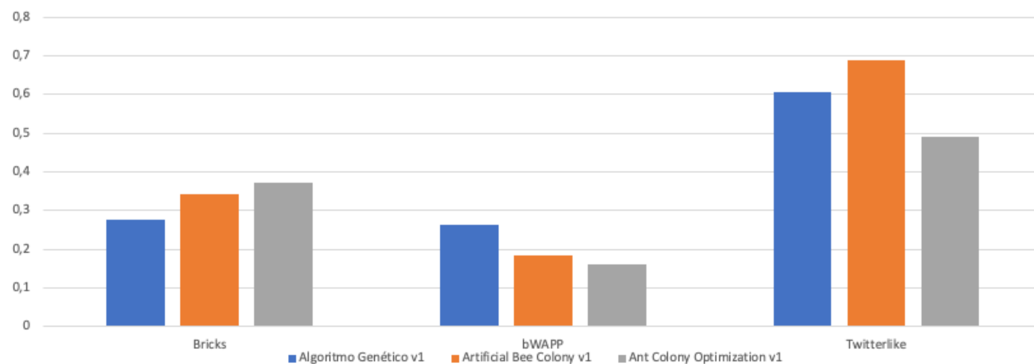

Figura 59: Comparação do desvio padrão entre os vários algoritmos e aplicações web estudadas

5.8.2 *Melhores indivíduos*

De seguida é apresentado para cada algoritmo e versão, o melhor indivíduo obtido no projeto *Bricks*. Para a versão 1, trata-se, portanto, de um conjunto de 5 vetores diferentes, enquanto que na versão 2 é apenas 1 vetor. Para o AG, versão 1, o melhor indivíduo é:

- *1*
- *') or 1=1 –*
- *AND (select substring(concat(1,password),1,1) from users limit 0,1)=1*
- *' or 1=1#*

- *" or 1=1#*

Para o AG, versão 2, o melhor indivíduo é:

- *1=1*
- *1*
- *ANY*
- *x'='x*

Para o ABC, versão 1, o melhor indivíduo é:

- *') or 1=1 –*
- *and or 1=1#*
- *' or '1'='1'#*
- *' or 1=1 –*
- *' and 1=1*

No caso do ABC, versão 2, o melhor indivíduo é:

- *1*
- *x'='x*
- *OR*
- *1=1*

Para o ACO, versão 1, o melhor indivíduo é:

- *') or 1=1 –*
- *' or 1=1 order by 1#*
- *" or 1=1 –*
- *' AND (select 1)=1*
- *' AND (select 1 from users limit 0,1)=1*

No caso do ACO, versão 2, o melhor indivíduo é:

- *1*
- *x'='x*
- *EXISTS*
- *1=1*

Tal como é possível observar, cada algoritmo gerou indivíduos bastante diferentes, capazes de detetar as vulnerabilidades dos projetos analisados.

5.8.3 *Estudo comparativo com outras ferramentas*

Tal como verificámos anteriormente, o AG, na versão 1, foi o que identificou mais vulnerabilidades do tipo SQL *Injection*. Desta forma, este será o algoritmo utilizado como base de comparação com as outras ferramentas observadas.

A Tabela 21 mostra os resultados obtidos ao executar as diferentes ferramentas analisadas. Tal como é possível observar, a abordagem utilizando AG conseguiu identificar mais vulnerabilidades, do que outras abordagens que usam técnicas estáticas.

No projeto *Bricks*, tanto o AG como o *Sonar PHP* conseguiram obter uma eficácia de 100%, ou seja, identificaram todas as vulnerabilidades. O WAP obteve um falso negativo.

O projeto *bWAPP* foi aquele em que a abordagem utilizando AG obteve maior sucesso, com 52 das 56 vulnerabilidades identificadas. Surgindo, de seguida, o WAP com apenas 15. O *Sonar PHP* apenas identificou corretamente 14 vulnerabilidades.

No projeto *Twitterlike*, o AG volta a obter maior percentagem de sucesso, quando comparado com o WAP e o *Sonar PHP*.

Tabela 21: Comparação entre o AG desenvolvido e outras ferramentas de análise estática

Projeto	GA v1	WAP	*Sonar PHP*
Bricks	12	11	12
bWAPP	52	15	14
Twitterlike	13	5	9

A Figura 60 resume a eficácia de cada ferramenta comparada. Tal como se pode constatar, a abordagem atingiu sempre uma eficácia igual ou superior às outras ferramentas.

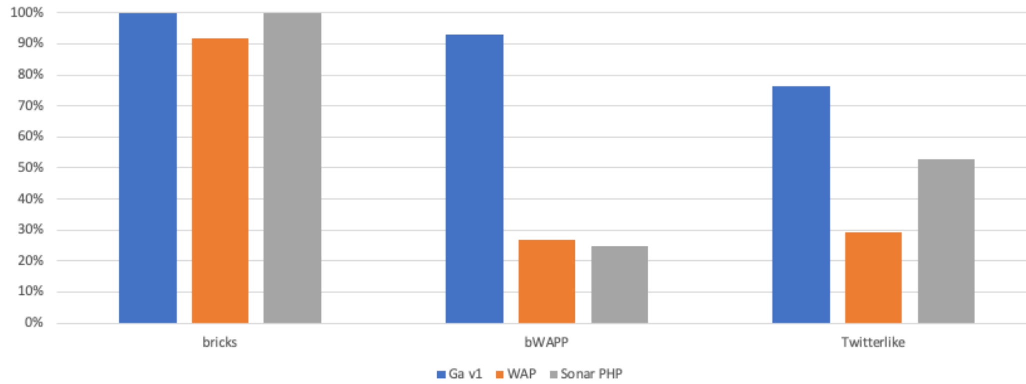

Figura 60: Comparação da eficácia das várias ferramentas com a abordagem usando algoritmos genéticos

Com base nos resultados obtidos, podemos afirmar que uma abordagem utilizando algoritmos de IA pode trazer benefícios na deteção precoce de vulnerabilidades do tipo SQL *Injection*, sendo os vários algoritmos implementados mais eficazes do que outras análises estáticas.

6

CONCLUSÃO

Este capítulo inicia com uma síntese e análise crítica ao trabalho desenvolvido (Secção 6.1). São ainda apresentadas as dificuldades e limitações encontradas (Secção 6.2). Por fim, são ainda indicadas direções para trabalho futuro (Secção 6.3) e as contribuições, nomeadamente, os artigos publicados (Secção 6.4).

6.1 SUMÁRIO E ANÁLISE CRÍTICA

O trabalho começou por apresentar uma revisão literária para identificar quais são as vulnerabilidades mais perigosas e recorrentes na *web*. Tal como constatámos na Secção 2.1, a OWASP apresenta um projeto com as vulnerabilidades com maior risco em 2017, sendo *Injection* a vulnerabilidade com maior risco. De seguida, foi efetuado um estudo para perceber quais são os trabalhos, nesta área de estudo, que permitem a exploração destas vulnerabilidades.

Com base no estudo efetuado, percebeu-se que SQL *Injection* trata-se de uma vulnerabilidade facilmente tratada, mas que ainda assim tem uma grande insurgência na *web*, sendo que as consequências podem ser bastante devastadoras. Neste sentido, estudou-se a possibilidade de detetar esta vulnerabilidade com o uso de algoritmos de IA, em vez de análises estáticas.

O problema foi formulado para encontrar o melhor conjunto de vetores de ataque. Estando perante um problema de otimização, foram utilizados algoritmos de IA. Tal como vimos, a solução proposta foi dividida em duas etapas, sendo a primeira, uma pré-análise de *queries* no código-fonte que serviu para a fase seguinte, em que são utilizados os vários algoritmos implementados. Esta abstração de conceitos é bastante conveniente, possibilitando uma escalabilidade de funcionalidades e a oportunidade de implementar vários algoritmos.

A solução apresentada foi testada em várias aplicações *web* desenvolvidas em PHP. Sendo que apenas a primeira etapa é dependente da linguagem do código-fonte, os algoritmos de procura implementados posteriormente não têm dependências diretas ao código-fonte. Esta abordagem permite que no futuro, a aplicação seja estendida

para outras linguagens de programação sem a necessidade de alterar os algoritmos de pesquisa. Portanto, será apenas necessária a adição de um novo módulo que implemente as funcionalidades necessárias para a extração dos dados necessários, não afetando o processo de procura.

Da mesma forma, a fase de exploração, em que utilizamos um algoritmo de IA, permite que sejam implementados vários algoritmos. De facto, durante o desenvolvimento deste projeto foram implementados vários algoritmos: AG, ABC e ACO, sem a necessidade de efetuar ajustes ao código já desenvolvido.

Tal como vimos, durante a realização do projeto foram modeladas duas versões para identificar vulnerabilidades no código-fonte, ao qual designamos de versão 1 e 2. A versão 1 desenvolvida apresentou melhores resultados, sendo que o AG foi o que mais se destacou. Tal como constatámos na Secção 5.8, o AG quando comparado com outras ferramentas analisadas, conseguiu obter melhores resultados.

A definição de uma função de *fitness* tornou-se um dos grandes desafios desta investigação. Tal como indicado anteriormente, esta função define a qualidade de um indivíduo no ambiente inserido. No nosso problema, um maior valor de *fitness* é indicativo de uma melhor solução, sendo que essa solução consegue detetar mais vulnerabilidades.

Os resultados obtidos sugerem a viabilidade do uso de algoritmos de IA, para a deteção de vulnerabilidades, visto que se obtiveram valores bastantes superiores a outras ferramentas estáticas.

É importante referir que várias unidades curriculares do Mestrado em Cibersegurança e Informática Forense tiveram um papel essencial para a realização deste projeto, nomeadamente "Laboratório de Testes de Penetração" e "Administração Segurança de Sistema Informáticos". Nestas foi possível explorar e aprofundar vários conhecimentos relacionados com falhas de sistemas e, como estes podem ser escalados. Também a unidade curricular de "Projeto de Segurança II" teve um papel muito importante, visto que serviu como base de lançamento para o presente projeto. Esta pesquisa inicial serviu como base para o estado da arte deste trabalho.

6.2 LIMITAÇÕES

A investigação focou-se apenas em vulnerabilidades do tipo SQL *Injection*. Existem, contudo, outras vulnerabilidades que podem ser estudadas. Sendo que a natureza de outras vulnerabilidades pode ser bastante diferente, será necessário novamente

modular o problema, para satisfazer as novas condições. Portanto, apesar de a ferramenta atual poder ser facilmente adaptada a outras linguagens de programação ou dar uso a outros algoritmos, não está abstraída o suficiente para detetar outra categoria de vulnerabilidades. Por exemplo, para identificar possíveis falhas no processo de autenticação, este problema teria de ser modulado para estarmos perante um problema de otimização.

Outra limitação considerada é a linguagem de programação sobre a qual atua, neste caso PHP. Devido à arquitetura implementada é possível adicionar um novo módulo capaz de detetar vulnerabilidades em outras linguagens de programação, no entanto, visto que depende do código-fonte, não foi possível uma solução que seja agnóstica à linguagem de programação. Uma possível solução para este problema, seria a implementação de uma solução que atue na aplicação, em vez de no código.

Uma limitação desta abordagem com base em algoritmos de IA, é a necessidade de uma base de dados inicial, que contém um conjunto de vetores de ataques identificados como potenciais candidatos. Isto leva a que a qualidade da solução esteja dependente deste *dataset* introduzido inicialmente. Para colmatar este problema, desenvolveu-se a versão 2, em que o gene de um indivíduo representa qualquer "palavra" ou caractere e a construção pode ser feita com base num operador lógico (*OR*, *AND*, *WHERE*...). Apesar de, em parte o problema ser resolvido, isto leva a uma evolução mais lenta da população, quando comparando com a versão 1, em que temos vetores completos.

A existência de um genoma fixo, é outra limitação, principalmente na versão 2. Uma vez que o genoma não é dinâmico, *queries* mais complexas nunca são construídas. Esta versão, apesar de ter obtido piores resultados, quando comparada, com a primeira versão, pode-se tornar mais promissora, se a construção do vetor for melhorada.

6.3 TRABALHO FUTURO

No que se refere ao desenvolvimento futuro, a aplicação de algoritmos de IA na área da segurança informática ainda tem bastante por explorar. Desde a aplicação da abordagem desenvolvida a outras vulnerabilidades, bem como a criação de um modelo suficientemente abstrato, capaz de identificar qualquer categoria de vulnerabilidades.

Uma categoria a ser explorada é, tal como indicado anteriormente, executar este modelo numa aplicação *live*, em vez de através do código-fonte. Desta forma, a ferramenta seria agnóstica à linguagem desenvolvida.

Existe, ainda, algum trabalho que pode ser conduzido na obtenção de um *dataset* inicial, dado que se tratou de uma das grandes dificuldades iniciais deste projeto e, a qualidade da solução depender bastante destes valores iniciais.

Seria ainda interessante observar os resultados deste modelo aplicado a outras linguagens de programação que têm surgido mais recentemente em aplicações *web*, tais como *NodeJS* e *Python*.

Por fim, podemos concluir que esta abordagem é viável na deteção de vulnerabilidades e, consequentemente no auxílio de profissionais. De facto, a área de estudo da IA aplicada à segurança informática ainda tem um longo caminho a percorrer. Começam a surgir vários estudos que interligam estas áreas de conhecimento, no entanto, dada a expansão da *web* nos últimos anos, é fulcral conciliar ambos, para minimizar vulnerabilidades.

6.4 CONTRIBUIÇÕES

Tal como referido na Secção 1.2, durante o desenvolvimento deste projeto, foram escritos 4 artigos, encontrando-se já 3 publicados. Na conferência *ICS 2021: 15. International Conference on Security*, em Lisboa, foi publicado um artigo denominado *Exploring SQL Injection Vulnerabilities Using Genetic Algorithms*. Neste artigo, o foco de estudo é o AG utilizando a versão 2 do indivíduo.

Na *18th International Conference Applied Computing*, foi publicado outro artigo, *Exploring sql injection vulnerabilities using artificial bee colony*, sendo que neste o foco principal foi no ABC, na versão 2.

Na *ICITS22 - The 2022 International Conference on Information Technology & Systems* foi aceite outro artigo, *Abordagem baseada em Algoritmos Genéticos para deteção de vulnerabilidades de SQL Injection em Aplicações Web PHP*, que tinha como base o AG que recorre à versão 1 do indivíduo, ou seja, o algoritmo e versão em que se obteve melhores resultados.

Encontra-se ainda, de momento, outro artigo em processo de revisão, em que compara os dois algoritmos *swarm-based*: ABC e ACO, designado *Detecting SQL injection vulnerabilities using Artificial Bee Colony and Ant Colony Optimization*.

BIBLIOGRAFIA

Ahn, Chang Wook e R.S. Ramakrishna (ago. de 2003). «Elitism-based compact genetic algorithms». Em: *IEEE Transactions on Evolutionary Computation* 7.4, pp. 367–385. DOI: 10.1109/tevc.2003.814633. URL: https://doi.org/10.1109%2Ftevc.2003.814633.

Akay, Bahriye e Dervis Karaboga (mar. de 2010). «Artificial bee colony algorithm for large-scale problems and engineering design optimization». Em: *Journal of Intelligent Manufacturing* 23.4, pp. 1001–1014. DOI: 10.1007/s10845-010-0393-4. URL: https://doi.org/10.1007%2Fs10845-010-0393-4.

— (ago. de 2020). «Artificial Bee Colony Algorithm». Em: *Swarm Intelligence Algorithms*. CRC Press, pp. 17–30. DOI: 10.1201/9780429422614-2. URL: https://doi.org/10.1201%2F9780429422614-2.

Alenezi, Mamdouh e Yasir Javed (set. de 2016). «Open source web application security: A static analysis approach». Em: *2016 International Conference on Engineering & MIS (ICEMIS)*. IEEE. DOI: 10.1109/icemis.2016.7745369. URL: https://doi.org/10.1109%2Ficemis.2016.7745369.

Bernardino, Eugénia Moreira et al. (2008). «A Genetic Algorithm with Multiple Operators for Solving the Terminal Assignment Problem». Em: *New Challenges in Applied Intelligence Technologies*. Springer Berlin Heidelberg, pp. 279–288. DOI: 10.1007/978-3-540-79355-7_27. URL: https://doi.org/10.1007%2F978-3-540-79355-7_27.

Casey, William et al. (2014). «Cyber Security via Signaling Games: Toward a Science of Cyber Security». Em: *Distributed Computing and Internet Technology*. Springer International Publishing, pp. 34–42. DOI: 10.1007/978-3-319-04483-5_4. URL: https://doi.org/10.1007%2F978-3-319-04483-5_4.

Dorigo, Marco et al. (1996). *The ant system: Optimization by a colony of cooperating agents*. The MIT Press. DOI: 10.7551/mitpress/1290.001.0001. URL: https://doi.org/10.7551%2Fmitpress%2F1290.001.0001.

Elderman, Richard et al. (2017). «Adversarial Reinforcement Learning in a Cyber Security Simulation». Em: *Proceedings of the 9th International Conference on Agents and Artificial Intelligence*. SCITEPRESS - Science e Technology Publications. DOI: 10.5220/0006197105590566. URL: https://doi.org/10.5220%2F0006197105590566.

Gen, Mitsuo e Runwei Cheng (nov. de 2007). «Foundations of Genetic Algorithms». Em: pp. 1–52. ISBN: 9780471315315. DOI: 10.1002/9780470172261.ch1.

Goldberg, D. (out. de 1988). «Genetic algorithms and Machine Learning». Em: *Machine Learning* 3.2-3, pp. 95–99. DOI: 10.1007/bf00113892. URL: https://doi.org/10.1007%2Fbf00113892.

Goldberg e Deb (1991). «A Comparative Analysis of Selection Schemes Used in Genetic Algorithms». Em: *Foundations of Genetic Algorithms*. Elsevier, pp. 69–93. DOI: 10.1016/b978-0-08-050684-5.50008-2. URL: https://doi.org/10.1016%2Fb978-0-08-050684-5.50008-2.

Holland, John H. (jul. de 1992). «Genetic Algorithms». Em: *Scientific American* 267.1, pp. 66–72. DOI: 10.1038/scientificamerican0792-66. URL: https://doi.org/10.1038%2Fscientificamerican0792-66.

Hu, Xiao-Bing e Ezequiel Di Paolo (2009). «An Efficient Genetic Algorithm with Uniform Crossover for the Multi-Objective Airport Gate Assignment Problem». Em: *Multi-Objective Memetic Algorithms*. Springer Berlin Heidelberg, pp. 71–89. DOI: 10.1007/978-3-540-88051-6_4. URL: https://doi.org/10.1007%2F978-3-540-88051-6_4.

Huizinga, Dorota e Adam Kolawa (set. de 2007). *Automated Defect Prevention*. John Wiley & Sons, Inc. DOI: 10.1002/9780470165171. URL: https://doi.org/10.1002%2F9780470165171.

Ilieva, S., P. Ivanov e E. Stefanova (2004). «Analyses of an agile methodology implementation». Em: *Proceedings. 30th Euromicro Conference, 2004*. IEEE. DOI: 10.1109/eurmic.2004.1333387. URL: https://doi.org/10.1109%2Feurmic.2004.1333387.

Karaboga, Dervis (jan. de 2005). «An Idea Based on Honey Bee Swarm for Numerical Optimization, Technical Report - TR06». Em: *Technical Report, Erciyes University*.

Kile, J. (2007). «Agile Software Development Quality Assurance». Em: *Agile Software Development Quality Assurance*. IGI Global. DOI: 10.4018/9781599042169.ch010. URL: https://doi.org/10.4018%2F9781599042169.ch010.

Kumar, Diksha Gautam e Madhumita Chatterjee (jul. de 2014). «MAC based solution for SQL injection». Em: *Journal of Computer Virology and Hacking Techniques* 11.1, pp. 1–7. DOI: 10.1007/s11416-014-0219-6. URL: https://doi.org/10.1007%2Fs11416-014-0219-6.

Liu, Miao et al. (2019). «A Survey of Exploitation and Detection Methods of XSS Vulnerabilities». Em: *IEEE Access* 7, pp. 182004–182016. DOI: 10.1109/access.2019.2960449. URL: https://doi.org/10.1109%2Faccess.2019.2960449.

Mckinnel, Dean et al. (mai. de 2019). «A systematic literature review and meta-analysis on artificial intelligence in penetration testing and vulnerability assessment R». Em: *Computers & Electrical Engineering* 75. DOI: 10.1016/j.compeleceng.2019.02.022.

Medeiros, Ibéria, Nuno F. Neves e Miguel Correia (2014). «Automatic detection and correction of web application vulnerabilities using data mining to predict false positives». Em: *Proceedings of the 23rd international conference on World wide web - WWW '14*. ACM Press. DOI: 10.1145/2566486.2568024. URL: https://doi.org/10.1145%2F2566486.2568024.

Michalewicz, Zbigniew (out. de 2018). «Mutation operators». Em: *Evolutionary Computation 1*. CRC Press, pp. 275–293. DOI: 10.1201/9781482268713-39. URL: https://doi.org/10.1201%2F9781482268713-39.

Mirjalili, Seyedali (jun. de 2018). «Genetic Algorithm». Em: *Studies in Computational Intelligence*. Springer International Publishing, pp. 43–55. DOI: 10.1007/978-3-319-93025-1_4. URL: https://doi.org/10.1007%2F978-3-319-93025-1_4.

Mishra, D. (fev. de 2021). «SQL Injection Bypassing WAF». Em: *OWASP*.

Nadarajah, S. (jun. de 2008). «An Explicit Selection Intensity of Tournament Selection-Based Genetic Algorithms». Em: *IEEE Transactions on Evolutionary Computation* 12.3, pp. 389–391. DOI: 10.1109/tevc.2007.899589. URL: https://doi.org/10.1109%2Ftevc.2007.899589.

— (out. de 2018). «Tournament selection». Em: *Evolutionary Computation 1*. CRC Press, pp. 219–224. DOI: 10.1201/9781482268713-31. URL: https://doi.org/10.1201%2F9781482268713-31.

Nakama, Takéhiko (2008). «Markov Chain Analysis of Genetic Algorithms Applied to Fitness Functions Perturbed by Multiple Sources of Additive Noise». Em: *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. Ed. por Roger Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 123–136. DOI: 10.1007/978-3-540-70560-4_11. URL: https://doi.org/10.1007/978-3-540-70560-4_11.

Niculae, Stefan (out. de 2018). «Reinforcement Learning vs Genetic Algorithms in Game-Theoretic Cyber-Security». Em: DOI: 10.31237/osf.io/nxzep. URL: https://doi.org/10.31237%2Fosf.io%2Fnxzep.

OWASP Top Ten (2017). URL: https://owasp.org/www-project-top-ten/ (acedido em 16/08/2021).

Raghuwanshi, M. M. e G. G. Kakde (2006). «Multi-parent Recombination Operator with Multiple Probability Distribution for Real Coded Genetic Algorithm». Em: *Advances in Intelligent and Soft Computing*. Springer Berlin Heidelberg,

pp. 393–402. DOI: 10.1007/978-3-540-36266-1_38. URL: https://doi.org/10.1007%2F978-3-540-36266-1_38.

Rexhepi, Rexhep, Ramiz Hoxha e Betim Gashi (out. de 2017). «Comparison of Web Development Technologies ASP.NET vs. PHP». Em: *2017 UBT International Conference*. University for Business e Technology. DOI: 10.33107/ubt-ic.2017.95. URL: https://doi.org/10.33107%2Fubt-ic.2017.95.

Semenkin, Eugene e Maria Semenkina (jun. de 2012). «Self-configuring genetic programming algorithm with modified uniform crossover». Em: *2012 IEEE Congress on Evolutionary Computation*. IEEE. DOI: 10.1109/cec.2012.6256587. URL: https://doi.org/10.1109%2Fcec.2012.6256587.

Shang, Yi e Guo-Jie Li (nov. de 1991). «New crossover operators in genetic algorithms». Em: *[Proceedings] Third International Conference on Tools for Artificial Intelligence - TAI 91*. IEEE Comput. Soc. Press. DOI: 10.1109/tai.1991.167090. URL: https://doi.org/10.1109%2Ftai.1991.167090.

Spears, William M. e Kenneth D. De Jong (jan. de 1995). *On the Virtues of Parameterized Uniform Crossover*, rel. téc. DOI: 10.21236/ada293985. URL: https://doi.org/10.21236%2Fada293985.

Stefinko, Yaroslav, Andrian Piskozub e Roman Banakh (fev. de 2016). «Manual and automated penetration testing. Benefits and drawbacks. Modern tendency». Em: *2016 13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET)*. IEEE. DOI: 10.1109/tcset.2016.7452095. URL: https://doi.org/10.1109%2Ftcset.2016.7452095.

Stiawan, Deris et al. (2017). «Cyber-attack penetration test and vulnerability analysis». Em: *International Journal of Online Engineering* 13.1, pp. 125–132. ISSN: 18612121. DOI: 10.3991/ijoe.v13i01.6407.

APÊNDICES

A

CONFIGURAÇÕES

A.1 FICHEIRO *DOCKER* PARA INSTALAÇÃO DOS PROJETOS ESTUDADOS NUMA MÁQUINA LOCAL

```yaml
version: '3.3'

services:
  php:
    image: php:5.5.29-apache
    ports:
      - 8090:80
    volumes:
      - .:/var/www/html/
    depends_on:
      - mysql

  mysql:
    image: mysql
    command: [
        '--default_authentication_plugin=mysql_native_password',
        '--character-set-server=utf8mb4',
        '--collation-server=utf8mb4_unicode_ci'
    ]
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: example
    ports:
      - 3308:3306
```

Listagem 34: Ficheiro para instalação de projetos

B

INICIALIZAÇÃO DOS PROJETOS ESTUDADOS

De seguida são apresentados os *scripts* executados para a criação e população de dados para os projetos analisados.

B.1 INICIALIZAR *BRICKS*

```
1  create DATABASE bricks;
2
3  CREATE table bricks.users
4  (
5      idusers  INT           NOT NULL,
6      name     VARCHAR(45)   NOT NULL,
7      email    VARCHAR(45)   NOT NULL,
8      password VARCHAR(45)   NOT NULL,
9      ua       VARCHAR(45)   NOT NULL,
10     ref      VARCHAR(145)  NOT NULL,
11     host     VARCHAR(45)   NOT NULL,
12     lang     VARCHAR(45)   NOT NULL,
13     PRIMARY KEY (idusers)
14 );
15
16 INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
17 VALUES (0, 'admin', 'admin@getmantra.com', 'admin', 'Brick_Browser',
   ↪  '$server$scriptpath/content-13/index.php',
18         '127.0.0.1', 'en');
19
20 INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
21 VALUES (1, 'tom', 'tom@getmantra.com', 'tom', 'Block_Browser', '', '8.8.8.8',
   ↪  'en');
22
23
24 INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
25 VALUES (2, 'harry', '22@getmantra.com', '5f4dcc3b5aa765d61d8327deb882cf99',
   ↪  'Mantra', '', '127.0.0.1', 'en');
26
27
28 INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
29 VALUES (3, 'harry', '33@getmantra.com', '5f4dcc3b5aa765d61d8327deb882cf99',
   ↪  'Mantra', '', '127.0.0.1', 'en');
30
31
32 INSERT INTO bricks.users (idusers, name, email, password, ua, ref, host, lang)
33 VALUES (4, 'harry', '44@getmantra.com', '5f4dcc3b5aa765d61d8327deb882cf99',
   ↪  'Mantra', '', '127.0.0.1', 'en');
```

Listagem 35: Ficheiro para criar e popular o projeto *Bricks*

B.2 INICIALIZAR *BWAPP*

```
CREATE DATABASE bwapp;
USE bwapp;

CREATE TABLE IF NOT EXISTS users
(
    id              int(10) NOT NULL AUTO_INCREMENT,
    login           varchar(100) DEFAULT NULL,
    password        varchar(100) DEFAULT NULL,
    email           varchar(100) DEFAULT NULL,
    secret          varchar(100) DEFAULT NULL,
    activation_code varchar(100) DEFAULT NULL,
    activated       tinyint(1)   DEFAULT '0',
    reset_code      varchar(100) DEFAULT NULL,
    admin           tinyint(1)   DEFAULT '0',
    PRIMARY KEY (id)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8
  AUTO_INCREMENT = 1;

INSERT INTO users (login, password, email, secret, activation_code, activated,
↪   reset_code, admin)
VALUES ('A.I.M.', '6885858486f31043e5839c735d99457f045affd0',
↪   'bwapp-aim@mailinator.com',
        'A.I.M. or Authentication Is Missing', NULL, 1, NULL, 1),
       ('bee', '6885858486f31043e5839c735d99457f045affd0',
       ↪   'bwapp-bee@mailinator.com', 'Any bugs?', NULL, 1, NULL, 1);


CREATE TABLE IF NOT EXISTS blog
(
    id    int(10) NOT NULL AUTO_INCREMENT,
    owner varchar(100) DEFAULT NULL,
    entry varchar(500) DEFAULT NULL,
    date  datetime     DEFAULT NULL,
    PRIMARY KEY (id)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8
  AUTO_INCREMENT = 1;


CREATE TABLE IF NOT EXISTS visitors
(
    id         int(10) NOT NULL AUTO_INCREMENT,
    ip_address varchar(50)  DEFAULT NULL,
    user_agent varchar(500) DEFAULT NULL,
    date       datetime     DEFAULT NULL,
    PRIMARY KEY (id)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8
```

```
47    AUTO_INCREMENT = 1;
48
49
50  CREATE TABLE IF NOT EXISTS movies
51  (
52      id             int(10) NOT NULL AUTO_INCREMENT,
53      title          varchar(100) DEFAULT NULL,
54      release_year   varchar(100) DEFAULT NULL,
55      genre          varchar(100) DEFAULT NULL,
56      main_character varchar(100) DEFAULT NULL,
57      imdb           varchar(100) DEFAULT NULL,
58      tickets_stock  int(10)      DEFAULT NULL,
59      PRIMARY KEY (id)
60  ) ENGINE = InnoDB
61    DEFAULT CHARSET = utf8
62    AUTO_INCREMENT = 1;
63
64  INSERT INTO movies (title, release_year, genre, main_character, imdb,
    ↪  tickets_stock)
65  VALUES ('G.I. Joe: Retaliation', '2013', 'action', 'Cobra Commander',
    ↪  'tt1583421', 100),
66         ('Iron Man', '2008', 'action', 'Tony Stark', 'tt0371746', 53),
67         ('Man of Steel', '2013', 'action', 'Clark Kent', 'tt0770828', 78),
68         ('Terminator Salvation', '2009', 'sci-fi', 'John Connor', 'tt0438488',
           ↪  100),
69         ('The Amazing Spider-Man', '2012', 'action', 'Peter Parker', 'tt0948470',
           ↪  13),
70         ('The Cabin in the Woods', '2011', 'horror', 'Some zombies', 'tt1259521',
           ↪  666),
71         ('The Dark Knight Rises', '2012', 'action', 'Bruce Wayne', 'tt1345836',
           ↪  3),
72         ('The Fast and the Furious', '2001', 'action', 'Brian OConnor',
           ↪  'tt0232500', 40),
73         ('The Incredible Hulk', '2008', 'action', 'Bruce Banner', 'tt0800080',
           ↪  23),
74         ('World War Z', '2013', 'horror', 'Gerry Lane', 'tt0816711', 0);
75
76  CREATE TABLE IF NOT EXISTS heroes
77  (
78      id       int(10) NOT NULL AUTO_INCREMENT,
79      login    varchar(100) DEFAULT NULL,
80      password varchar(100) DEFAULT NULL,
81      secret   varchar(100) DEFAULT NULL,
82      PRIMARY KEY (id)
83  ) ENGINE = InnoDB
84    DEFAULT CHARSET = utf8
85    AUTO_INCREMENT = 1;
86
87
88  INSERT INTO heroes (login, password, secret)
89  VALUES ('neo', 'trinity', 'Oh why didnt I took that BLACK pill?'),
90         ('alice', 'loveZombies', 'Theres a cure!'),
91         ('thor', 'Asgard', 'Oh, no... this is Earth... isnt it?'),
92         ('wolverine', 'Log@N', 'Whats a Magneto?'),
```

```
93          ('johnny', 'm3ph1st0ph3l3s', 'Im the Ghost Rider!'),
94          ('seline', 'm00n', 'It wasnt the Lycans. It was you.');
```

Listagem 36: Ficheiro para criar e popular o projeto *bWAPP*

B.3 INICIALIZAR *TWITTERLIKE*

```
1   SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";
2
3
4   /*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
5   /*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
6   /*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
7   /*!40101 SET NAMES utf8 */;
8
9   --
10  -- Database: `twitterlike`
11  --
12
13  create DATABASE if not exists twitterlike;
14  -- --------------------------------------------------------
15
16  --
17  -- Table structure for table `follow`
18  --
19
20  use twitterlike;
21  CREATE TABLE IF NOT EXISTS `follow` (
22                                          `idrel` int(11) NOT NULL AUTO_INCREMENT,
23                                          `usera` varchar(30) COLLATE
                                            ↪  utf8_unicode_ci NOT NULL,
24                                          `userb` varchar(30) COLLATE
                                            ↪  utf8_unicode_ci NOT NULL,
25                                          PRIMARY KEY (`idrel`),
26                                          UNIQUE KEY `usera` (`usera`,`userb`)
27  ) ENGINE=MyISAM  DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci AUTO_INCREMENT=14 ;
28
29  --
30  -- Dumping data for table `follow`
31  --
32
33  INSERT INTO `follow` (`idrel`, `usera`, `userb`) VALUES
34  (1, 'saktidc', 'h1zbullah'),
35  (2, 'saktidc', 'gerrard'),
36  (10, 'saktidc', 'beukicot'),
37  (4, 'h1zbullah', 'saktidc'),
38  (13, 'saktidc', 'rakhmat'),
39  (11, 'gerrard', 'saktidc'),
```

```
40  (12, 'saktidc', 'cahyono');
41
42  -- --------------------------------------------------------
43
44  --
45  -- Table structure for table `tweet`
46  --
47
48  CREATE TABLE IF NOT EXISTS `tweet` (
49                                            `id` int(11) NOT NULL AUTO_INCREMENT,
50                                            `username` varchar(30) COLLATE
                                              ↪ utf8_unicode_ci NOT NULL,
51                                            `tglwaktu` datetime NOT NULL,
52                                            `isi` varchar(200) COLLATE
                                              ↪ utf8_unicode_ci NOT NULL,
53                                            PRIMARY KEY (`id`)
54  ) ENGINE=MyISAM  DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci AUTO_INCREMENT=37 ;
55
56  --
57  -- Dumping data for table `tweet`
58  --
59
60  INSERT INTO `tweet` (`id`, `username`, `tglwaktu`, `isi`) VALUES
61  (1, 'saktidc', '2010-05-30 22:09:16', '@saktidc tweet ke diri sendiri'),
62  (3, 'saktidc', '2010-06-01 10:06:27', 'haha twetting alone'),
63  (4, 'h1zbullah', '2010-06-01 11:43:40', 'la la la la'),
64  (5, 'cahyono', '2010-06-01 11:48:14', 'tangan sampai keriting, nulis terus'),
65  (6, 'saktidc', '2010-06-01 11:49:32', 'tangan sampai keriting'),
66  (7, 'saktidc', '2010-06-01 11:52:01', 'tangan sampai keriting'),
67  (8, 'gerrard', '2010-06-01 12:22:06', 'you''ll never tweet alone'),
68  (9, 'h1zbullah', '2010-06-02 07:55:15', 'twet lagi hari rabu'),
69  (10, 'rakhmat', '2010-06-02 07:58:20', 'coba tweet'),
70  (11, 'rakhmat', '2010-06-02 07:58:30', 'tweet lagi'),
71  (12, 'h1zbullah', '2010-06-02 07:59:15', 'hehe he'),
72  (13, 'saktidc', '2010-06-02 07:59:38', '@h1zbullah ngapain lo'),
73  (14, 'gerrard', '2010-06-02 08:57:22', 'mau ganti foto ah'),
74  (15, 'rakhmat', '2010-06-02 09:35:03', 'aku juga ganti foto\r\n\r\nhehe'),
75  (16, 'saktidc', '2010-06-02 09:38:04', '@gerrard kapan main bola lagi!
    ↪ <b>semangat</b>'),
76  (17, 'cahyono', '2010-06-02 09:47:11', 'aku juga pengen ganti icon'),
77  (18, 'cahyono', '2010-06-02 09:48:10', 'coba ngetik pake bahasa korean'),
78  (19, 'cahyono', '2010-06-02 09:48:38', '&lt;br/&gt;
    ↪ &lt;script&gt;alert(&#039;coba ngecrack&#039;);&lt;/script&gt;'),
79  (20, 'rakhmat', '2010-06-02 09:49:21', '@cahyono ngecracknya kok masih pake cara
    ↪ ecek2'),
80  (21, 'cahyono', '2010-06-02 10:17:57', 'ngga bosen apa?'),
81  (22, 'gerrard', '2010-06-02 10:18:25', 'lalala'),
82  (23, 'rakhmat', '2010-06-02 10:18:41', 'yoi'),
83  (24, 'saktidc', '2010-06-02 10:19:23', 'mau number 1 di jumlah tweet'),
84  (25, 'h1zbullah', '2010-06-02 10:49:01', '@saktidc dibales ya?'),
85  (26, 'saktidc', '2010-06-02 11:22:26', 'satu'),
86  (27, 'saktidc', '2010-06-02 11:22:35', 'one'),
87  (29, 'saktidc', '2010-06-02 15:34:57', 'he he'),
88  (30, 'saktidc', '2010-06-02 15:49:19', 'koding terus'),
```

```
(31, 'gerrard', '2010-06-02 16:12:29', '@saktidc   gantian balas ah'),
(32, 'saktidc', '2010-06-02 16:12:45', '@gerrard aku juga ikut ikutan'),
(33, 'h1zbullah', '2010-06-02 16:14:28', 'aaaa'),
(34, 'saktidc', '2010-06-02 16:14:47', '@h1zbullah halah pake bahasa korea'),
(35, 'h1zbullah', '2010-06-02 16:15:18', '@saktidc'),
(36, 'h1zbullah', '2010-06-02 16:16:01', ' tangan sampai keriting');

-- --------------------------------------------------------

--
-- Table structure for table `user`
--

CREATE TABLE IF NOT EXISTS `user` (
                                      `username` varchar(30) COLLATE
                                      ↪  utf8_unicode_ci NOT NULL,
                                      `password` varchar(32) COLLATE
                                      ↪  utf8_unicode_ci NOT NULL,
                                      `email` varchar(50) COLLATE
                                      ↪  utf8_unicode_ci NOT NULL,
                                      `fullname` varchar(100) COLLATE
                                      ↪  utf8_unicode_ci NOT NULL,
                                      `image` varchar(50) COLLATE
                                      ↪  utf8_unicode_ci DEFAULT NULL,
                                      `bio` text COLLATE utf8_unicode_ci,
                                      `lasttweet` int(11) DEFAULT NULL,
                                      PRIMARY KEY (`username`),
                                      UNIQUE KEY `email` (`email`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;

--
-- Dumping data for table `user`
--

INSERT INTO `user` (`username`, `password`, `email`, `fullname`, `image`, `bio`,
↪  `lasttweet`) VALUES
('saktidc', 'd77575cc735cf44edc38086c8df4e033', '54krpl@gmail.com', 'Sakti Dwi
↪  Cahyono', 'image.png', 'no comment lah', 34),
('h1zbullah', '7215ee9c7d9dc229d2921a40e899ec5f', 'h1zbullah@yahoo.co.id',
↪  'dwi', '882c931b20f15e24b6b1db6dcbd1cb2d.jpg', 'orang biasa', 36),
('cahyono', 'd77575cc735cf44edc38086c8df4e033', 'sakti@mail.com', 'cahyono dwi
↪  sakti', '9e23b6cb382283bca3ca06b6b586ef65.png', 'seorang mahasiswa', 21),
('gerrard', 'b2f5ff47436671b6e533d8dc3614845d', 'g@gmail.com', 'stepen gerrard',
↪  '7965cfaf9f7a6e4d6510d823dd6e81f9.jpg', 'asdf jkl; persis keyboard', 31),
('beukicot', 'fd6cee015c4ec774cc3e6937d603ed15', 'beukicot@gmail.com', 'fauzan
↪  riyadi malik', 'default_1.png', 'asli sukabumi kang', NULL),
('rakhmat', '7215ee9c7d9dc229d2921a40e899ec5f', 'rakhmat@gmail.com', 'rakhmat
↪  hidayat', '3a8beb09b4c06b1aa66dc46a169681aa.jpg', 'no need to know', 23);
```

Listagem 37: Ficheiro para criar e popular o projeto *Twitterlike*

DECLARAÇÃO

Declaro, sob compromisso de honra, que o trabalho apresentado neste projeto, com o título *"Exploração de Vulnerabilidades com Algoritmos de Inteligência Artificial"*, é original e foi realizado por Kevin Baptista (2190371) sob orientação da Professora Doutora Anabela Bernardino e da Professora Doutora Eugénia Bernardino.

Leiria, novembro de 2021

Kevin Baptista