UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

HENRIQUE VERMELHO DE TOLEDO
DAIGORO ALENCAR DE OLIVEIRA

wafamole++: Um fuzzer mutacional para injeções SQL

RIO DE JANEIRO
2022

HENRIQUE VERMELHO DE TOLEDO
DAIGORO ALENCAR DE OLIVEIRA

wafamole++: Um fuzzer mutacional para injeções SQL

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Paulo Henrique Rodrigues Aguiar
Co-orientador: Prof. Claudio Miceli de Farias

RIO DE JANEIRO
2022

CIP - Catalogação na Publicação

T649w

Toledo, Henrique Vermelho de
wafamole ++: um fuzzer mutacional para injeções SQL / Henrique Vermelho de Toledo, Daigoro Alencar de Oliveira. – 2022.

72 f.

Orientador: Paulo Henrique Rodrigues Aguiar.
Coorientador: Claudio Miceli de Farias.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2022.

1. Fuzzing. 2. Aplicações web. 3. Segurança. 4. Segurança da informação. 5. Aprendizado de máquina. 6. Firewalls. I. Oliveira, Daigoro Alencar de. II. Aguiar, Paulo Henrique Rodrigues (Orient). III. Farias, Claudio Miceli de (Coorient.). IV. Universidade Federal do Rio de Janeiro, Instituto de Computação. V. Título.

HENRIQUE VERMELHO DE TOLEDO
DAIGORO ALENCAR DE OLIVEIRA

wafamole++: Um fuzzer mutacional para injeções SQL

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 9 de Setembro de 2022

BANCA EXAMINADORA:

Paulo Henrique de Aguiar Rodrigues
Ph.D.
Professor Titular (IC/UFRJ)

Claudio Miceli Farias
D.Sc.
Professor Adjunto (NCE/UFRJ)

Valeria Menezes Bastos
D.Sc.
Professora Associada (IC/UFRJ)

João Carlos Pereira da Silva
D.Sc.
Professor Associado (IC/UFRJ)

AGRADECIMENTOS

Dirigimos nossos agradecimentos aos professores da Universidade Federal do Rio de Janeiro que fomentaram o caminho para chegar nesse trabalho e no nosso perfil profissional/acadêmico, com destaque a Claudio Miceli e Paulo Aguiar pela excelente orientação nessa área de interesse.

Henrique: Meus agradecimentos à família e amigos que me ajudaram enormemente a conseguir realizar este trabalho. Agradeço a minha mãe Alane Beatriz, por moldar e direcionar meu caráter, interesse profissional e me engrandecer como pessoa a cada dia. Ao meu pai, Cylo Homero, por me levar em suas inúmeras aventuras e acompanhar sempre minha trajetória de vida. Agradeço também o professor Marcello Goulart, por me mostrar que há muito mais a se viver além da faculdade e trabalho, e me apresentar o montanhismo e escalada.

RESUMO

Em constante mudança e sempre em relevância para o mercado, a área de segurança Web possui sempre novas iniciativas e exige dos profissionais comprometidos com a mesma um sério nível de engajamento para se adaptar às mais recentes vulnerabilidades. Nesse contexto, buscou-se, por meio de uma revisão de literatura com o tópico de Vulnerabilidades de Aplicações Web, soluções para serem complementadas/estendidas na área pelos autores. Dentre os resultados da pesquisa, gerou-se um complemento a uma aplicação *open-source* de segurança, o *WAF-A-MoLE*, dando luz a um *upgrade* denominado *wafamole++*. O *WAF-A-MoLE* é uma ferramenta de geração de exemplos adversariais para *Firewalls* de Aplicações Web baseados em Aprendizado de Máquina contra injeções SQL, elencando técnicas como Fuzzing Mutacional e contemplando diversos tipos de classificadores de dados, enquanto o *wafamole++* é uma versão estendida do mesmo com um módulo de treinamento, e tratamento de dados, novos modelos de ataque e operadores de mutação. Para a elaboração da ferramenta *wafamole++*, utilizou-se um *firewall open-source* desse tipo juntamente dos dados originais disponibilizados pelos autores e *datasets* oriundos de plataformas como o *Kaggle* para o treinamento de novos modelos. Além disso, uma série de novos operadores de mutação intrínsecos ao funcionamento do software base foram criados para uma geração de exemplos mais ricos e eficientes. Acredita-se que com essa expansão, um leque de aplicações Web modernas possa ser enriquecido com aprimoramentos aos seus *firewalls*.

Palavras-chave: fuzzing; aplicações web; segurança; segurança da informação; aprendizado de máquina; firewalls.

ABSTRACT

Web security has been in constant change and evolution for the past decades, and the constant stream of new initiatives demand a serious level of engagement from the professionals committed to it to adapt to the latest vulnerabilities. In this context, a literature review on the topic of Web Application Vulnerabilities has been conducted by the authors in order to extend and/or complement solutions in the area. Among the research results, an extension to an open-source security application (WAF-A-MoLE) has been made, entitled wafamole++. WAF-A-MoLE is a tool built for generating adversarial examples to be used in Web Application Firewalls based on Machine Learning against SQL injections, leveraging techniques such as Mutational Fuzzing and contemplating several types of data classifiers. wafamole++ is an extension of this tool including a dataset training/pruning module, several new models and mutation operators and improved documentation. For its elaboration, an open-source firewall of such type was used along with the original data made available by the authors, as well as datasets from platforms such as Kaggle for training new models. In addition, a series of new mutation operators intrinsic to the operation of the base software were created for the generation of richer and more efficient adversarial examples. It is believed that with this expanded release, a wide range of modern Web applications can be enriched with enhancements to their firewalls.

Keywords: fuzzing; web applications; security; information security; machine learning; firewalls.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

WAF	- Web Application Firewall
HTTP	- Hypertext Transfer Protocol
HTTPS	- Hypertext Transfer Protocol Secure
IP	- Internet Protocol
HTML	- HyperText Markup Language
DOM	- Document Object Model
SQL	- Structured Query Language
XSS	- Cross Site Scripting
ML	- Machine Learning
SVM	- Support Vector Machine
RBF	- Radial Basis Function
SVC	- Support Vector Classification
SGD	- Stochastic Gradient Descent
PAC	- Probably Approximately Correct
BBM	- Boost By Majority
CAFe	- Comunidade Acadêmica Federada
PICOC	- Population, Intervention, Comparison, Outcomes, Context
GDPR	- General Data Protection Regulation
REST	- Representational State Transfer
WSL	- Windows Subsystem for Linux

SUMÁRIO

1 INTRODUÇÃO

Para o contexto de desenvolvimento de software, segurança da informação é a área que potencialmente define entre o sucesso ou fracasso de uma aplicação. E em aplicações web com elevado tráfego, a preocupação com a segurança pode ter uma manifestação tardia e comprometer o uso do ambiente. Atualmente navegadores web como *Chrome*, *Firefox* e *Safari* ativamente notificam o usuário de falhas graves nesse quesito como certificação HTTPS, tornando ações fundamentais como transações (quando inseguras) na web uma área de atenção redobrada para consumidores finais. Nela, ações de segurança são um recurso crítico e sensível que faz com que poucos sites discorram sobre os seus segredos de segurança, até como uma estratégia contra *hackers*. Isso torna complicado assegurar como é o estado atual de vulnerabilidades web hoje em dia. Nesse sentido, este trabalho procura elencar o estado atual das medidas conhecidas para conter o avanço de ataques às aplicações web e introduzir uma nova ferramenta de código aberto como contribuição adicional para fortalecer equipes de segurança contra as vulnerabilidades associadas aos sítios web.

Historicamente, o conceito de vulnerabilidade na Internet modificou-se consideravelmente. A *World Wide Web* consistia, nos seus primórdios, de web sites cujas funções principais eram essencialmente fornecer ao usuário final documentos estáticos de informação, surgida inicialmente nos anos 90. O fluxo de informação era de via única, no qual o conteúdo saía do servidor ao usuário final, apenas. Autenticação não era uma realidade para a maioria dos sites, pois não havia necessidade. Cada usuário que visitasse uma página web recebia a mesma informação. (STUTTARD; PINTO, 2007) Vulnerabilidades nessa época consistiam principalmente em numerosos *exploits* [1] em cima de software de servidores, para distribuir software de computador ilegalmente copiado com seus códigos de proteção desativados [2] (conhecidos em inglês como *warez*) ou meramente deturpar a parte visual de um site (prática conhecida como *defacement*).

Na atualidade a maioria dos sites na Internet são aplicações robustas com diversas funcionalidades que eram inconcebíveis na época, dependendo de um fluxo de informação de via dupla entre cliente e servidor. Usuários precisam de credenciais para realizar cadastros que cuidam das mais modestas trocas de informações como postagens em fóruns de discussão até os mais sensíveis dados bancários transacionais. Surge a necessidade concreta de gerenciadores de senha de ponta, muitas vezes oferecidos pelo próprio navegador web (embora inseguro). Além disso, torna-se onipresente a linguagem de script/programa-

[1] Em português significa explorar, significando usar algo para própria vantagem. É um software ou sequência de comandos que toma vantagem de um defeito em uma aplicação para causar um comportamento imprevisto tipicamente com a finalidade de obter controle de tal aplicação.

[2] A desativação dos códigos de proteção é importante para que a cópia possa ser distribuída sem dificuldades.

ção *JavaScript* moderna, que fornece a essas aplicações um dinamismo inerente, tanto na parte visual do cliente (*front-end*), como na parte funcional do servidor (*back-end*, através de tecnologias como Node.js). Cada internauta torna-se diferente do próximo, tornando a interação com o site totalmente dependente do usuário, o que afeta o conteúdo que é acessado a cada instante.

O uso maciço da internet pela população traz inúmeras possibilidades de ataques maliciosos, com exploração constante de novas técnicas aproveitando as vulnerabilidades inerentes ao ambiente compartilhado da internet. Um tipo de componente amplamente usado para mitigar vulnerabilidades em uma dada aplicação web é um *Firewall* de Aplicação Web (em inglês: *Web Application Firewall*, comumente abreviado como WAF) - responsável por monitorar tráfego de internet que é conduzido pela aplicação, permitindo ou bloqueando pacotes que possam ser potencialmente maliciosos. E recentemente, com o advento de técnicas de Aprendizado de Máquina, uma série de *firewalls* de Aplicações Web baseados nesse subcampo de Inteligência Artificial foram criados com o intuito não apenas de explorar a área em si, como também de gerar *Firewalls* mais robustos e capazes de deter tráfego de ataques web.

Firewalls propriamente ditos são softwares ou até mesmo hardwares que são responsáveis por examinar e filtrar a informação que atravessa uma determinada conexão. WAFs monitoram dados que uma aplicação web recebe, permitindo averiguar suas vulnerabilidades. Há várias maneiras de testar WAFs hoje quanto a eficácia, tanto manualmente através de documentos padronizados como o *wafec* (CONSORTIUM, 2006), como através de ferramentas como o *GoTestWAF* (WALLARM, 2022). Mas nem toda ferramenta de teste é capaz de avaliar e/ou realçar a eficácia dos WAFs baseados em Aprendizado de Máquina para a filtragem de informações.

Uma das ferramentas com a capacidade de testar WAFs baseados em Aprendizado de Máquina é o *WAF-A-MoLE* (VALENZA et al., 2020) que permite o uso de técnicas (detalhadas no Capítulo 2 adiante) de *fuzzing* (ZELLER et al., 2022) para explorar as fraquezas de um WAF e com isso apontar caminhos para tornar os ambientes mais robustos.

Essa trabalho procurou complementar a ferramenta *WAF-A-MoLE* em uma série de falhas verificadas pelos autores - uma arquitetura engessada que dificultava o uso e reprodução da ferramenta, uma série de modelos de ataque faltantes que poderiam ser utilizados porém não foram, e uma lista de operadores de mutação (um aspecto chave do seu funcionamento, explicado mais adiante) que poderia ser ainda mais estendida.

Para mitigar essas dificuldades de desenvolvimento e uso, foram criadas várias ferramentas auxiliares que possibilitam a um contribuidor ou usuário treinar (e testar) um WAF baseado em aprendizado de máquina mais facilmente, uma interface para qualquer classificador da biblioteca `scikit-learn` [3] foi introduzida, e a eficiência do programa foi

[3] scikit-learn é uma biblioteca para a linguagem de programação *Python* especializada em análise de dados, com uma suíte de ferramentas de Aprendizado de Máquina.

incrementada com novos operadores de mutação.

Resumidamente, esse complemento é intitulado de *wafamole++*. É uma ferramenta em linha de comando que, com o recebimento de um modelo de um WAF baseado em Aprendizado de Máquina pelo usuário, gera o que é conhecido como exemplo adversarial. Exemplos adversariais para o *wafamole++* são entradas especializadas criadas com o intuito de confundir um WAF. Em Aprendizado de Máquina, eles possuem um significado mais amplo por confundir o componente conhecido como classificador, mais detalhado no Capítulo 2.

Nesse caso, a cada execução dessa ferramenta, gera-se uma entrada que o WAF considera como inócua, porém é uma entrada maliciosa que pode comprometer um sistema que esse WAF estaria a defender. A ferramenta é montada de uma maneira que o usuário possa adaptá-la para avaliar seu próprio WAF contra vulnerabilidades e então treiná-lo novamente com entradas maliciosas para que ele seja capaz de prevê-las melhor.

E nesse contexto, o trabalho está organizado em seis capítulos: uma revisão de literatura sobre a área e trabalhos relacionados é amostrada, traçando comparações com a ferramenta que foi escolhida como alvo de extensão/implementação do trabalho. Posteriormente, é feito um detalhamento sobre a implementação, arquitetura e conceitos por trás da extensão do *WAF-A-MoLE*, intitulada de *wafamole++*. Após isso, um compilado de experimentos responsáveis pela avaliação do *wafamole++*, e seus resultados. Finalmente, as conclusões acerca dos resultados, dificuldades encontradas e apêndices.

2 CONCEITOS BÁSICOS

Nesse capítulo busca-se elucidar as principais terminologias, técnicas, algoritmos e tecnologias no geral que estão relacionadas a esse trabalho, realizando-se um apanhado geral e incluindo alguns exemplos esclarecedores.

2.1 INJEÇÃO SQL

O tipo mais comum e mais prevalecente de injeção de código é o da injeção SQL (em inglês: *SQL Injection*), que manipula bancos de dados via SQL (abraviado do inglês: *Structured Query Language*), a linguagem clássica de manuseio de bancos de dados, para receber acesso a informações sensíveis.

Exemplificando algumas situações de injeções SQL (PORT..., 2022):

a) Consideremos uma aplicação que possui uma funcionalidade básica de login, com usuário e senha. Quando o usuário fornece as entradas `‘usuário’` e `’123’` como credenciais, a seguinte consulta SQL é realizada pelo sistema na checagem:

```
SELECT * FROM users
WHERE username = ‘usuario’ AND password = ‘123’
```

Caso o usuário e senha sejam iguais a um registro na base de dados, todas as colunas da tabela `user` serão retornados. Caso contrário, os dados não serão obtidos. A sequência de caracteres `--` representa comentário em SQL. Se a aplicação não tiver uma higienização dos dados, é possível inserir esses caracteres como entrada e eles serão processados como uma instrução na consulta. Nesse caso, se o atacante usa a sequência de comentário SQL `--`, ele é capaz de remover a checagem de senha da cláusula `WHERE`. Por exemplo, se no parâmetro que é o `username` da consulta for inserido `administrator’--` e uma senha qualquer, a seguinte consulta é processada:

```
SELECT * FROM users
WHERE username = ’administrator’--’ AND password = ’’
```

Como o comentário SQL `--` aponta para o sistema lendo a consulta SQL que tudo após ele é um comando comentado, a verificação de senha introduzida após a condição `AND` é burlada. Essa manobra não equivale a passar uma senha em branco

em um formulário, mas sim à pular a própria etapa de inserção de senha no banco. Isso efetivamente retorna os dados do usuário cujo `username` é `administrator` e realiza o login do atacante como tal, expondo funções sensíveis da aplicação web inteira.

b) Em uma aplicação de e-commerce que organiza produtos em categorias diferentes, podemos ter um caso onde o usuário clica em uma categoria `'Presentes'`, requisitando o seguinte URL do navegador web:

https://insecure-website.com/products?category=Presentes

Isso faz com que a aplicação realize uma consulta SQL para resgatar todos os detalhes (identificado por `*`) da tabela de produtos, onde a categoria é `'Presentes'` e a restrição `released` é 1:

```
SELECT * FROM products
WHERE category = 'Presentes' AND released = 1
```

A restrição, quando observada por um atacante astuto, se revela como algo usado para esconder produtos não lançados. Esse atacante pode então construir um ataque digitando a seguinte URL, dado que o site não previne contra ataques SQL:

https://insecure-website.com/products?category=Presentes'–

Que provoca a seguinte consulta SQL:

```
SELECT * FROM products
WHERE category = 'Presentes'--' AND released = 1
```

Uma vez que a sequência `--` anteriormente vista denota um comentário SQL, a parte depois de `'Presentes'` é interpretada como tal e não é executada, revelando todos os produtos escondidos.

c) Também é possível recuperar dados de outras tabelas de um banco de dados, utilizando a palavra chave `UNION` (MYSQL..., 2022), que combina o resultado de múltiplas consultas `SELECT` em apenas um conjunto. Se um atacante executa a seguinte consulta contendo a entrada de usuário `'Presentes'`, ainda no último exemplo de site:

```
SELECT name, description
FROM products WHERE category = 'Presentes'
```

Então ele pode também submeter uma entrada nociva com `UNION`:

```
' UNION SELECT username, password FROM users--
```

Tal entrada faz com que todos os usuários e senhas venham juntos das descrições dos supostos presentes, fundamentalmente comprometendo a segurança da aplicação

No contexto deste trabalho, é estudado o uso de WAFs, mais especificamente os baseados em técnicas de Aprendizado de Máquina, como forma de prevenir ataques de injeção SQL. O uso de *firewalls* para aplicações Web, tanto comerciais como de código aberto, tem tido amplo crescimento no mercado.

Além dos exemplos básicos de injeção SQL, existem variantes (BACH-NUTMAN, 2020) como: baseada em erro, baseada em união, às cegas e fora de banda. e *out-of-band* (INVICTI..., 2022). Baseadas em erro são injeções que fornecem erros de SQL para o atacante, vazando informações críticas do banco de dados. Baseadas em união são ataques que utilizam uma consulta existente na lógica do sistema concatenada a outra através de um comando `UNION` de maneira a executar uma segunda lógica além daquela já estabelecida pelo sistema. Injeções às cegas (em inglês: *blind-based*) são aquelas que o sistema não dá nenhuma informação após a injeção, de forma que o atacante precisa ser engenhoso para obter informações e prosseguir com a ofensiva. Uma injeção SQL fora de banda é uma injeção sem o fornecimento de uma saída para o atacante, porém é possível tipicamente redirecionar a saída para um ponto como um servidor http. É fundamental para mitigar as injeções ter uma validação dos caracteres e declarações válidas ao receber uma entrada de dados na aplicação web.

Injeção (em especial SQL) é uma das vulnerabilidades mais comuns, de baixa complexidade para o atacante e historicamente impactante, o que motivou o trabalho em aprofundar em um problema subestimado por desenvolvedores, que tem impactos profundos na confiabilidade, integridade e disponibilidade de um serviço web.

2.2 CROSS SITE SCRIPTING, OU XSS

Cross Site Scripting, também conhecida como XSS, é uma técnica onde geralmente há uma injeção de *scripts* maliciosos na aplicação web, revelando dados sensíveis, serviços internos ou divulgação de *cookies* de usuários com privilégios. Existem três variantes de XSS. A primeira, XSS Armazenado, onde o atacante injeta uma entrada no servidor web, de maneira que quando outro usuário faz a requisição para acessar a página, essa entrada é ativada. A segunda, intitulada de XSS Refletido, ocorre quando o atacante injeta dados através de um método HTTP gerando uma resposta imediata na aplicação. Este permite o

malfeitor forjar uma URL específica que ao ser acessada pela vítima vaza dados sensíveis, como *cookies*. E por último a variante XSS baseada em DOM que consiste na inserção de código malicioso no DOM (*Document Object Model*), sem refletir no código-fonte HTML, sendo ativado apenas pelo console DOM. Dados obtidos de fontes não confiáveis devem ser devidamente validados e normalizados para evitar XSS.

2.3 ALGORITMOS DE FUZZING

O termo *fuzz* foi cunhado pelo professor da Universidade de Wisconsin-Madison Barton Miller nos anos 80, após sofrer uma interferência por uma tempestade de raios atuando no funcionamento de aplicações que rodavam em um ambiente *unix* remoto na época (FUZZING. . . , 2022).

Pouco depois disso o mesmo passou aos seus alunos uma tarefa denominada o *"Fuzz Generator"*, na qual era necessário implementar uma ferramenta que testasse a robustez de programas *unix* através de um bombardeio de informações aleatoriamente geradas.

Atualmente é uma técnica amplamente aceita na testagem/sondagem da segurança de diversas aplicações, com um leque de ofertas de *fuzzing* comerciais no mercado. E naturalmente é empregável em WAFs, permitindo extrair informações valiosas que rendam aprimoramentos para os mesmos. Enquanto a ideia em si de *fuzzing* permanece a mesma, as maneiras de efetuá-lo tiveram um grande progresso de mudança.

Uma função elementar de *fuzzing* pode ser vista no Código 1, que constrói, a partir de tamanho máximo e caracteres no intervalo fornecidos pelo usuário, uma *string* aleatória que pode ser chamada de *string* "fuzzeada". Usa-se o pacote `random` para lidar com a aleatoriedade dos pedaços da *string* que são construídos iterativamente.

Essa *string* serviria como entrada de um programa a ser testado, a fim de quebrá-lo e expor falhas/vulnerabilidades em sua implementação. Dependendo da complexidade do programa, pode servir como entrada tanto do fluxo de execução principal do programa ou em uma função menor do mesmo, para testar partes menores e afins. Não se usa sempre a mesma *string* necessariamente, sendo comum a geração de várias delas para se ter uma testagem mais profunda.

Código 1 – Função de um *fuzzer* elementar

```
import random

def fuzzer(max_length: int = 100, char_start: int = 32, char_range: int
    = 32) -> str:
    """String com um tamanho `max_length` de caracteres
       no intervalo [`char_start`, `char_start` + `char_range`)"""
    string_length = random.randrange(0, max_length + 1)
    out = ""
    for i in range(0, string_length):
        out += chr(random.randrange(char_start, char_start + char_range)
           )
    return out

fuzzer()
# A saida e a string aleatoria: '!7#%"*#0=)$;%6*;>638:*>80"=</>(/*:-(2<4
    !:5*6856&?""11<7+%<%7,4.8,*+&,,$,."'
```

Tais funções de *fuzzing* podem ser aprimoradas e customizadas para testar os mais diversos programas. Uma das maneiras de polir os resultados oriundos de *fuzzing* elementar, que podem ser rejeitados facilmente por muitos programas inicialmente, é um processo chamado *fuzzing* mutacional (em inglês: *mutational fuzzing*), projetado para melhorar as chances de obter entradas maliciosas mas que são consideradas válidas pela aplicação testada.

Na estratégia de *fuzzing* mutacional, a execução se inicia com uma entrada válida, e ela subsequentemente sofre uma mutação pequena, como no caso de uma *string* a mudança de um caractere, adição/remoção de um número, ou até mesmo uma troca de um bit (todos naturalmente aleatórios, pelo princípio do funcionamento de *fuzzing*).

Código 2 – Classe de *fuzzer* mutacional

```
import random

class MutationalFuzzer():
    def flip_random_character(s):
        """Retorna s com um caractere aleatorio com seu bit modificado."
           ""
        if s == "":
            return s

        pos = random.randint(0, len(s) - 1)
        c = s[pos]
        bit = 1 << random.randint(0, 6)
        new_c = chr(ord(c) ^ bit)
        # print("Flipping", bit, "in", repr(c) + ", giving", repr(new_c)
           )
        return s[:pos] + new_c + s[pos + 1:]

    def delete_random_character(s):
        # Modifica s de forma que um caractere aleatorio seja deletado.
        return s

    def insert_random_character(s):
        # Modifica s de forma que um caractere aleatorio em inserido
           aleatoriamente
        # na string s.
        return s

    def mutate(self, s: str) -> str:
        """Retorna s com uma mutacao aleatoria aplicada"""
        mutators = [
            self.flip_random_character,
            self.delete_random_character,
            self.insert_random_character
        ]
        mutator = random.choice(mutators)
        return mutator(s)
```

O programa no Código 2 realiza, dentre os três métodos operadores de mutação acima do método `mutate()`, uma mutação aleatória ao chamar `mutate()` propriamente dito. Realizar essa mutação consiste em executar um dos métodos (`insert_random_character()`, `delete_random_character()`, ou `flip_random_character()`) na entrada dada pelo usuário `mutate()`.

Um desses métodos foi exemplificado e está comentado no código (`flip_random_character()`), e os demais são simples o suficiente para depreender o

funcionamento dos comentários e nome do método.

Na prática, são realizadas uma série de mutações em cadeia, com o resultado de uma mutação servindo de entrada para a seguinte, para que se produzam entradas viáveis. Com um simples laço `for` ou `while` para chamar o método `mutate()` isso é possibilitado.

As técnicas de *fuzzing* geralmente encontram erros de corrupção de memória simples, mas a aleatoriedade e dispersão fazem com que a eficiência do ataque seja baixa, cobrindo poucas partes do código. O *fuzzing* aumenta seu potencial quando o alvo está em execução em um ambiente real. Outro ponto positivo é o fato de poder aplicar a técnica em aplicações sobre as quais há pouco conhecimento prévio.

Em suma, *fuzzing* consiste na geração massiva de entradas normais e anômalas para uma aplicação específica, seguido da detecção de exceções no algoritmo devido o processamento das entradas geradas e monitoramento dos estados de execução da aplicação. Quando comparado com outras técnicas como análise estática, dinâmica e execução simbólica, o *fuzzing* apresenta facilidade em ser implementado, boa precisão (em ambientes reais) e boa escalabilidade, sem acesso ao código fonte.

Expandindo um pouco o conceito, um teste de *fuzzing* tipicamente consiste na geração em larga escala de entradas para um programa específico, entradas conhecidas também como casos de teste. A qualidade de tais casos de teste criados influencia diretamente na qualidade do teste *fuzzing*. As entradas precisam atender os requisitos do programa testado para o padrão de entrada. Em contrapartida, as entradas precisam ser errôneas o suficiente para gerarem falhas ou erros no processamento do programa. De acordo com o alvo, as entradas podem ser arquivos com diferentes formatos, dados de comunicação entre redes, binários executáveis específicos, entre outros. No caso do *wafamole++*, que é a aplicação alvo deste trabalho, trabalha-se com injeções SQL, nas quais é aplicada a técnica de *fuzzing* iterativamente a fim de gerar uma entrada que consiga passar despercebido por um WAF avaliado, e seja malicioso. Essa entrada é o exemplo adversarial anteriormente mencionado.

Gerar casos de teste suficientemente errôneos é uma tarefa árdua, com duas variantes: *fuzzing* baseado em geração, e *fuzzing* mutacional, das quais a última é a adotada pelo *WAF-A-MoLE* e *wafamole++*. Depois de gerados, os casos de teste alimentam o programa alvo, tendo variáveis de ambiente e outros parâmetros do software a ser testado devidamente configurados pelo programador. Geralmente um teste de *fuzzing* tem como condição final um tempo limite pré-definido ou uma falha de execução do programa.

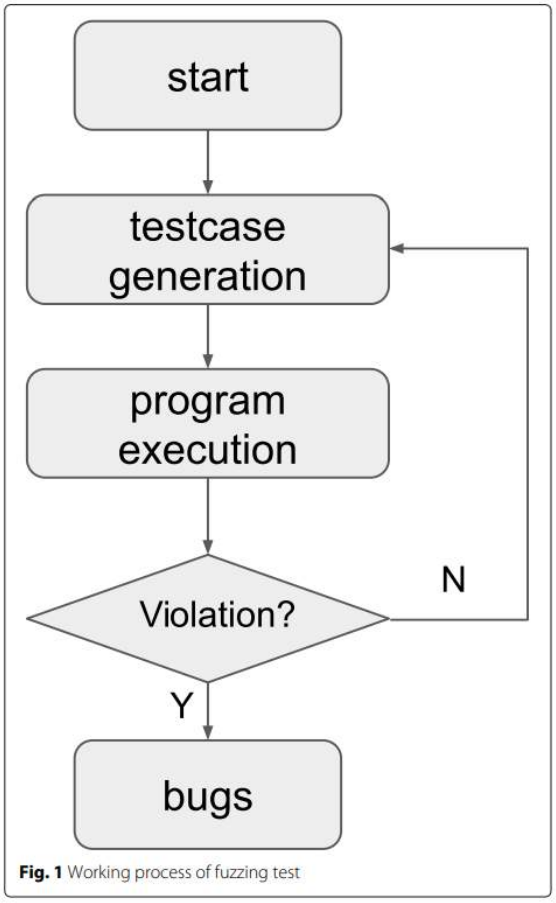
Fuzzers monitoram o estado de execução durante o processo, aguardando exceções ou falhas. Formas comuns de metodologia de monitoramento de exceções incluem monitorar sinais específicos do sistema, falhas e outras violações. Quando uma violação é capturada, o *fuzzer* armazena o caso de teste que a causou para análise futura e replicação (LI; ZHAO; ZHANG, 2018).

Por fim, na fase de análise, tem-se por objetivo determinar a localidade e a causa da

violação no software. A análise geralmente é feita com o auxílio de *debuggers* como `GDB` (GNU, 2021), `windbg` (MICROSOFT, 2022) ou outras ferramentas de análise binária, como o `IDA Pro` (RAYS, 2022) e outras. O exato estado de execução dos casos de teste, informações da *thread*, instruções e outras informações fazem parte do monitoramento. Para o caso do trabalho em questão, considerando-se o ambiente em *Python*, o *debugger* padrão `pdb` mostrou-se suficiente para diagnosticar a maioria dos erros encontrados.

A Figura 1 ilustra uma visão de alto nível de um algoritmo de *fuzzing*. Um caso teste é gerado, executado no programa, e caso o programa emita problemas (*violation* no fluxograma), significa que existem erros a serem corrigidos. Caso contrário, mais casos teste são gerados até que erros sejam encontrados ou até o usuário parar a execução.

Figura 1 – Workflow Fuzzing



Fonte: Li, J: Fuzzing - A Survey (2018)

O *fuzzer* baseado em geração necessita saber como é a entrada que o programa é capaz de processar. No caso de um *fuzzing* para arquivos, geralmente é fornecido um arquivo de configuração com o modelo pré-definido de entrada. Os casos de teste são gerados de acordo com o arquivo de configuração. Fuzzers desse tipo tem facilidade em burlar validações do software testado, permitindo ao desenvolvedor entender com mais profundidade a

aplicação estudada. Por outro lado, sem uma documentação amigável, analisar o formato do arquivo pode ser uma tarefa árdua. Em termos de praticidade, *fuzzers* mutacionais são mais fáceis de utilizar, com aplicabilidade genérica, sendo geralmente o tipo adotado por testes de intrusão no estado da arte. Diferente dos *fuzzers* baseados em geração, o *fuzzer* mutacional precisa de algumas entradas iniciais válidas. As entradas vão sofrendo mutações e novos casos de teste vão sendo gerados. O *wafamole++* requer apenas uma entrada inicial válida - uma consulta que é fornecida pelo usuário no início da execução do programa.

O contexto relevante para o *WAF-A-MoLE* é produzir, iterativamente, múltiplas injeções SQL a serem testadas a cada "rodada"de mutação. Uma injeção SQL é usada como base inicial, e a mesma sofre mutações por um *fuzzer* dedicado até ser considerada válida pelo WAF testado, embora seja no fundo ainda uma injeção maliciosa. Dessa maneira, o *wafamole++* pode ser resumidamente descrito como um *fuzzer* mutacional, operando em cima de WAFs com entradas de injeções SQL "fuzzeadas".

2.4 APRENDIZADO DE MÁQUINA

Tendo seus primórdios em torno das décadas de 60 e 70, com dispositivos como o *Cybertron* intitulados de "máquinas de aprendizado", esse subcampo de Inteligência Artificial teve uma tremenda ressurgência com recentes avanços em capacidade de processamento em tempos atuais. Com isso, dentre as áreas de interesse de computação enriquecidas por técnicas, naturalmente foram investigadas novas empreitadas em segurança da informação.

A ideia por trás de Aprendizado de Máquina (DANTAS, 2021) é o ensino/aprendizado de computadores por meio de exemplos - diferente da automação usual que envolve o programador elaborando algoritmos envolvendo manipulação de dados de entrada e saída desejados. Nesse contexto, um típico objetivo é rotular dados novos em categorias definidas nos dados iniciais. Para tal, temos dois principais procedimentos:

a) **Aprendizado supervisionado** - em casos nos quais dados de entrada e saída de um algoritmo desse tipo são conhecidos, e o algoritmo resultando é aprimorado com base nisto.

b) **Aprendizado não-supervisionado** - casos nos quais os dados de saída não são conhecidos, sendo possível identificar padrões no próprio conjunto de entrada da aplicação.

Alguns termos de relevância dentro de tais procedimentos podem ser elencados:

a) **Classificação** - É a ação qualitativa de rotular um dado avaliado com uma categoria, feito com um algoritmo de um classificador. Aplicando a um exemplo direto de Segurança da Informação, um classificador que rotula um pacote como "malicioso"realiza essa ação.

b) **Treinamento** - Essa ação envolve o fornecimento de exemplos para um determinado classificador, para que esse programa possa realizar a eventual classificação de dados vindouros. Com dados de texto, para o fornecimento desses exemplos ao classificador é comum passar os dados brutos por um algoritmo conhecido como vetorizador, que os transforma em vetores numéricos que serão lidos devidamente ao longo do treinamento.

c) **Dados de Teste** - São dados usados para a avaliação de um classificador realizada logo após seu treinamento, para que seja possível averiguar a eficácia e corretude do mesmo. É ideal ser uma porcentagem menor do que a de dados de validação, correspondendo usualmente em torno de 15% a 30% do conjunto de dados.

d) **Dados de Validação** - Durante o treinamento, são separados dados para a orientação do mesmo, com uso de uma parcela dos dados originais. É importante que esta parcela seja distinta dos dados de teste, para que o algoritmo resultante possa ter seu desempenho eventualmente testado contra dados completamente novos.

Sumariamente, o conhecimento de Aprendizado de Máquina traz abordagens de automatização relevantes para o contexto deste trabalho, pois as aplicações de segurança usam diversos algoritmos oriundos desse subcampo de Inteligência Artificial.

Algoritmos que vêm por padrão no *WAF-A-MoLE* em modelos de exemplo são *Naive Bayes*, Floresta Aleatória e Máquina de vetores de suporte Gaussiana/Linear, ofertados pela biblioteca *scikit-learn*.

2.4.1 Máquina de vetores de suporte

Máquinas de vetores de suporte, popularmente abreviados como SVMs (do inglês *Support Vector Machines*) são modelos de aprendizado supervisionado (dentro do contexto de teoria de Aprendizado de Máquina) associado a algoritmos de aprendizado que analisam dados para classificação e análise regressiva. É um dos métodos de predição mais robustos, baseado em arcabouços de aprendizado estatístico ou na teoria de Vapnik e Chervonenkis (comumente abreviada como teoria VC) (CHAPELLE; HAFFNER; VAPNIK, 1999) (BEN-HUR et al., 2001).

Dado um conjunto de exemplos para treinamento, onde cada exemplo é marcado em uma de duas categorias, um algoritmo de treinamento SVM constrói um modelo que associa novos exemplos para uma categoria ou outra, sendo um classificador binário linear não-probabilístico.

O SVM mapeia exemplos de treinamento para pontos no espaço de maneira a maximizar a distância entre as duas categorias. Novos exemplos são então mapeados no mesmo espaço. Em seguida, é feita uma predição sobre qual categoria o exemplo novo está associado.

Quando os dados ainda não foram rotulados, o aprendizado supervisionado não é possível e é necessário uma abordagem não-supervisionada, que tenta encontrar *clusters* naturais para o grupo de dados e então mapear os novos dados nos grupos formados.

O algoritmo *SV clustering*, criado por Vladimir Vapnik (BEN-HUR et al., 2001) com outros parceiros na AT&T Bell Laboratories, aplica a teoria estatística aos vetores de suporte, permitindo categorização de dados não-rotulados.

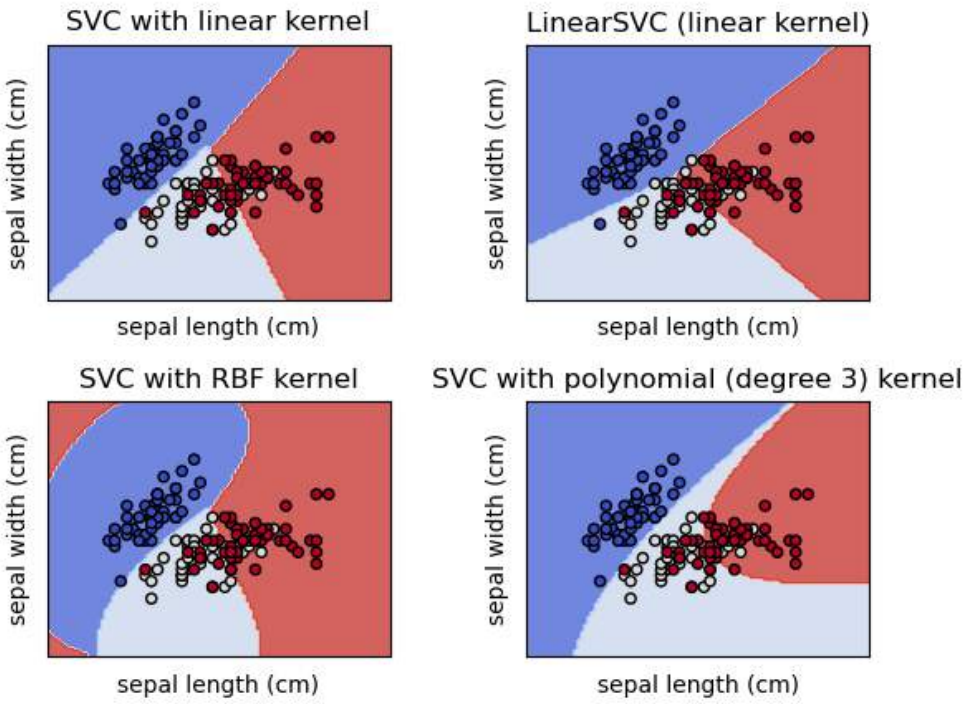
Python possui no pacote *sklearn* uma classe `SVC` (abreviação para *Support Vector Classifier* em inglês, ou classificador de vetor de suporte) e variantes com eficácias distintas em cenários diferentes.

Além do desempenho como um classificador linear, SVMs conseguem efetuar classificações não-lineares utilizando um mapeamento implícito de entradas em espaços multidimensionais de maneira eficiente. O novo modelo proposto *wafamole++* usa a versão não-linear, diferente da proposta no artigo original de Vladimir (BEN-HUR et al., 2001). Essa escolha é feita com base em experimentação de melhores resultados e uma otimização chamada *GridSearchCV*, que sumariamente testa exaustivamente combinações possíveis dentro uma série de valores diferentes fornecidos para cada parâmetro de um classificador.

Tomando um exemplo abstrato, com um classificador que possui um parâmetro determinado parâmetro chamado de **parametro_a** (que pode assumir um valor numérico de 0 até 150), e um outro chamado de **parametro_b** com um intervalo parecido. Para seu treinamento, pode-se fornecer à rotina *GridSearchCV* o classificador e um conjunto de alternativas como 1, 10 e 100 no lugar desses parâmetros, e o treinamento do classificador será feito testando cada uma das possíveis combinações entre os conjuntos fornecidos para os parâmetros, sendo indicado no final qual seria o mais próximo do ideal. Essa otimização é também explorada mais adiante no Capítulo 4.

A Figura 2 ilustra visualmente as diferenças ao variar o parâmetro do *kernel* em um classicador SVM na biblioteca *scikit-learn*. O *kernel* RBF (*Radial Basis Function*) e polinomial de grau 3 correspondem aos não lineares, com o RBF sendo utilizado no *wafamole++*. As zonas pintadas de azul e vermelho indicam elementos pertencentes a uma determinada predição feita pelo classificador, referente aos atributos de sépala de plantas na base de dados Íris (comumente usada no ensino e demonstração de Aprendizado de Máquina).

Figura 2 – Gráfico com quatro variantes do SVC



Fonte: (SCIKIT-LEARN. . . , 2022) (2022)

2.4.2 Gradiente Descendente Estocástico

O Gradiente Descendente Estocástico (ou *Stochastic Gradient Descent*, em inglês), comumente abreviado como SGD, é um método iterativo para otimização de uma função objetivo com propriedades "suaves"(diferenciais ou subdiferenciais) adequadas (RUDER, 2016a).

Pode ser considerada uma aproximação estocástica da otimização por gradiente descendente, uma vez que substitui o gradiente atual (calculado do conjunto de dados inteiro) por uma estimativa do mesmo calculada a partir um subconjunto dos dados selecionado aleatoriamente.

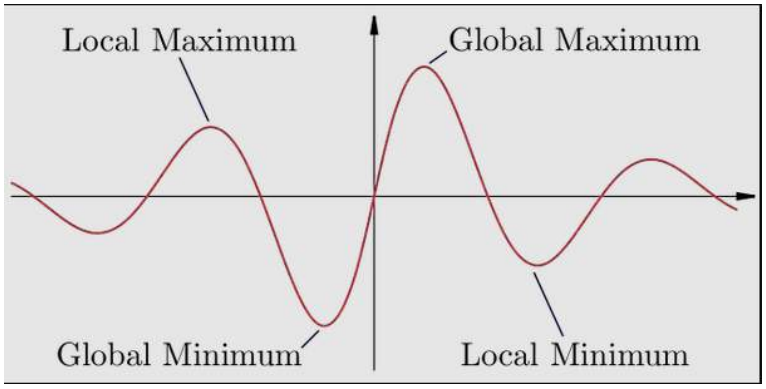
Em problemas de otimização com muitas dimensões isso reduz o custo computacional, tendo resultados mais rápidos com o *tradeoff* de perda no que é chamado de taxa de convergência. Para melhor compreender esta taxa: no aprendizado supervisionado, uma função de perda é definida, possuindo um valor mínimo global que é buscado através do gradiente descendente. O quanto se aproxima desse valor com cada etapa/iteração do algoritmo é a taxa de convergência.

O SGD procura encontrar o mínimo global de uma função através de ajustes de configuração a cada tentativa. A Figura 3 mostra um exemplo dessa função, com valores mínimos/máximos globais e locais. Ao invés de reduzir o erro ou achar o gradiente para o

conjunto de dados inteiro, o método reduz o erro através de aproximações em direção ao gradiente para um subconjunto aleatório do conjunto de dados, que pode ser tão pequeno quanto o conjunto de dados de treinamento inicial.

De forma heurística, se o modelo inicial erra muito, as alterações na configuração vão tornando o modelo mais preciso. Em contrapartida, é possível que questões antes acertadas, possam passar a serem classificadas como erradas ou pode acontecer um aumento da taxa de erro geral em favor de acertos específicos. Nem toda iteração necessariamente irá fazer o modelo melhorar. Positivamente, o SGD consegue sair do mínimo local em direção ao mínimo global através dos ajustes de configuração com baixo custo computacional em relação ao gradiente descendente clássico (RUDER, 2016b).

Figura 3 – Função Objetivo



Fonte: deepai.org (2022)

No *wafamole++*, um dos modelos novos é baseado em um classificador SGD, com seu funcionamento detalhado no Capítulo 4 e resultados demonstrados no Capítulo 5 após experimentos realizados.

2.4.3 AdaBoost

O método de *Boosting* vem da análise teórica do modelo de aprendizado chamado PAC (*Probably Approximately Correct*). Os autores Kearns e Valiant propuseram os conceitos de aprendizado forte e fraco. No modelo de aprendizado do PAC, se um algoritmo de aprendizado polinomial para identificar um grupo de conceito tem uma acurácia de reconhecimento muito alta, esse algoritmo é considerado como aprendizado forte. Porém, se a taxa de corretude do algoritmo de aprendizado de identificação é um pouco superior à tentativa aleatória de adivinhar, então esse grupo é considerado aprendizado fraco.

Kearns e Valient perceberam uma relação de equivalência entre aprendizado fraco e forte, onde o algoritmo de aprendizado fraco pode ser promovido para aprendizado forte, na qual a combinação dos resultados de uma série de algoritmos de aprendizados fracos

(como tentativas de adivinhação aleatórias) gera o resultado um algoritmo de aprendizado forte. O requisito para isso é saber de antemão o algoritmo forte em específico.

Em 1990, Schapire publicou o primeiro método de *Boosting* (SCHAPIRE, 1990). *Boosting* opera produzindo uma série de classificadores antes e depois do treinamento, que compartilham certos resultados durante o treinamento geral. O conjunto de treinamento utilizado em cada classificador é um subconjunto advindo do conjunto geral de treinamento e se cada amostra aparece no subconjunto ou não depende da performance dos classificadores anteriores. No caso das amostras julgadas como incorretas pelos classificadores produzidos anteriormente, a probabilidade de surgirem no novo subconjunto de treinamento será muito maior, fazendo com que os classificadores seguintes tenham foco em lidar com as diferenças nas amostras. Esse tratamento das diferenças seria uma tarefa árdua sem os resultados dos classificadores gerados na execução (WU; ZHAO, 2011).

O método *Boosting* pode aumentar a capacidade de generalização de um dado algoritmo, mas o algoritmo precisa ter estabelecido o limite inferior da acurácia de aprendizagem de um classificador fraco, que na prática é uma tarefa difícil.

O *AdaBoost* é um algoritmo do tipo *Boosting*, concebido em 1995 e publicado em 1999 (FREUND; SCHAPIRE; ABE, 1999), com capacidade auto-adaptativa que melhora a performance de classificadores fracos por estabelecer um conjunto de múltiplos classificadores.

Sendo o algoritmo mais representativo da família, ele mantém a distribuição de um conjunto de probabilidades para amostras de treinamento e ajusta a probabilidade de distribuição de cada amostra durante a iteração. Um algoritmo específico de aprendizado é usado para gerar um dos determinados classificadores gerados (dentre os vários anteriormente mencionados) e calcular a taxa de erro nas amostras de treinamento. O *AdaBoost* irá usar a taxa de erro para ajustar a probabilidade de distribuição das amostras de treino. O papel de mudar os pesos é para aumentar o impacto das classificações incorretas e reduzir o impacto das classificações corretas. Por fim, com o ajuste dos pesos nos classificadores únicos, é obtido um classificador forte.

Essas características o tornaram um bom classificador para experimentar e o *AdaBoost* eventualmente foi incorporado ao *wafamole++*, na forma de um dos modelos novos. Sua performance é detalhada no Capítulo 5, de experimentação, e detalhes da sua implementação podem ser encontrados no Capítulo 4.

3 REVISÃO DE LITERATURA

Em um campo com artigos, publicações e materiais em larga parte espalhados por blogs na internet, foi necessário realizar uma revisão de literatura para que as ferramentas de ponta, estratégias ainda sendo testadas e vulnerabilidades principais no mercado fossem corretamente identificadas. Isso se torna especialmente verdadeiro pela mudança no cenário de segurança dos anos 2000 para os anos 2010-2022, aonde boa parte de ameaças antigas podem ser de pouca relevância.

Neste capítulo estão alguns dos principais achados.

3.1 AVALIAÇÃO DE PROPOSTAS DE SEGURANÇA PARA APLICAÇÕES WEB

Ao fim da coleta de artigos, além do embasamento teórico e melhor enquadramento do escopo a ser estudado, buscou-se uma série de ferramentas e/ou casos reais de modelos e sistemas sugeridos na área de segurança da informação em sistemas Web, para que um desses pudesse ser expandido.

3.1.1 nscanner

Essa solução, publicada apenas na forma de um artigo (SURIAN; RAHMAN; NATHAN, 2020), provém de um estudo dirigido sobre uma série de outras ferramentas existentes hoje no mundo de segurança da informação (e de utilidade/notoriedade no mercado) como *Metasploit*, *Wapiti* e *Acunetix*. O nscanner agrupa essas ferramentas e usa cada uma delas em determinadas partes de seu funcionamento, assim ganhando relevância por ter uma funcionalidade mais ampla. Trata-se de uma aplicação dinâmica, flexível e automatizada para detectar os ataques mais comuns e problemáticos em segurança de Informação, que são injeções SQL e *Cross Site Scripting*. É capaz também de detectar malware via Aprendizado de Máquina para prevenir resultados que seriam falsos positivos ou negativos.

Há uma fase de testagem de vulnerabilidade em aplicação Web. Uma etapa independente permite o escaneamento de arquivos supostamente maliciosos fornecidos pelo usuário para detecção de malware. Todas as informações trabalhadas são armazenadas localmente, tornando desnecessário o uso de uma solução de Banco de Dados, que seria tipicamente usada com uma ferramenta dessa natureza.

Para a primeira fase, as seguintes etapas são observadas pelo autor:

a) Tendo sido estabelecida uma conexão com a internet, um usuário da ferramenta escolhe o tipo de vulnerabilidade que será testada (se é injeção SQL ou *Cross Site Scripting*). Após isso, ele fornece um endereço web ou de IP que armazena a aplicação que receberá os testes. Não é especificado pelos autores se há suporte a

todos os protocolos HTTP/1.1, /2, /3, ou HTTPS, porém idealmente a aplicação suportaria todos.

b) Uma requisição com os dados fornecidos pelo usuário é feita ao web *crawler* do Nscanner, um script responsável por acessar a aplicação em si e realizar a verificação por vulnerabilidades. Ele busca o site da aplicação pelo domínio, usado para identificar o servidor armazenando tal site.

c) Após isso, uma conexão com o servidor encontrado é requisitada pelo *crawler*, estabelecendo então uma ponte entre web *crawler* e servidor. Assim as páginas e componentes da aplicação em análise são navegadas e todas as partes com formulários são identificadas.

d) Nessa etapa (a mais importante da pipeline), os bugs são identificados pelo Nscanner. A ferramenta injeta entradas de acordo com a seleção de vulnerabilidade do usuário no início do fluxo, em todas as partes críticas (formulários) visualizadas na análise anteriormente feita a fim de verificar se alguma delas omite erros fundamentais deixando passar uma vulnerabilidade. Espera-se um tempo de processamento não desprezível por ser um processo iterativo. Finalmente é gerado um relatório que o usuário final pode optar por baixar para fins de histórico ou apenas visualizar no momento.

Enfim tem-se a segunda parte da proposta, de escaneamento de arquivos:

a) É feito um upload de um arquivo/amostra que o usuário considera malicioso ou suspeito ao Nscanner. Um critério para essa suspeita pode ser análise do código fonte desse arquivo que mostre etapas que possam comprometer uma aplicação, um pacote com conteúdo que pareça estar buscando se aproveitar de vulnerabilidades e assim por diante.

b) O principal componente da detecção de malware do Nscanner é acionado ao verificar que o upload foi bem sucedido, iniciando uma análise dinâmica. Por exemplo, um classificador de Aprendizado de Máquina baseado no algoritmo de *Random Forest*, treinado para detectar arquivos malware com uma série árvore de decisões geradas em seu treinamento pode ser usado. Verificando com sucesso algum caso em que a precisão seja suficiente para definir como malicioso, o arquivo é designado como tal e um relatório é gerado semelhantemente a etapa anterior para o usuário.

Embora seja um projeto conveniente, do qual a ideia de utilizar um classificador para detecção de vulnerabilidades pode ser muito aproveitada, percebem-se algumas falhas nessa implementação.

Em particular, na etapa mais importante do fluxo de detecção de vulnerabilidades em aplicação web, há pontos críticos que o autor não cuidou de verificar: não há uma proposta suficientemente concreta para fornecer entradas que testem a aplicação corretamente nesse fluxo. Este processo, que é bastante complexo, existe em algumas aplicações

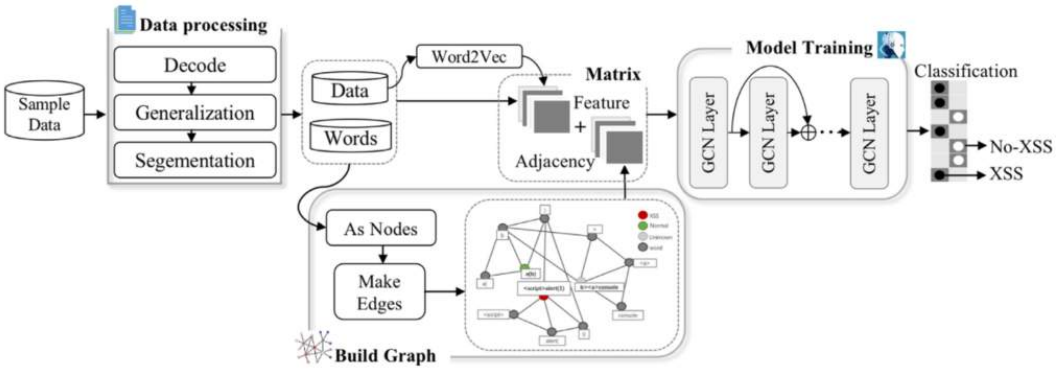
no mercado. Além disso, a diferença entre injeções SQL e *Cross Site Scripting* (XSS) não garante necessariamente que as vulnerabilidades aparecerão nos mesmos pontos críticos da aplicação. É frequente observar vulnerabilidades XSS em partes até impossíveis de se realizar injeções SQL.

Como esta proposta é apenas uma ideia sem implementação e considerando as questões críticas apontadas acima, foi descartada a expansão desta proposta dentro do escopo pretendido pelo trabalho.

3.1.2 GraphXSS

A proposta do GraphXSS (LIU et al., 2022) sugere uma implementação de uma Rede Convolucional de Grafos (em inglês: *Graph Convolutional Network*, abreviado como GCN) para realizar a detecção de ataques de *Cross-Site Scripting* (explicado anteriormente na Seção 2.2) em aplicações web. A *pipeline* básica envolve um pré-processamento dos dados, a composição dos mesmos em uma estrutura de grafos, e a utilização de algoritmos que realizam *Deep Learning* em cima dessa estrutura para enfim realizar a detecção de maliciosidade em uma determinada entrada. A solução é mostrada no diagrama da Figura 4:

Figura 4 – Arquitetura do Sistema GraphXSS proposto



Fonte: Z. Liu, Y.Fang, C. Huang et al via Scopus (2022)

O pré-processamento do bloco inicial *Data Processing* da arquitetura mostrada na Figura 4 tem por finalidade reduzir a parametrização excessiva, técnica na qual os parâmetros do algoritmo são mais numerosos do que os próprios dados de treinamento. Os autores acreditam que essa técnica possa interferir com os resultados finais de detecção do sistema.

Com a realização dessa etapa, também se obtém uma redução no custo computacional de subsequentes treinamentos efetuados nas etapas de classificação (justamente o momento mais exigente do sistema), melhorando também a eficiência do algoritmo. No bloco *Decode*, localizado após *Sample Data* na Figura 4, são utilizados algoritmos comuns

de encodificação/decodificação como primeira etapa, compreendidos por: *URL decoding*, *HTML entity encoding*, *Unicode decoding* e *Base64 decoding* para decodificação. Os resultados destes quatro algoritmos são generalizados (via discretização, generalização e/ou padronização de dados) e segmentados no Tokenizador de expressões regulares do módulo NLTK (*Nature Language Toolkit*).

Como mostrado na Figura 4, este pré-processamento dá entrada no módulo de composição para grafo, responsável por conectar palavras e dados de uma maneira regular para formar um grafo dotado de *data points* (no caso, um tipo de nó) e arestas. Esse grafo é formado para uso no módulo de *Deep Learning* (baseado em algoritmos de GCNs e mostrado no bloco *Model Training na Figura 4*) que otimiza seus parâmetros. Finalmente, após o treinamento inicial, obtém-se uma distribuição de probabilidade de cada vértice que levará à classificação do *data point* fornecido ao sistema.

Em comparação com as demais soluções levantadas, essa arquitetura não trabalha com injeções SQL em nenhuma forma, não oferece a usuários finais uma solução acionável e tem alta complexidade. Muitos dos detalhes da implementação do GraphXSS, com exceção dos diagramas da arquitetura geral e das saídas da etapa de classificação final, não foram encontrados nas referências, tornando também impossível o aproveitamento desta ferramenta para a extensão pretendida no trabalho realizado.

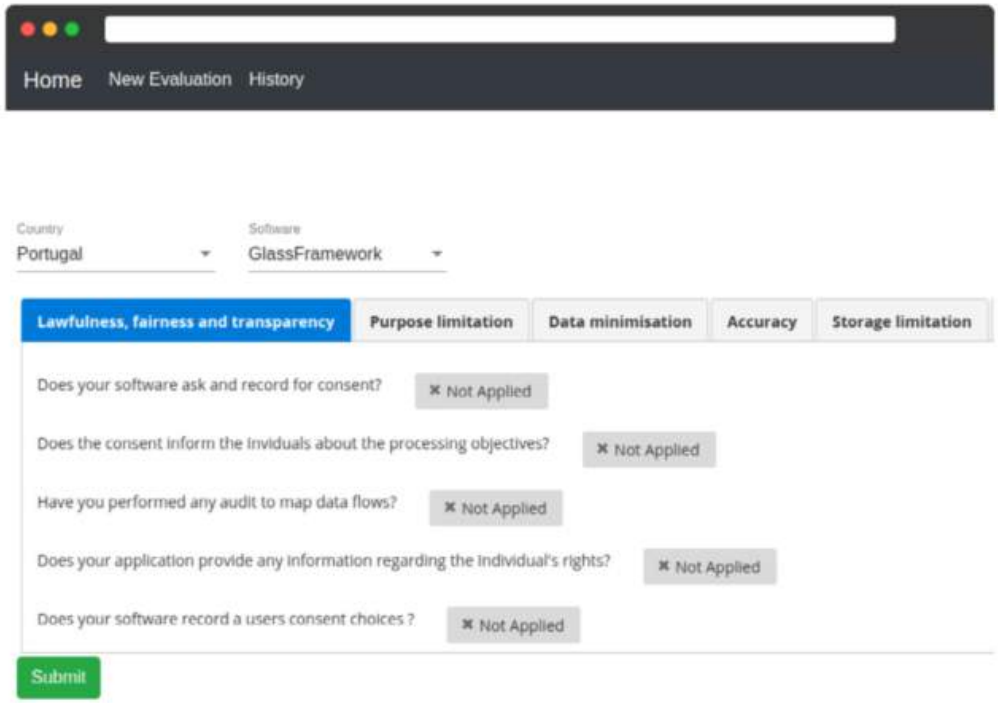
3.1.3 PADRES

O PADRES (PEREIRA; CROCKER; LEITHARDT, 2022), sigla para PrivAcy, Data REgulation and Security, é um software desenvolvido inicialmente para auxiliar a infraestrutura do EPOS (*European Plate Observatory System*) e a comunidade do GNSS (**Global Navigation Satellite System**) em um ponto pouco explorado em Segurança da Informação no contexto de Internet - complacência com a GDPR (*General Data Protection Regulation*), uma regulação introduzida em 2016 na legislatura Europeia com finalidade de proteger a privacidade e acesso aos dados alheios. Porém sua solução tornou-se generalizada o suficiente para auxiliar empresas de escopos variados, servindo para qualquer aplicação Web que seja aberta. A parte de interesse do PADRES é a funcionalidade embutida de analisar aplicações Web por vulnerabilidades, dado que privacidade de dados é comumente associada com segurança.

O fluxo básico dessa aplicação envolve o usuário fornecer um endereço de uma aplicação Web a ser analisada, seguido de uma série de perguntas sobre GDPR que são respondidas manualmente pelo administrador da mesma. Com isso, a aplicação é analisada pelo PADRES, e um relatório sobre suas vulnerabilidades e pendências com a regulação GDPR (baseadas nas respostas fornecidas no questionário manual) é disponibilizado para baixar.

A Figura 5 é uma captura de tela do PADRES em funcionamento, em sua tela inicial mostrando opções de seleção de país e de software para ser analisado.

Figura 5 – Screenshot do PADRES em funcionamento



Fonte: Fábio Pereira, Paul Crocker, Valderi R.Q. Leithardt et al via Scopus (2022)

Em termos de arquitetura, o PADRES é dotado de um modelo de Cliente e Servidor. Há um *front-end* desenvolvido no framework Angular (para JavaScript), e o *back-end* implementa uma arquitetura REST por meio do framework Flask, em *Python*. Um banco de dados preenchido de questões GDPR a serem respondidas é conectado ao *back-end*, e uma estrutura para armazenar metadados do relatório final em formato `.blob` é providenciada. Esse banco é modelado de maneira que as perguntas são extensíveis, tornando o software um bom candidato para expansão. Algumas ferramentas de código aberto de segurança da informação são utilizadas no módulo de detecção de vulnerabilidades Web, são elas: Wapiti, ZAP e NMAP. Elas são executadas pelo usuário via a interface gráfica, com suas saídas sendo acopladas à saída do questionário GDPR.

Todo esse arcabouço técnico é encapsulado pelo Docker, a principal ferramenta de conteinerização de software hoje no mercado, escrita em Golang. Através desse artifício, qualquer sistema operacional utilizando Docker consegue instalar o ambiente de desenvolvimento sem problemas e executar a aplicação.

Esse e o fato da extensibilidade da aplicação ser algo muito enfatizado pelos autores fez do PADRES uma opção inicial para a extensão pretendida pelo trabalho. Todavia, uma série de dificuldades imprevistas foram encontradas. Não só o repositório com a aplicação fornecida carecia gravemente de documentação, como a aplicação no estado atual não funcionou como previsto em uma variedade de ambientes testados. Apenas um esqueleto do *front-end* é corretamente disponibilizado, porém a parte da interface gráfica

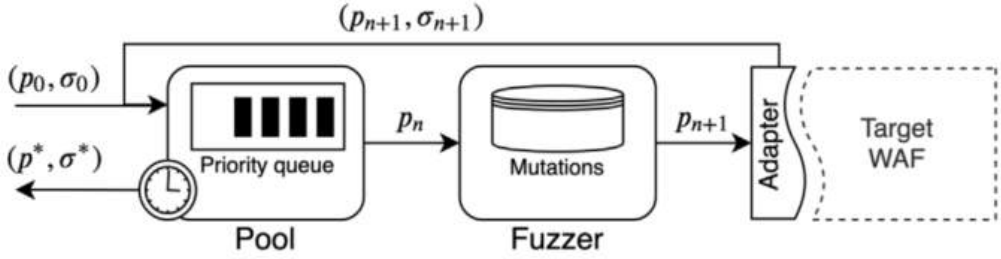
responsável por selecionar uma aplicação Web de fato encontrava-se impossível de ser selecionada. Algumas estratégias para contornar isso foram tentadas, porém sem sucesso. Não foi encontrada também uma forma de verificar o correto funcionamento do *back-end*.

Com isso o PADRES no estado atual não pode ser recomendado como uma ferramenta operacionalmente viável, mesmo que sua ideia resgate interesses de privacidade de alta relevância, até mesmo fora da Europa, e reforce a importância de documentação e testagem robusta de aplicações que sejam criadas com o intuito de reforçar software no geral.

3.1.4 WAF-A-MoLE

Desenvolvido como uma tese, o *WAF-A-MoLE* (VALENZA et al., 2020) é uma ferramenta geradora de testes para WAFs baseados em Aprendizado de Máquina. A versão atual é particularmente dedicada a injeções SQL. Sendo um dos poucos softwares encontrados nos artigos com uma implementação concreta (a outra sendo o PADRES, que mostrou-se defeituosa), o *WAF-A-MoLE* certamente foi um dos mais convidativos a colaboração. Não só os autores providenciam uma documentação robusta para uma aplicação inicialmente desenvolvida no meio acadêmico, como também mostram nela caminhos a serem expandidos por contribuintes futuros.

Figura 6 – Arquitetura do *WAF-A-MoLE* - p_i indica entrada (do inglês *payload*) i e σ_i indica pontuação (do inglês *score*) de confiança i.



Fonte: A. Valenza, L. Demetrio, G. Costa and G. Lagorio et al via arxiv (2020)

A arquitetura do *WAF-A-MoLE* é mostrada na Figura 6. Ela consiste do bloco `Pool`, uma fila com prioridade, do bloco `Fuzzer`, que implementa mutações, e da classe `Adapter`, comentados a seguir.

Inteiramente codificado na linguagem de programação *Python 3*, esse software faz uso de uma metodologia de testagem baseada em *fuzzing* (inspirada pelo *Fuzzing Book*) (ZELLER et al., 2022) para criar ataques que penetrem com sucesso um determinado WAF baseado em Aprendizado de Máquina a ser analisado. Mais precisamente, fazendo uso da pontuação de classificação do WAF como métrica, o processo de *fuzzing* é guiado de maneira que as entradas mais promissoras são priorizadas.

Ele é implementado como uma biblioteca e como uma ferramenta de linha de comando, por meio de *Click decorators* (uma função da biblioteca *Click* para minimizar o código por trás de aplicações de linha de comando). O usuário pode escolher qual modelo será utilizado, além de uma série de parâmetros como `timeout`, rodadas máximas [1], caminho para o arquivo de saída e afins.

Tomando como base a Figura 6 de sua arquitetura, seu funcionamento é iniciado com uma entrada (*payload*) inicial p_0 (fornecido pelo usuário ao executar o programa) ao qual o WAF a ser testado confere uma pontuação de confiança $\sigma_0 \epsilon [0, 1]$ que será inserido em `Pool`, uma estrutura contendo entradas. Essa estrutura `Pool` é uma fila de prioridade que armazenará entradas subsequentes por ordem decrescente de suas pontuações de confiança, de modo que as entradas com menor pontuação de confiança terão maior prioridade de saída.

Com isso, a cada iteração a cabeça da fila p_n é selecionada pela `Pool`, passada adiante para a classe `Fuzzer`, que realiza uma mutação em cima de p_n transformando-o em $p_{\mathrm{n+1}}$, aplicando aleatoriamente um dos operadores de mutação padrões do programa (A lista completa de operadores de mutação pode ser conferida nos apêndices). $p_{\mathrm{n+1}}$ é submetido então ao WAF alvo para classificação/avaliação e sua pontuação de confiança é gerada por uma classe `Adapter` após isso. A natureza extensível do *WAF-A-MoLE* permite que uma variedade de WAFs sejam testados desde que estejam com adaptadores específicos para garantir compatibilidade. A classe `Adapter` retorna então a pontuação de $\sigma_{\mathrm{n+1}}$ de $p_{\mathrm{n+1}}$, sendo colocado de volta na fila `Pool` de entradas para sofrer novas mutações nas próximas iterações.

Com esse funcionamento, a condição de parada é dada da seguinte forma:

a) No momento em que uma pontuação de confiança σ^* é menor do que um limite dado pelo usuário no início do programa (no parâmetro `threshold`) - retorna-se o melhor par (p^*, σ^*) encontrado;

b) Caso o número de iterações máximas seja atingido e nenhuma pontuação de confiança σ^* menor que `threshold` seja encontrada, o par (p^*, σ^*) mais próximo do mesmo é retornado;

c) O programa é interrompido via `SIGKILL`, `SIGTERM` ou semelhante.

Para testagem de WAFs de diferentes origens e arquiteturas, uma classe `Model` é disponibilizada pelos autores como uma interface, generalizando o comportamento de modelos usados. Assim, o componente amostrado na Figura 6 como *Adapter* ao lado de *Target WAF* pode ser criado a partir dessa classe. Ao adaptar um WAF para ser testado, o desenvolvedor precisa compará-las com os adaptadores dos modelos de exemplo do *WAF-*

[1] Rodadas máximas: Número máximo de iterações a serem realizadas

A-MoLE para saber quais características são usadas pelo mesmo no funcionamento de sua arquitetura.

Essa necessidade de experimentação para adaptação não se mostrou muito clara, porém foi possível (com a ajuda dos autores originais) construir um *wrapper*/interface geral para modelos do `scikit-learn` com o *wafamole++* - a extensão desse software melhor detalhada nos capítulos 4 e 5 deste documento.

3.1.5 Conclusão

Em suma, percebe-se que o *WAF-A-MoLE* é uma excelente ferramenta base para reforçar WAFs baseados em Aprendizado de Máquina. Configurando rotinas para gerar uma série de exemplos adversariais a partir do mesmo, especialistas de segurança podem refazer o treinamento de seus *Firewalls* complementando seus exemplos de base com os adversariais recém-gerados, chegando a níveis de robustez mais aceitáveis.

Entretanto, não é uma solução sem imperfeições, das quais podem ser destacadas:

a) Uma falha na documentação de suas dependências (em particular quanto à versões);

b) A arquitetura se mostrou engessada para adaptar para modelos novos, mesmo com incentivo pelos autores originais;

c) Muitos dos detalhes de treinamento de dados se mostraram ofuscados na documentação, juntamente de estruturas de dados usadas em seu funcionamento.

Por esses motivos foi decidida a extensão do mesmo, nomeada de *wafamole++*, a fim de amenizar as dificuldades encontradas na sua execução e enriquecer seu funcionamento para futuros desenvolvedores e profissionais de segurança.

4 PROPOSTA DE SOLUÇÃO - WAFAMOLE++

Tendo em consideração o embasamento teórico estabelecido e a oportunidade de contribuir para um projeto de código aberto na área de interesse deste trabalho que tivesse abrangência e impacto, foi concebido o *wafamole++* [1], uma versão estendida do *WAF-A-MoLE* anteriormente descrito no capítulo de Revisão de Literatura. Foram seguidas diretrizes e recomendações de contribuição estabelecidas pelos autores, e alguns módulos adicionais contendo funcionalidades não disponibilizadas pelos mesmos referentes ao treinamento de novos classificadores, geração de novos modelos, e tratamento de conjuntos de dados foram disponibilizados com este trabalho.

A elaboração do *wafamole++* forçou um entendimento mais profundo da ferramenta original, recebendo apoio dos autores originais e atualmente conta com uma série de operadores de mutação novos e sobretudo modelos de classificadores novos implementados para um WAF de código aberto encontrado na plataforma de colaboração GitHub.

Acredita-se que essa colaboração contribuiu para tornar a ferramenta mais abrangente, com uma menor barreira de entrada para futuros colaboradores e mais flexível para avaliação de WAFs baseados em Aprendizado de Máquina no geral. Assim espera-se que mais contribuições possam ser geridas no futuro, estabelecendo-se como um projeto complementar ao *WAF-A-MoLE* original.

4.1 VISÃO GERAL

A introdução do *wafamole++* como extensão do *WAF-A-MoLE* é, sumariamente, uma versão do mesmo com uma série de módulos adicionais. O ambiente original carecia de uma documentação para as versões de suas dependências, e documentação de todas as suas etapas de treinamento, ocasionando em uma série de dificuldades para seu uso esperado. Além disso, havia espaço para adição de mais operadores de mutação e uma série de modelos diferentes que não foram inicialmente explorados pelos autores.

O *wafamole++* resolve essas pendências introduzindo uma série de modelos e operadores de mutações novos, além de possuir uma documentação mais extensa (facilitando a instalação), com as etapas de treinamento transparentes o suficiente para que outros desenvolvedores possam replicá-las. Possui também como parte de suas extensões uma série de funções utilitárias para tratamento de dados, bastante importante para garantir a extensibilidade.

[1] Disponível no GitHub: https://github.com/nidnogg/wafamole-plusplus

4.2 ARQUITETURA

O *wafamole++* possui uma estrutura composta de um classificador para realizar avaliações em cima dos dados trabalhados, uma interface em linha de comando para o usuário parametrizar seu funcionamento, e uma classe que realiza as técnicas de *fuzzing* durante as rodadas de mutação.

Os operadores de mutação responsáveis por gerir as dadas iterações de *fuzzing* ficam, idealmente, todos agrupados na classe `SqlFuzzer` dentro de seus métodos.

Diferentemente do *WAF-A-MoLE* no entanto, a proposta inclui também um módulo separado responsável pelo tratamento, treinamento e geração de dados para alicerçar o funcionamento da aplicação principal.

Uma representação visual da proposta, sem especificações como linguagens usadas, extensões de arquivo e detalhes mais profundos pode ser conferida no diagrama da Figura 7. Sua versão com maior nível de detalhamento é disponibilizada na Figura 9, mais adiante na Seção de implementação.

Descrevendo de forma geral essa estrutura, cada retângulo com título indica um módulo, seus subtítulos representam submódulos/métodos contidos nesses, e setas indicam ações ou comandos realizados por essas estruturas. Resumidamente, detalhando do topo da arquitetura até a base, tem-se:

a) Um módulo `Database` - Nele se encontra um gerador de consultas `mole_query_generator` que executa o programa diversas vezes para acumular casos a serem adicionados na base de dados, um programa `sqli_cleaner` para tratamento de dados brutos e formatação para o WAF a ser testado, e os dados em si.

b) O MLBasedWafClassifier - Esse módulo responsável por receber os dados do módulo de `Database`, estruturar o classificador do WAF *MLBasedWAF* através de treinamento, gerar seu modelo correspondente e exportá-lo para uso na aplicação principal na parte de modelos.

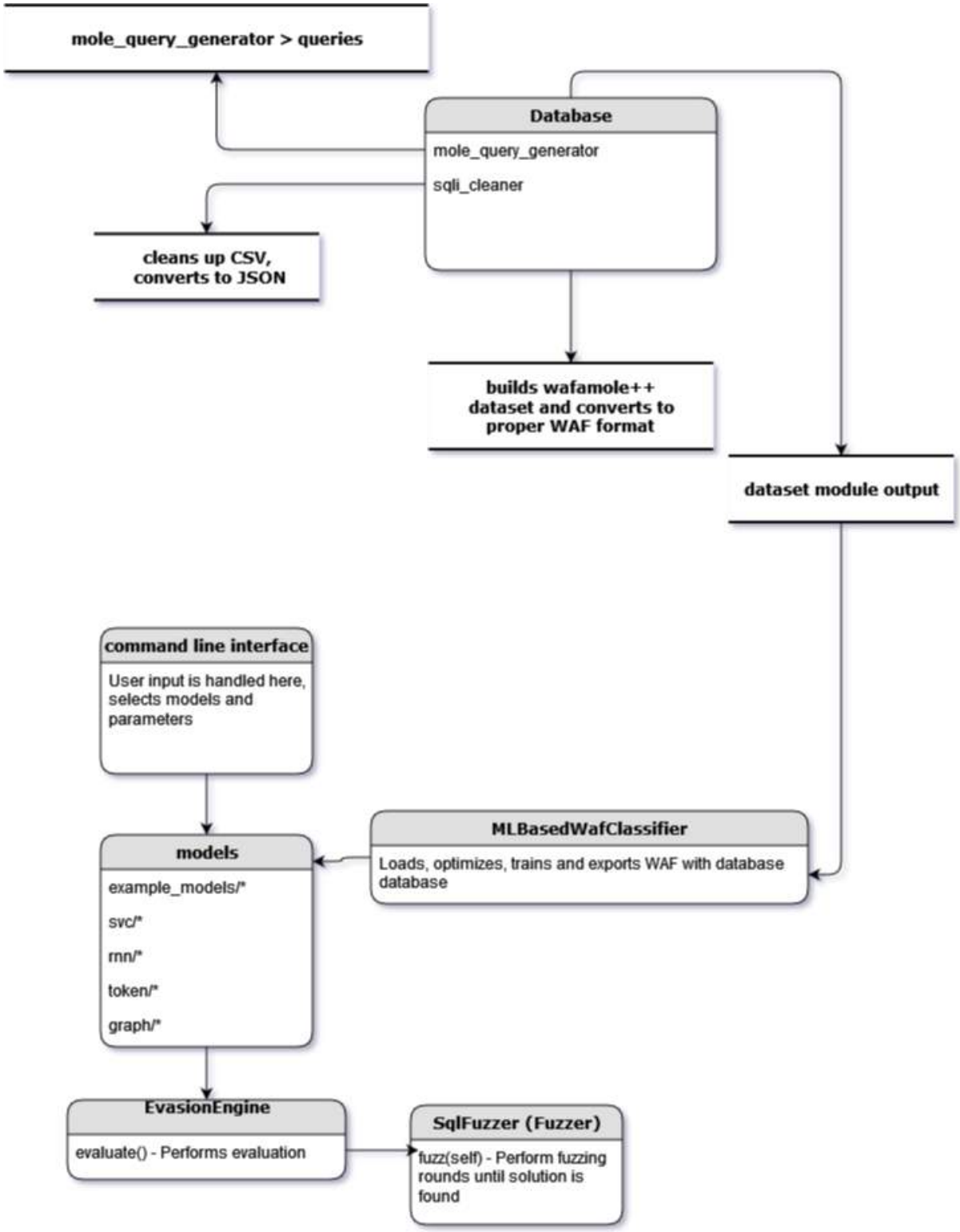
c) Command Line Interface, ou Aplicação de linha de comando - Aqui é abrigada a típica função **main**, sendo aonde são recebidos os parâmetros a serem executados na aplicação pelo usuário como o número máximo de iterações, modelo a ser usado, e injeção SQL inicial para ser transformada em um exemplo adversarial. Esses parâmetros são explicados na Seção adiante.

d) Models - Aonde são armazenados os modelos. Modelos são classificadores de WAFs já treinados, exportados de forma que não seja necessário realizar o treinamento a cada iteração.

e) EvasionEngine + SqlFuzzer - A classe `EvasionEngine` realiza cada iteração da aplicação que consiste em chamar a classe relacionada `SqlFuzzer` que transforma (via *fuzzing*) a entrada inicial com os operadores de mutação nela contidos, armazenar

a saída dessa transformação, e chamá-la novamente para obter uma transformação mais sofisticada. A condição de parada é fornecida pelo usuário, tanto na forma de iterações máximas, como na forma de uma *threshold* máxima de confiabilidade (mais explicada na Seção adiante).

Figura 7 – Proposta wafamole++



4.3 IMPLEMENTAÇÃO

A solução foi implementada na linguagem *Python* 3.8, seguindo as dependências encontradas nos arquivos intitulados `requirements.txt` no repositório final (disponível no GitHub [2]). O cerne do projeto encontra-se nos seguintes módulos:

a) `Main` - Responsável pela lógica da linha de comando, com todos os decorators da biblioteca `Click` que facilitam o desenvolvimento de utilidades de terminal. Aqui é feito o recebimento e passagem de parâmetros para a pipeline do programa aqui, além de ser instanciada a classe principal `wafamole` e o módulo `EvasionEngine` com o modelo selecionado pelo usuário.

b) `models` - Em models estão localizados os modelos de exemplo para experimentação, modelos padrões do *WAF-A-MoLE* e sobretudo modelos novos (também customizáveis) introduzidos no *wafamole++*, localizados na pasta `/models/svc`. Modelos novos de classificadores de WAFs a serem testados são tipicamente arquivos `.dump` das classes de tais classificadores, gerados através biblioteca `joblib` após o treinamento (no caso do `scikit-learn`, da função `.fit()`). Alternativamente, essa biblioteca permite também a geração de um arquivo `.dump` de uma pipeline caso o treinamento exija mais de uma etapa, como uma tokenização para possibilitar o treinamento de um dado classificador.

c) `EvasionEngine` - Essa classe contém o loop principal que requisita as mutações responsáveis pela transformação do *payload* fornecido pelo usuário em um *payload* considerado inócuo para o WAF sendo avaliado. As requisições são feitas ao módulo `Fuzzer`.

d) `Fuzzer` - Nesse módulo ficam armazenados os operadores de mutação que transformam o *payload* a cada rodada de mutação efetuada, sendo que cada mutação ocorre dentro do mesmo. Um operador de mutação dentre estes é escolhido aleatoriamente a cada iteração da pipeline principal.

Esses módulos, no entanto, não podem ser trivialmente estendidos sem alguns componentes auxiliares criados para o *wafamole++*, que são:

a) `datasets` - Aqui são contidos os conjuntos de dados SQLiV3, SQLiV4, e SQLiV5, funções de tratamento para os mesmos e uma função de geração/adaptação do conjunto de dados original do *WAF-A-MoLE* para servir de treinamento para o MLBasedWAF, um WAF que será descrito na seção seguinte. O `SQLiV3.json` é o primeiro conjunto de dados utilizado para tal, disponibilizado pelo Kaggle [3] (SAQLAIN, 2021) e também está descrito adiante. Versões subsequentes (indicadas

[2] GitHub - Repositório Final: https://github.com/nidnogg/tcc

[3] Kaggle - SQL Injection Dataset: https://www.kaggle.com/datasets/syedsaqlainhussain/sql-injection-dataset

pelos números ascendentes) são incrementos realizados e testados nele com reforços de *payloads* oriundos do wafamole++.

- `mole` - Essa função pode ser visualizada no Código 3 mais adiante. Ela executa em linha de comando o *wafamole++* para cada linha do conjunto de dados SQLiV3, de modo a gerar um leque mais amplo de ataques. Como se trata de um processo iterativo, um limite superior de 7000 novos *payloads* foi adotado - e o tempo de execução médio foi em torno de 8 horas para o processo todo. Para gerar novos ataques (dado que o *wafamole++* produz saídas diferentes a cada execução), basta executá-lo com redireção de saída padrão (`>`):

  ```
  ./mole_query_generator.py > output.json
  ```

- `sqli_cleaner` - Essa função auxiliar foi codificada inicialmente com a finalidade de transformar o conjunto de dados `SQLiV3.json` do formato .csv para .json que seria mais simples de trabalhar em *Python*, além de adaptá-lo para o padrão visto no *MLBasedWAF*. Essa função também contorna de uma série de inconsistências nos dados originais através de um polimento e tratamento de certos casos para que a saída do *wafamole++* seja ideal. Seu código está disponível nessa seção, mais adiante no Código 8..
- `wafamole_dataset_generator` - Fornecido apenas após um certo período de espera de alguns meses (compreendido entre a troca de emails com os autores originais), o conjunto de dados do *WAF-A-MoLE* encontra-se disponibilizado no seu repositório[4] de maneira modular - com uma série de arquivos `.json` pequenos devido ao limite imposto pela plataforma GitHub. Além disso, ele precisou ser adaptado para o formato do *MLBasedWAF* assim como o `SQLiV3.json` (sem dificuldades nesse caso). Possui uma abrangência e riqueza maior no geral do que o antecessor. Essa função realiza a combinação e adaptação das partes disponibilizadas pelos autores.

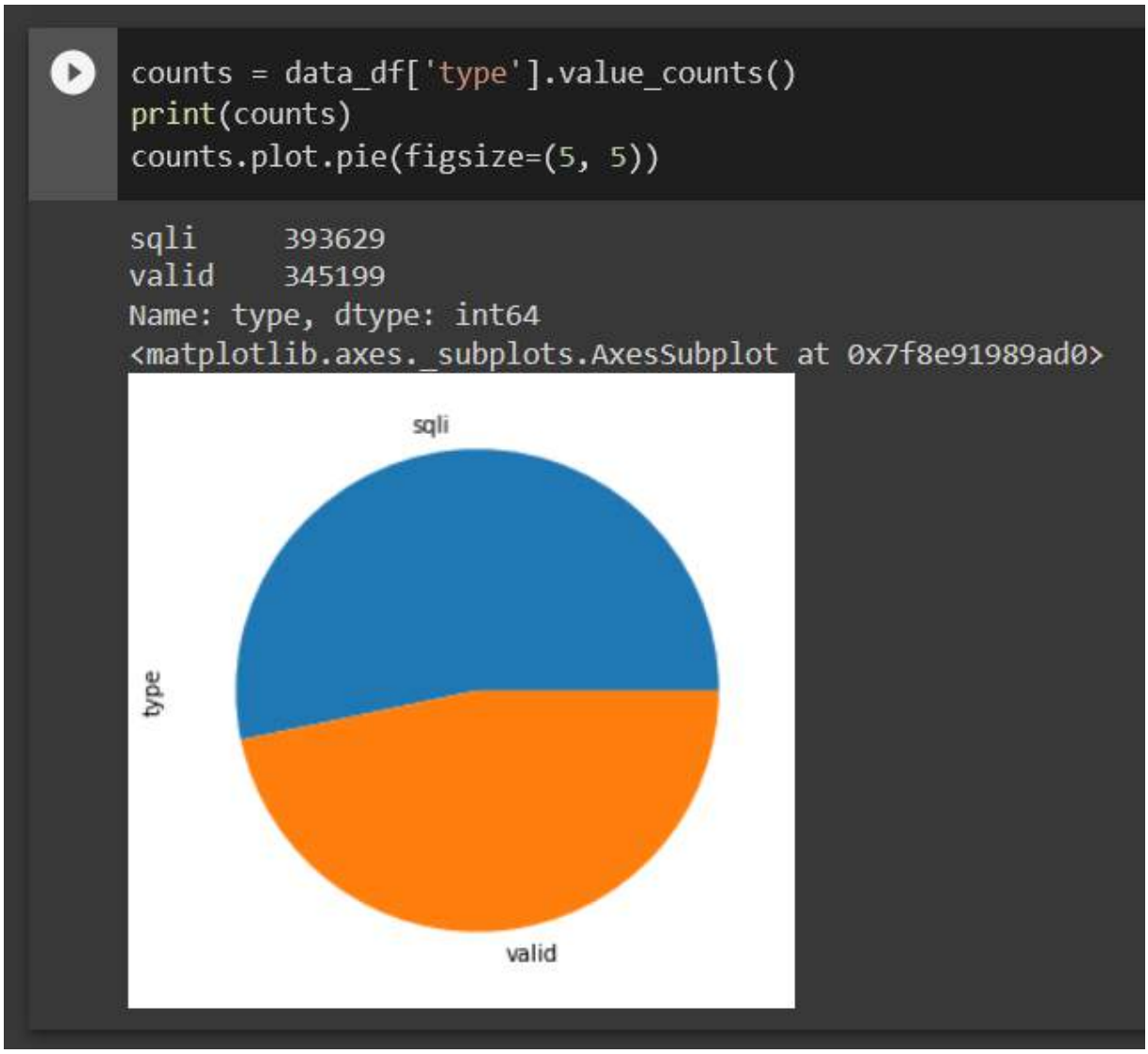
b) `MLBasedWafClassifier` - Localizada na pasta dos modelos customizados (models/svc), esse arquivo *Jupyter Notebook* [5] é responsável por uma série de funções realizadas em cima dos conjuntos de dados baseados no `SQLiV3.json`. É realizada a visualização dos dados contidos nos mesmos via gráficos, e os dados são agrupados de maneira a serem classificados para uso no *wafamole++*. Dois tipos de classificadores são suportado Máquina de vetores de suporte não linear e Gradiente Descendente Estocástico.

Um exemplo de um gráfico gerado pelo notebook com seu código correspondente pode ser encontrado na Figura 8.

[4] GitHub - wafamole_dataset: https://github.com/zangobot/wafamole$_dataset$

[5] Jupyter Notebook ou arquivo .ipynb: Um ambiente de desenvolvimento que permite organizar uma aplicação separando em pedaços de código e trechos de texto

Figura 8 – Exemplo de gráfico gerado pelo Notebook



Para cada classificador é feita uma divisão em dados de treinamento e teste, para que seja averiguada a eficácia do modelo gerado ao fim da execução do código. Também é executada uma rotina *GridSearchCV* para exaustivamente obter parâmetros mais otimizados para uso na classificação. Por fim, é chamada a função `fit()` responsável pela classificação, e o classificador resultante é testado.

Os resultados desses testes sendo colocados em métricas como a matriz de confusão [6] e pontuação, e a biblioteca *joblib* exporta um arquivo `.dump` com o modelo final para uso no *wafamole++*. O final do componente contém testes com consultas SQL geradas pelo *wafamole++* que penetram o WAF avaliado no notebook.

Para maior detalhamento e visualização das métricas no mesmo, o notebook está

[6] Também conhecida como Matriz de Erro, essa matriz permite extrair uma série de métricas acerca da qualidade do modelo gerado, como taxas de falso positivo/negativo e verdadeiro positivo/negativo, além de acurácia e precisão.

disponibilizado no repositório final [7], localizado sob a pasta `models/custom/svc`.

c) `MLClassifierWafamole` - Trata-se de uma versão semelhante ao notebook anterior, porém fazendo uso do conjunto de dados original do wafamole, um banco consideravelmente mais robusto e amplo no geral. Por robustez, entende-se por consultas em maior numerosidade (uma diferença quase 690.000 registros a mais do que o conjunto de dados do Kaggle), e por amplitude, a variedade de comandos SQL nos registros, sendo mais representativo.

   O tamanho deste conjunto de dados exigiu algumas alterações para que os algoritmos rodassem apropriadamente, sendo um obstáculo no desenvolvimento do trabalho no geral, pois o processamento do mesmo exigia mais recursos, tempo e algumas modificações em eficiência para que funcionasse. Destaca-se aqui o uso do *back-end* paralelo `ray` (RAY..., 2022) como considerável ganho de performance. Mesmo assim, algumas execuções foram realizadas por um período maior do que 48 horas e abandonadas sem resultados na etapa de treinamento.

   No geral, este notebook possui um tempo de execução consideravelmente maior em razão do conjunto de dados utilizado, o que é de se esperar e não pode ser evitado. Outra diferença chave são os classificadores usados - foram testados além dos dois anteriores o classificador *AdaBoost*, com resultados promissores. A Máquina de vetores de suporte não linear, no entanto, não foi frutífera, requerendo um tempo de execução que excedia 48 horas. Cabe salientar que a versão mais recente desse notebook foi disponibilizada no *Google Colab* sob o nome `MLClassifierColabWafamole.ipynb`, via um link providenciado no repositório final.

De forma a ilustrar a implementação de uma maneira geral e como esses módulos se comunicam, a Figura 9 é disponibilizada, e para melhor detalhamento o componente `mole` pode ser visto no Código 3 por inteiro.

[7] GitHub - Implementação Final: https://github.com/nidnogg/tcc

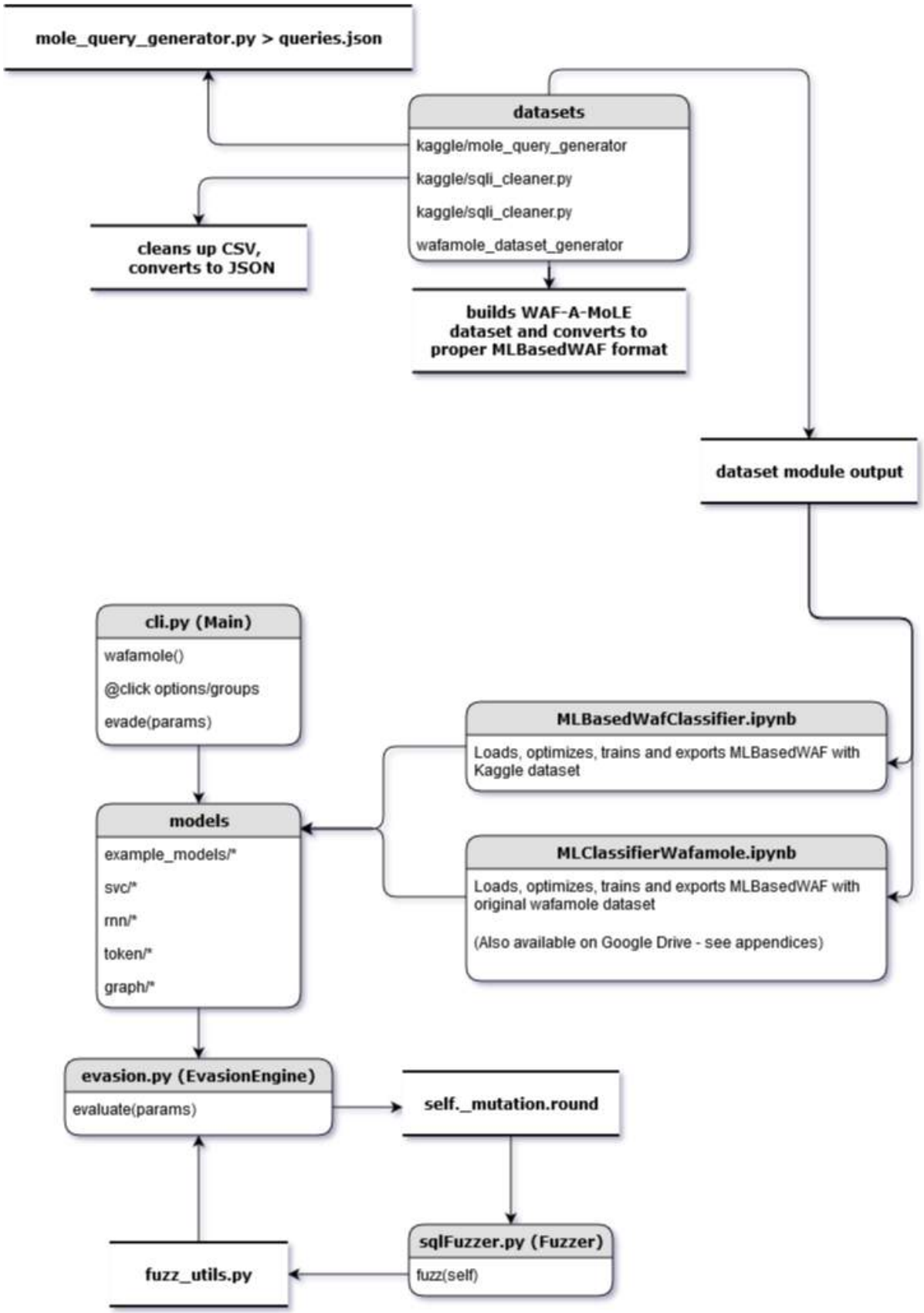
Código 3 – Reforço de conjunto de dados SQLiV3.json do Kaggle

```
import subprocess
import json

f = open('SQLiV3.json')
queries = json.load(f)

i = 0
for query in queries:
    bash_cmd = ["wafamole", "evade", "--model-type", "svc", "../wafamole
       /models/custom/svc/test_svc_classifier_no_mole.dump", query['
       pattern']]
    process = subprocess.Popen(bash_cmd, stdout=subprocess.PIPE)
    output, error = process.communicate()
    i+=1
    if(i == 7000): break
```

Na Figura 9, as principais diferenças para a Figura 7 (que ilustra a arquitetura independente de tecnologias empregadas) se encontram no nome dos módulos, na extensão de arquivos e um detalhamento maior em módulos como `datasets` e `cli.py`. Destaca-se dentre os detalhes os operadores da biblioteca *Python* `click`, que facilitam o uso do programa através da linha de comando (localizado no módulo `cli.py`), o método `evade()` que é chamado na execução principal do programa e dá início ao funcionamento do mesmo, e uma melhor ilustração à direita de `evasion.py` de como o *loop* principal de mutação é feito, com o método `mutation_round()`.

Além desses detalhes, há uma bipartição no módulo de classificador, que se deu pela particularidade dos múltiplos conjuntos de dados que foram usados no *wafamole++*. Acredita-se que em uma implementação nova, tendo acesso aos dados originais no início do desenvolvimento, não seria necessário explorar mais de um banco inicial de injeções e comandos SQL, apenas aprimorá-lo com mais entradas, fazendo uso do mesmo módulo de classificador caso desejado.

Figura 9 – Implementação do wafamole++



4.4 OPERADORES DE MUTAÇÃO

O papel dos Operadores de Mutação tanto no *WAF-A-MoLE* quanto no *wafamole++* é fundamental, atuando como as ferramentas para guiar as mudanças realizadas no *payload* atual a cada iteração do algoritmo. Sendo assim, considerando-se como o primeiro desenvolvimento feito na extensão do programa original, o *wafamole++* conta com três novos operadores de mutação adicionados ao montante original. Todos estes encontram-se agrupados na classe `sqlfuzzer.py`, localizada na pasta **payloadfuzzer** da aplicação.

A finalidade destes novos operadores é incrementar a variedade de exemplos adversariais gerados. O primeiro destes é o `shuffle_integers`, mostrado no Código 4.

Código 4 – Operador de mutação shuffle integers

```
def shuffle_integers(payload):

    # Numeros sao selecionados pela regex como candidatos para muta   o
    candidates = list(re.finditer(r'[0-9]', payload))

    if not candidates:
        return payload

    # Um e aleatoriamente escolhido e sua posicaoo armazenada
    candidate_pos = random.choice(candidates).span()

    # Um inteiro aleatorio entre 0 e 9 e selecionado para substitui-lo
    return payload[:candidate_pos[0]] + str(random.choice(range(10))) +
       payload[candidate_pos[1]:]
```

Sua implementação é a mais simples dentre os operadores novos - consistindo apenas na troca de números presentes no *payload* atual que lidera a fila principal por números inteiros diferentes entre 0 e 9. Selecionam-se números no *payload* (denominados de candidatos), a posição dos mesmos e seus conteúdos são armazenados e então uma substituição aleatória é escolhida e o *payload* modificado é retornado.

Interessantemente, modificando-se uma parte desse funcionamento para trocar a base numérica entre binário, octal, decimal e hexadecimal, obtém-se o próximo operador de mutação, com uma performance mais robusta. Trata-se do `shuffle_bases` (Código 5).

Código 5 – Operador de mutação *shuffle bases*

```
def shuffle_bases(payload):

    # Do payload atual, busca-se pela regex [0-9] todos os caracteres
       que sao numericos
    # Estes serao os candidatos a sofrerem uma mudanca
    candidates = list(re.finditer(r'[0-9]+', payload))

    # Nao havendo n meros, nao ocorre mutacao
    if not candidates:
        return payload

    # A posicao de cada candidato e seu conteudo sao armazenados
    candidate_pos = random.choice(candidates).span()
    candidate = payload[candidate_pos[0]:candidate_pos[1]]

    # Uma lista de possiveis muta  es a serem realizadas - todas sao
       conversoes
    # entre bases binarias, octais e hexadecimais.
    replacements = [
        bin(int(candidate)),
        int(candidate),
        oct(int(candidate)),
        hex(int(candidate)),
    ]

    # Uma mutacaoo e aleatoriamente selecionada
    replacement = random.choice(replacements)

    # Se ela nao altera o candidato, nao ha mutacao
    if (str(candidate) == str(replacement)):
        return payload

    # Payload e modificado na posicao do candidato com a mutacao
       selecionada
    return payload[:candidate_pos[0]] + str(replacement) + payload[
       candidate_pos[1]:]
```

Nessa função a seleção e armazenamento de posições a serem modificadas é virtualmente idêntico - porém a substituição conta com uma lista de modificações em potencial dentre as quais uma é amostrada aleatoriamente na hora de ser realizada a mutação. Também são considerados os casos em que a mutação não ocasiona em variações no *payload* final. Finalmente, há o `spaces_to_symbols` (Código 6).

Código 6 – Operador de mutação spaces to symbols

```
def spaces_to_symbols(payload):

    # Caracteres alfanumericos com excecao de espacos sao excluidos
    excluded_characters = '[^a-zA-Z0-9]'

    # Regex para tal
    r = re.compile(excluded_characters)

    # Lista de simbolos a serem considerados para substituirem espacos
    symbols_to_try = []

    # Sao adicionados sinais de pontuacao
    for symbol in string.punctuation:
        symbols_to_try.append(symbol)


    symbols_to_try = list(filter(r.match, symbols_to_try))

    # Espacos serao trocados por symbols_to_try
    symbols = {" ": symbols_to_try}

    # Funcao filter_candidates filtra payload para encontrar espacos
    symbols_in_payload = filter_candidates(symbols, payload)

    # Sem espacos nao ha mutacao
    if not symbols_in_payload:
            return payload

    # Aleatoriamente escolhe um espaco para ser substituido
    candidate_symbol = random.choice(symbols_in_payload)
    # Checa por possiveis trocas
    replacements = symbols[candidate_symbol]
    # Escolhe uma troca aleatoriamente
    candidate_replacement = random.choice(replacements)

    # Aplica mutacao na posicao, via funcao do wafamole replace_random
       que realiza a troca
    # em payload de candidate_symbol (espaco) por seu substituto,
       candidate_replacement
    return replace_random(payload, candidate_symbol,
       candidate_replacement)
```

O operador `spaces_to_symbols` difere dos anteriores no que tange a posições que serão potencialmente modificadas - pois busca espaços em branco e não números. Com isso, a partir de uma lista gerada no algoritmo de símbolos como sinais de pontuação para

substituição, um espaço é escolhido aleatoriamente e substituído por um desses símbolos.

4.5 MODELOS E WAFS AVALIADOS

Relembrando alguns conceitos vistos anteriormente, para melhor compreender essa seção é importante ter em mente que modelos, classificadores e WAFs estão relacionados da seguinte maneira:

a) Um determinado WAF baseado em Aprendizado de Máquina possui um classificador em seu funcionamento;

b) Um classificador deste WAF pode, após a etapa de treinamento de dados (que consome bastante tempo), ser "exportado"na forma de um modelo como um arquivo `.dump` ou `.h5` através da biblioteca *Python* `joblib`.

As seções a seguir podem então ser interpretadas da seguinte forma:

a) **WAF-Brain** - Um WAF, cujo classificador após o treinamento é usado como modelo no *wafamole++*;

b) **SQLiGOT** - Mesma coisa que a seção anterior;

c) **MLBasedWAF** - Um WAF, cujo classificador foi treinado de diferentes formas e cada forma gerou um modelo diferente;

d) **Máquina de vetores de suporte** - Um modelo baseado no classificador do MLBasedWAF;

e) **Gradiente Descendente Estocástico** - Outro modelo baseado no classificador do MLBasedWAF;

f) **AdaBoost** - Outro modelo baseado no classificador do MLBasedWAF;

g) **Interface Scikit-Learn** - Uma classe que permite com que cada modelo *scikit-learn* funcione no *wafamole++*. Não é um modelo.

4.5.1 WAF-Brain

Autores: Sergio D Fdez, cr0hn, Enrique Garcia. Disponível no GitHub. [8]

O WAF-Brain (FDEZ, 2018) foi um dos primeiros firewalls a serem testados pela equipe do *WAF-A-MoLE*, e subsequentemente é o primeiro modelo de exemplo disponibilizado para testes na documentação do mesmo. Dessa maneira, foi testado com os novos operadores de mutação, sendo de ajuda para diagnosticar os incluídos no wafamole++. A Figura 10 é uma captura de tela do seu repositório, destacando a versão de *Python* em que foi escrito, quantia de problemas abertos e logomarca.

[8] GitHub - waf-brain: https://github.com/BBVA/waf-brain

Figura 10 – Dependências e logomarca do WAF-Brain

Fonte: GitHub (2019)

Seu funcionamento se dá, como anteriormente mencionado, através da estratégia de *Deep Learning* com Redes Neurais, pela qual são analisados cada campo de um determinado pedido HTTP que passam pelo *Firewall* sob a ótica de um classificador treinado desta maneira, determinando se esse campo é malicioso ou não. Caso haja um campo marcado como malicioso, o pedido HTTP é bloqueado.

Apenas o código `inferring.py` responsável pela análise de fato de um *payload* do algoritmo é usado no *WAF-A-MoLE* e *wafamole++*. Nele há uma série de funções que preparam o conteúdo para ser avaliado pelo modelo treinado, como `row_parse()` (para pegar apenas caracteres definidos na constante `VOCABULARY`, parametrizável pelo desenvolvedor), e `reduce_dimension()` (para adaptar as dimensões das frases analisadas em um *array* da biblioteca *numpy*).

Essas funções são utilizadas dentro da rotina `process_payload()`, que realiza a chamada para o modelo através de `model.evaluate()` tendo como parâmetros os dados tratados pelas funções auxiliares anteriormente mencionadas. O resultado dessa função é efetivamente a probabilidade da entrada ser maliciosa ou não.

Na adaptação *WAF-A-MoLE*, esse poder de predição é aproveitado exclusivamente na forma de probabilidade - tendo a probabilidade de ser uma injeção SQL acima de um determinado limite, temos um caso malicioso. Isso requer que o classificador treinado no modelo não apenas mostre a previsão como usualmente é feito, mas também a probabilidade de aquela previsão estar correta de fato.

Por mais que se trate de um algoritmo sofisticado por natureza, surpreendentemente o *WAF-Brain* fora implementado pelos seus autores originais de uma maneira simplória o suficiente para ser um dos WAFs mais vulneráveis às estratégias de *fuzzing* do *WAF-A-MoLE*. Vários operadores de mutação isolados conseguem resultados frutíferos em pouco tempo, como será visto na seção de testes adiante.

4.5.2 SQLiGoT

> *BEFORE LAUNCHING EVALUATION ON SQLiGoT*
>
> *These classifiers are more robust than the others, as the feature extraction phase produces vectors with a more complex structure, and all pretrained classifiers have been strongly regularized. It may take hours for some variants to produce a payload that achieves evasion*

Conforme essa advertência no `README.md` do repositório do *WAF-A-MoLE* feita pelos autores, esse WAF exige um tempo de muitas horas (possivelmente além de um dia, verificado durante o trabalho) para ser penetrado pela aplicação. Além disso, seu código encontra-se inacessível, proibindo a experimentação para o escopo desse trabalho, impossibilitando até mesmo que sejam feitas as etapas de treinamento. Por esse motivo, o SQLiGOT (KAR; PANIGRAHI; SUNDARARAJAN, 2016) foi dispensado como alvo de estudo.

4.5.3 MLBasedWAF

Todos os modelos implementados tiveram como base o firewall ML-Based-WAF, disponível por vladan-stojnic no GitHub [9] (STOJNIC, 2020) que fora o WAF de código aberto mais promissor e mais plausível de adaptar. Sua performance também era factível no contexto de recursos disponíveis para a conclusão deste projeto final, uma vez que não havia complexidade na arquitetura/implementação comparável a firewalls como o SQLiGOT (KAR; PANIGRAHI; SUNDARARAJAN, 2016). Isso permitiu que os modelos resultantes pudessem ser executados tanto na plataforma Google Colab como localmente em alguns casos.

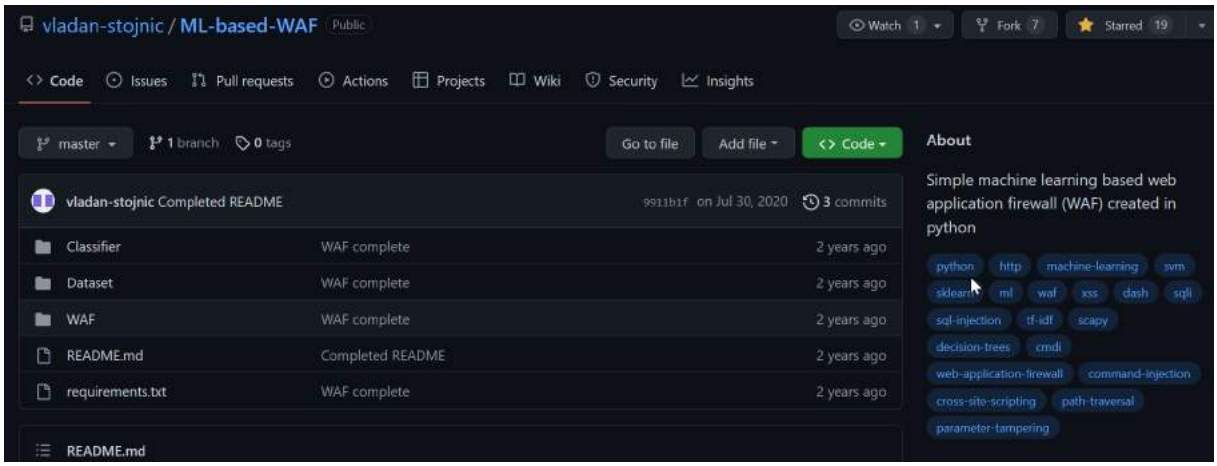
A Figura 11, retirada do repositório original desse WAF, mostra de forma geral a estrutura do MLBasedWAF. A pasta *Classifier* contém alguns classificadores que são treinados com dados do conjunto de dados localizado na pasta *Dataset*. Esses classificadores são replicados na pasta *WAF* que contém as funcionalidades de um WAF propriamente dito, e o núcleo da aplicação.

Seu funcionamento se dá por uso de um *script* (`sniffing.py`, localizado na pasta *WAF*) que vasculha tráfego real e/ou simulado de pacotes HTTP enviados à rede atual, que são então direcionados para um classificador baseado na biblioteca *scikit-learn* que faz a predição se o pacote tem cabeçalhos maliciosos ou não. Esses cabeçalhos são agrupados entre benignos, ataques XSS, cmdi, e injeções SQL.

Embora esse seja um WAF orientado ao tráfego real ou simulado de uma rede de fato, avaliando parâmetros HTTP em tempo real no seu componente `sniffing.py` tanto por injeções SQL como por outras formas de ataque (XSS, cmdi), ainda foi possível restringir seu funcionamento para adaptá-lo ao *wafamole++* de forma que apenas injeções SQL

[9] GitHub - ML-based-WAF: https://github.com/vladan-stojnic/ML-based-WAF

Figura 11 – Repositório ML-Based-WAF



Fonte: GitHub (2020)

fossem avaliadas, pois o *WAF-A-MoLE* comporta apenas esse tipo de vulnerabilidade e o tempo de pesquisa e desenvolvimento para expandir para outros tipos não era viável para este trabalho.

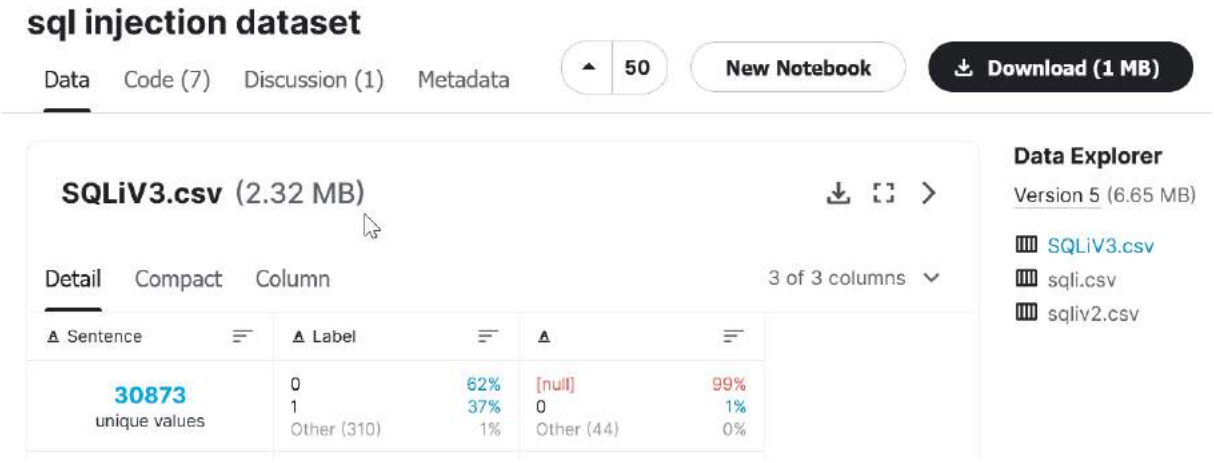
Para tal, foi necessário prever uma estratégia própria de treinamento para o classificador, além de um conjunto de dados especial para o mesmo. É cabível mencionar, no entanto, que isso introduziu uma série de dificuldades uma vez que sua documentação era consideravelmente modesta, e o banco novo a ser usado deveria seguir o padrão de formatação seguido pelo WAF.

Dentre as opções de conjuntos de dados disponíveis com injeções SQL, foi escolhido inicialmente a versão `SQLiV3.json` da coleção do Kaggle [10] (SAQLAIN, 2021), que foi anteriormente mencionada na seção de arquitetura do *wafamole++*. A Figura 12 é uma captura de tela da página Web deste conjunto de dados, destacando alguns detalhes e metadados de relevância como número de valores únicos, categorias de dados, tamanho de arquivo e etc.

A etapa de correção de uma variedade de erros de formatação, polimento de algumas consultas incorretas (e de difícil proveito), e adaptação para o padrão .json utilizado pelo WAF, encontra-se codificada na função auxiliar `sqli_cleaner.py`, no Código 7.

[10] Kaggle - SQL Injection Dataset: https://www.kaggle.com/datasets/syedsaqlainhussain/sql-injection-dataset

Figura 12 – Detalhes conjunto de dados SQL Injection Kaggle.



Fonte: Kaggle (2021) https://www.kaggle.com/datasets/syedsaqlainhussain/sql-injection-dataset

Código 7 – Para tratamento de dados brutos

```
import csv
import json

def cleaner(csvFilePath, jsonOutputFilePath):
    outputData = []
    with open(csvFilePath) as csvFile:
        csvReader = csv.reader(csvFile)
        i = 0
        escapes = ''.join([chr(char) for char in range(1, 32)])
        translator = str.maketrans('', '', escapes)
        for row in csvReader:
            jsonRow = {}
            if(row[1].isnumeric() == False):
                continue
            jsonRow["pattern"] = row[0]
            if(int(row[1]) == 1):
                jsonRow["type"] = "sqli"
            else:
                jsonRow["type"] = "valid"

            outputData.append(jsonRow)
            i+=1


    with open(jsonOutputFilePath, 'w') as jsonFile:
        jsonFile.write(json.dumps(outputData, indent=None))


filepath = './SQLiV3.csv'
jsonFilePath = './SQLiV3.json'
cleaner(filepath, jsonFilePath)
```

4.5.4 Máquina de vetores de suporte

Este foi o primeiro modelo implementado no *wafamole++*, antes do recebimento do conjunto de dados original da equipe dos autores. Foi executado no módulo do conjunto de dados do Kaggle e no módulo do conjunto de dados original do *WAF-A-MoLE*. Dado isso, esse classificador foi implementado com êxito apenas usando os conjuntos de dados `SQLiV3.json`, `SQLiV4.json` e `SQLiV5.json`. Tais conjuntos de dados correspondem, respectivamente, aos seguintes modelos:

a) `test_svc_classifier_no_mole.dump`

   (equivalente ao `svc_trained.dump`);

b) `test_svc_classifier_moled.dump`;

c) `test_svc_classifier_extra_moled.dump`

A nomenclatura *moled* foi escolhida para mensurar o quão reforçado o conjunto de dados usado no treinamento é - sendo tal reforço definido como a quantia de consultas "fuzzeadas"pelo *wafamole++* acrescentadas ao conjunto de dados inicial. E é evidente uma diferença na performance de cada modelo gerado - observou-se uma progressão de aumento (em média) de rodadas mutacionais (consumidas durante a execução) para os dois últimos modelos, especialmente no último.

Em termos de implementação, como fora usado no `SVCClassifierWrapper` criado em torno do **ML-Based-WAF** que usa a biblioteca `scikit-learn`, esse modelo também a utiliza no seu funcionamento. O classificador usado no modelo é o `SVC()`, definido como uma classe e trazido através de um comando de `import - from sklearn.svm import SVC`. Essa classe é instanciada na pipeline `make_pipeline` (`joblib`) com os parâmetros encontrados através de experimentação e otimização via `GridSearch` (uma espécie de força bruta para testar várias combinações de parâmetros e seus resultados subsequentes, explorado também no Capítulo 2).

A divisão entre dados de validação e de teste é de 25% do conjunto de treinamento como dados de teste, sem uso do algoritmo de *K-fold* para a divisão (usando-se apenas o método da biblioteca *scikit-learn* `train_test_split()`).

Antes de fornecer os dados ao classificador `SVC()`, os comandos SQL do conjunto de dados de entrada passam por uma etapa comum a todos os novos modelos, em virtude do funcionamento do **ML-Based-WAF** - uma vetorização através da classe `TfidfVectorizer`, com os parâmetros padrões do WAF. Não foram feitos experimentos acerca dos parâmetros dessa classe intermediária. Isso divide os dados brutos, agrupando-os por palavras organizadas por seu peso em relação a entrada inteira, na forma vetores de rótulos (em inglês: *feature vectors*) que serão lidos pelo classificador. Uma das vantagens de trabalhar com as pipelines da biblioteca `joblib` é poder usar a saída do vetorizador `TfidfVectorizer` como entrada da classe `SVC()` nela com facilidade, como ilustrado no Código 8.

Ao treinar um SVM com kernel RBF (*Radial Basic Function*) que corresponde ao classificador não linear (discutido no Capítulo 2) existem dois parâmetros a serem considerados: **C** (que não é exclusivo do kernel RBF) e **Gamma**.

C faz uma troca entre classificar erroneamente durante os treinamentos e ter uma superfície de decisão [11] mais simples. Um C baixo permite uma superfície de decisão suave, enquanto um C alto caminha para classificar os exemplos de teste corretamente.

Já Gamma determina o quanto apenas um teste irá influenciar. Quanto maior o Gamma, mais próximos os outros valores precisam estar do teste para serem influenciados.

Valores de C e Gamma devidamente calibrados são críticos para a performance do SVM. Dado isso, os parâmetros utilizados para o classificador `SVC()` foram:

a) C (Parâmetro de Regularização - potência da regularização sendo o inverso proporcional a C): **10**

b) kernel (Tipo de kernel usado no classificador): **rbf**;

c) probability (Para fazer uso da métrica de probabilidade): **true**;

d) gamma (Coeficiente para o kernel escolhido): **scale**

Infelizmente um dos pontos de dificuldades foi o treinamento de um classificador deste tipo para o conjunto de dados original do *WAF-A-MoLE*, recebido após os testes iniciais. Em suma, tempos de execução excederam 48 horas em alguns casos, impossibilitando a conclusão da função `fit` de treinamento, mesmo fazendo uso do backend de paralelização `ray`. Isso revelou uma inadequação do SVM para esse caso, sugerindo que outros classificadores sejam usados em seu lugar para conjuntos grandes de dados desse tipo.

Resumidamente, com isso tem-se que além da testagem em cima do reforço via *wafamole++*, estruturalmente buscou-se diferenciar a implementação de Máquina de vetores de suporte da utilizada anteriormente no *WAF-A-MoLE*. Diversos parâmetros foram modificados, e fazendo-se uso de uma busca *paramgrid* (para testar exaustivamente combinações de parâmetros) complementada de experimentação após sua execução, foi possível obter essa configuração vista acima.

Nesse contexto destaca-se um que troca o algoritmo fundamental por trás da Máquina de vetores de suporte por um com abordagem não linear (no caso, de parâmetro `kernel = 'rbf'`) em contraste com o algoritmo linear empregado no modelo baseado em *tokens* (intitulado simplesmente de *Token-based*) padrão do WAF-A-MoLE. Isso permitiu resultados comparáveis aos originais (e, após reforço, com performance superior como indicado no Capítulo 5 adiante) com um classificador fundamentalmente diferente na mesma arquitetura, expandindo o leque de classificadores suportados na aplicação. Sua

[11] Existe uma representação visual bidimensional para classificação de aprendizado denominada superfície de decisão, onde as características que o classificador pode escolher são espaços delineados e as decisões são pontos dentro desses espaços.

escolha foi impulsionada pela experimentação e a rotina *GridSearchCV* de otimização de parâmetros.

Por fim, sua célula de código principal contida no *Python* Notebook pode ser vista no Código 8.

Código 8 – Pipeline classificador SVC Não-Linear

```
# Support Vector Classification
pipe = make_pipeline(
    # Vetorizador Tfidf para trabalhar com numpy arrays compat veis com
        o dataset, com parametros otimizados via paramgrid
    TfidfVectorizer(input = 'content', lowercase = True, analyzer = '
        char', max_features = 1024, ngram_range = (1, 2)),
    # Classe do Classificador Support Vector Machine e seus parametros
        finais otimizados.
    # Nota-se o parametro probability=true, essencial para o modelo
        funcionar no wafamole++ fazendo uso da funcao predict_proba().
    SVC(C = 10, kernel = 'rbf', probability=True, gamma='scale'))

# Variaveis trainX e trainY estao devidamente documentadas nos notebooks
    .
print(pipe.fit(trainX, trainY))
```

4.5.5 Gradiente Descendente Estocástico

O segundo modelo a ser programado pôde ser testado tanto para os conjuntos de dados do Kaggle como do *WAF-A-MoLE* original, sem problemas de performance. No caso posterior, de fato houve uma exigência maior da máquina, requerendo um ambiente com mais memória que o desktop originalmente usado para executar os notebooks. Com isso, foi empregada a plataforma Google Colaboratory, da subdivisão de pesquisa da Google.

Como no modelo anterior e nos subsequentes, sua implementação é realizada a partir de um comando `import` (`from sklearn.linear_model import SGDClassifier`) da biblioteca `scikit-learn` e o classificador é instanciado na função `make_pipeline()`. Uma diferença na sua implementação é que, como fora gerado um modelo para o conjunto de dados *WAF-A-MoLE* também, este possui alguns pequenos ajustes no tratamento dos dados que servem de entrada para o treinamento. Os dados passam também pela etapa intermediária de vetorização através da classe `TfidfVectorizer()`, conforme funcionamento do *ML-Based-WAF*.

A divisão entre dados de validação e teste é a comum a todos os modelos novos: de 25% do conjunto de treinamento como dados de teste, sem uso do algoritmo de *K-fold* para a divisão (usando-se apenas o método da biblioteca *scikit-learn* `train_test_split()`).

Para melhor compreender os parâmetros utilizados, **loss** é uma função para cálculo de perda do gradiente descendente estocástico a cada amostra em uma unidade de tempo. O parâmetro **penalty** tambem chamado de *regularization* é uma penalidade adicionada à função **loss** que reduz os parâmetros do modelo em direção ao vetor zero. Por fim, **max_iter** é o parâmetro que controla o número máximo de iterações durante o treinamento de dados, variando de 1 a infinito. Os valores utilizados no treinamento foram:

a) loss (Função de *loss* - apenas essa retorna as estimativas de probabilidade necessárias): **modified_huber**;

b) penalty (Também conhecido como termo de regularização. Valores como l1 aumentam esparsidade dos dados, não necessário nesse caso): **l2**;

c) max_iter (Número máximo de "épocas"ou passadas sobre os dados de treinamento): **500**

A mudança de ambiente de notebooks localmente armazenados para o notebooks na plataforma Google Colaboratory ocasionou em uma série de vantagens, dentre as quais destacam-se: uma facilidade em executar um ambiente funcional, disponibilização de um link compartilhável, colaboração simultânea e principalmente mais memória RAM para executar as pipelines de treinamento.

Um fator indesejável é a impossibilidade de gerenciar com maior precisão a versão de Python e algumas dependências - ocasionando numa séria poluição de logs em virtude da deprecação de algumas técnicas empregadas em versões antigas dos pacotes `numpy`, `scikit-learn` e `tensorflow`, dificultando o desenvolvimento e diagnóstico de erros.

Acredita-se que a principal falha da plataforma, no entanto, seja a limitação de sua versão gratuita, no entanto, que desliga todas as células [12] do Notebook em uso após um período fixo de 12 horas, a menos que o usuário esteja ativamente codificando na mesma. Isso dificultou o modelo discutido anteriormente, cujo fluxo de execução excedia facilmente esse intervalo para conjuntos de dados suficientemente grandes.

Não obstante, no caso do Gradiente Descendente Estocástico a performance foi satisfatória o suficiente para gerar um par de modelos a serem testados: um para o conjunto de dados `SQLiV5.json`, intitulado de `test_sgd_wafamole.dump` (cujos parâmetros podem ser vistos no Código 9, na etapa antes de seu treinamento) e um para o conjunto de dados do *WAF-A-MoLE*, `sgd_trained.dump`. A performance dos mesmos será comparada mais detalhadamente no capítulo de testes adiante.

[12] Uma célula em um Jupyter Notebook é um pedaço qualquer do programa que pode ser ou um trecho de código em Python, ou um trecho escrito na linguagem de markup Markdown. Jupyter Notebooks são compostos inteiramente de tais células.

Código 9 – Pipeline classificador SGD sem backend de paralelização ray (Para SQLiV5.json)

```
# Stochastic Gradient Descent Classifier
pipe = make_pipeline(
    TfidfVectorizer(input = 'content', lowercase = True, analyzer = '
        char', max_features = 1024, ngram_range = (1, 2)),
    # Nota-se a funcao modified_huber de perda, requerida para fazer uso
        da predict_proba() no wafamole++
    # e de fato usar o modelo no mesmo
    SGDClassifier(loss="modified_huber", penalty="l2", max_iter=500))
```

4.5.6 AdaBoost

O último modelo, e um dos mais interessantes, é o do classificador AdaBoost, implementado apenas para o conjunto de dados do *WAF-A-MoLE* no final da implementação do trabalho. Sua performance é promissora, superando alguns dos modelos padrões do *WAF-A-MoLE* como alguns dos baseados em Token e o WAF-Brain. Para treiná-lo, foi necessário o uso do backend de paralelização `ray`, tomando ainda um tempo considerável de vários minutos. Sua respectivo `import` é do módulo `ensemble` do `scikit-learn`, com o comando `from sklearn.ensemble import AdaBoostClassifier`. Novamente, os dados passam pela etapa intermediária de vetorização através da classe `TfidfVectorizer()`, conforme funcionamento do *ML-Based-WAF*.

A divisão entre dados de validação e teste é a comum a todos os modelos novos: de 25% do conjunto de treinamento como dados de teste, sem uso do algoritmo de *K-fold* para a divisão (usando-se apenas o método da biblioteca *scikit-learn* `train_test_split()`).

Os parâmetros deste classificador, por sua vez, precisam de ajustes pequenos:

a) n_estimators (Número máximo de estimadores que dita quando o *boosting* [13] é finalizado): **100**;

b) random_state (Controla uma *seed* aleatória dada a cada `base_estimator` em iterações de *boosting*. Com um inteiro nesse parâmetro, obtêm-se uma saída reproduzível em múltiplas chamadas a funções): **0**

O resultado desse modelo é intitulado de `test_ada_classifier.dump`. No Código 10 é demonstrado e comentado com mais detalhes como é feita essa adaptação além da etapa de treinamento (note os métodos `pipe.fit()` e `pipe.score()`, responsáveis pelo treinamento e avaliação respectivamente).

[13] Boosting é o algoritmo que transforma um processo de aprendizado fraco em um de aprendizado forte e é melhor detalhado na Subseção 2.4.3, do Capítulo 2. Isso reduz viés e variância em aprendizado supervisionado.

Código 10 – Pipeline classificador AdaBoost com backend de paralelização ray

```
# Ada Boost Classifier
pipe = make_pipeline(
    TfidfVectorizer(input = 'content', lowercase = True, analyzer = '
        char', max_features = 1024, ngram_range = (1, 2)),
    # O classificador AdaBoost foi otimizado com esses parametros, e
        nenhum parametro extra
    # e requerido para uso da funcao predict_proba()
    AdaBoostClassifier(n_estimators=100, random_state=0))

# a seguinte funcao encapsula o treinamento no backend de paralelizacao
    ray,
# consideravelmente reduzindo o tempo de execucao. Tambem utilizado no
    modelo SGD
register_ray()
with joblib.parallel_backend('ray'):
  pipe.fit(trainX, trainY)
  pipe.score(testX, testY)
  joblib.dump(pipe, 'drive/My Drive/@tcc/test_qda_classifier.dump')
```

4.5.7 Interface Scikit-Learn

Por fim, para cada um desses novos modelos funcione, é necessária a criação de uma classe intitulada de *SVCClassifierWrapper* (pois foi inicialmente criada com o SVC/SVM em mente, mas depois foi utilizada nos demais modelos criados) para servir de ponte entre a pipeline gerada pela biblioteca `joblib` (arquivos `.dump`) e o *wafamole++*. Relembrando a arquitetura do *WAF-A-MoLE* original na Figura 6, essa classe seria o componente `WAF Adapter`, porém um pouco mais abrangente servindo para qualquer classificador da biblioteca `sciki-learn`.

A classe encontra-se disponibilizada no Código 11 a seguir. É composta por dois métodos, `classify()` e `extract_features()`. O método `classify()` é responsável por realizar a predição em si, e classificar uma entrada em maliciosa ou não. Essa classificação é expressa na forma de uma probabilidade da sua entrada ser maliciosa, e é desse formato que sai a necessidade de todos os classificadores implementarem o método `predict_proba()` anteriormente discutido no detalhamento dos parâmetros.

Já o método `extract_features()` atua como uma simples função identidade, sendo usado dentro do método `classify()` também para verificação se as entradas usadas na classe são do tipo `String` ou não.

Código 11 – Classe SVCClassifierWrapper para modelos novos

```
from wafamole.models import SklearnModelWrapper
from wafamole.utils.check import type_check
from wafamole.exceptions.models_exceptions import (
    SklearnInternalError,
    ModelNotLoadedError,
)

class SVCClassifierWrapper(SklearnModelWrapper):
    def extract_features(self, value: str):
        type_check(value, str, "value")
        return value

    def classify(self, value):
        """Produce probability of being sql injection.

        Arguments:
            value (str) : input query

        Raises:
        TypeError: value is not string
        ModelNotLoaderError: no model is loaded
        SklearnInternalError: generic exception

        Returns:
           float : probability of being a sql injection
        """
        if(self._sklearn_classifier == None):
            raise ModelNotLoadedError()
        feature_vector = self.extract_features(value)
        try:
            y_pred = self._sklearn_classifier.predict_proba([
               feature_vector])
            return y_pred[0,0]
        except Exception as e:
            raise SklearnInternalError("Internal sklearn error.") from e
```

5 EXPERIMENTOS

O propósito inicial dos testes realizados é compreender amplamente o impacto das adições ao programa. Para tal, uma série de métricas foram empregadas para avaliação na execução do programa, de uma maneira exaustiva, para todos os modelos relevantes (excluindo-se apenas o SQLiGOT, por motivos já mencionados). Através dela foi possível obter uma ideia do impacto que os novos modelos têm, e como o *wafamole++* se adapta com *wrappers* novos.

O parâmetro principal para tal avaliação foram **500** execuções por modelo no *wafamole++*, com 1000 rounds (rodadas de mutação/iterações) disponíveis por execução, e a *threshold* [1] de 0.5. Tanto os modelos antigos como os modelos novos foram examinados. As medidas bases extraídas foram:

a) **Número de rodadas restantes** - A partir dessas obtém-se o número de rodadas ou iterações consumidas;

b) **Menor probabilidade de *payload* encontrada** - É possível medir com essas a precisão de um determinado modelo;

c) **Tempo de execução em segundos** - Tanto para o tempo médio de execução como para diagnosticar a robustez. Um tempo maior indica uma maior eficiência em virtude da reforço do conjunto de dados;

d) **Execuções com iterações máximas alcançadas** - Quanto mais ocorrências dessa medida, mais robusto é o modelo uma vez que não é gerado um exemplo adversarial na execução.

Os modelos antigos incluídos no *WAF-A-MoLE* foram todos testados, com exceção do `SQLiGOT` (por motivos anteriormente citados de performance e tempo de execução), para fins de comparação com os novos modelos introduzidos no *wafamole++*.

5.1 AMBIENTE DE TESTES

Para quesito de reprodutibilidade dos testes, o ambiente utilizado para efetuá-los foi uma máquina Windows 10, com as seguintes condições:

a) Kernel Debian executando *Windows Subsystem for Linux* (WSL), edição WSL 2;

b) Todos arquivos armazenados no sistema de arquivos do WSL, e não no Windows. Sem essa condição, a performance sofre consideravelmente;

c) 8GB de memória RAM, com 6GB alocados para a execução WSL;

d) Processador Intel i7 4770k 3.8ghz;

[1] Um valor do parâmetro *threshold* de 0.5 indica que quando o modelo prever uma probabilidade menor que 50% de um payload ser malicioso, a execução para.

e) GPU Nvidia GeForce 1060, edição 3GB VRAM;

f) Disco SSD Sata, com pelo menos 12GBs de espaço livre em armazenamento

Embora o programa estivesse funcional em máquinas virtuais Debian via *VirtualBox*, escolheu-se WSL 2 pela performance mais próxima de um sistema original Linux.

5.2 RESULTADOS E DISCUSSÃO

Nessa seção é disponibilizada a Tabela 1, com informações relevantes, associadas às métricas definidas acima. A quantidade de rounds restantes se refere ao número de rounds (de um valor inicial de 1000) restantes na execução ao atingir a probabilidade de *threshold* (este parâmetro fornecido pelo usuário é explicado na nota de rodapé no início da seção). Os nomes dos modelos são os mesmos do Capítulo 4 na seção de implementação, com seus respectivos parâmetros:

Tabela 1 – Visão Global de Execuções Realizadas

Modelo	Runtime médio em segundos	Qtd. de Rounds restantes média (arredondado)	Execuções sem êxito
`test_ada`	181.5269	334	198
`test_svc_extra_moled`	8.6750	715	26
`test_svc_moled`	7.6486	752	16
`test_svc_no_mole`	2.4484	993	0
`sgd_trained`	2.3822	988	0
`test_sgd_wafamole`	2.1332	996	0
`token_random_forest`	3.0299	986	1
`token_naive_bayes`	2.7982	969	1
`token_linear_svm`	2.3284	997	0
`token_gauss_svm`	2.2002	999	0
`waf_brain`	3.6010	1000	0

Os fatores que indicam uma performance melhor são relativamente contraintuitivos - um *runtime* médio grande indica dificuldade em gerar um exemplo adversarial (TENSORFLOW..., 2022), que na verdade é um bom indicador para o WAF sendo avaliado. Similarmente, poucos rounds restantes ao final da execução apontam para muitas iterações sendo consumidas e um WAF robusto, e um alto número de execuções sem êxito sugere casos em que o WAF não se mostra suscetível a um ataque naquela execução.

Nesse contexto, um ponto de interesse é que todos os modelos do *WAF-A-MoLE* apresentaram uma performance semelhante, com tempos de penetração rápidos e no geral mostrando-se como WAFs pouco robustos e vulneráveis, sendo superados por uma boa parcela dos modelos novos (em particular os reforçados).

Curiosamente, o modelo `test_sgd` gerado a partir do conjunto de dados original do *WAF-A-MoLE* não apresentou significativas melhorias em comparação com o modelo do conjunto de dados `SQLiV3.json`, `sgd_trained`. Acredita-se que seja uma questão de adequação do classificador utilizado, uma vez que com a respectiva substuição do Gradiente Descendente Estocástico pelo *AdaBoost* e virtualmente a mesma implementação o mesmo se torna o WAF mais robusto do *wafamole++*.

Dado isso, os modelos de maior interesse com base nos dados obtidos certamente são os intitulados de `test_ada`, `svc_extra_moled`, `svc_moled`, `svc_no_mole`. Neles, é possível ver uma melhora nítida em robustez quando comparado aos demais. O primeiro conjunto de histogramas dos tempos de execução, disponibilizados na Figura 13 a seguir detalha a evolução da robustez do *Firewall* alvo em função dos reforços incrementados ao conjunto de dados (indicados por `no_mole` como sem reforço, `moled` para um conjunto pequeno de reforço e `extra_moled` para um conjunto ainda maior de reforço), além de comparar com o modelo `test_ada` para perspectiva.

Atentando-se aos valores no eixo das abscissas, é possível identificar que os tempos de execução ficam consideravelmente maiores conforme os reforços são aplicados ao modelo `test_svc`. Somado a isso, uma diferença radical pode ser avaliada no modelo `test_ada`, com a maioria dos tempos de execução levando uma quantia consideravelmente maior do que todos os modelos avaliados. Isso aponta para uma performance próxima do `SQLiGOT`, um modelo virtualmente inacessível por exigir muito do *WAF-A-MoLE* e de código fechado.

Para melhor ilustrar a amplitude de tempos de execução, a distribuição dos mesmos foi mapeada em um gráfico *Box plot* (mostrado na Figura 14), agrupada por cada modelo. O único caso não contemplado é um *outlier* no modelo `token_random_forest`, que fora omitido para não comprometer a visualização do gráfico. Foi também feita uma separação entre o `test_ada` e os demais modelos, uma vez que seu intervalo é distinto o suficiente para distorcer as proporções. O *Box plot* do `test_ada` encontra-se na Figura 15.

Sumariamente, esses resultados permitem compreender que o classificador Gradiente Descendente Estocástico possivelmente não é o mais indicado para esse tipo de modelagem, com um desempenho semelhante independente do banco usado para treiná-lo. Não obstante, fica estabelecida a capacidade de aprimoramento que os modelos gerados pelo *wafamole++* e *WAF-A-MoLE* oferecem a determinados conjuntos de dados de treinamento. Espera-se que a abordagem deste trabalho sendo transparente permita resultados ainda mais enriquecedores de modelos novos em um breve futuro.

Figura 13 – Distribution plot de tempos de execução (em segundos)

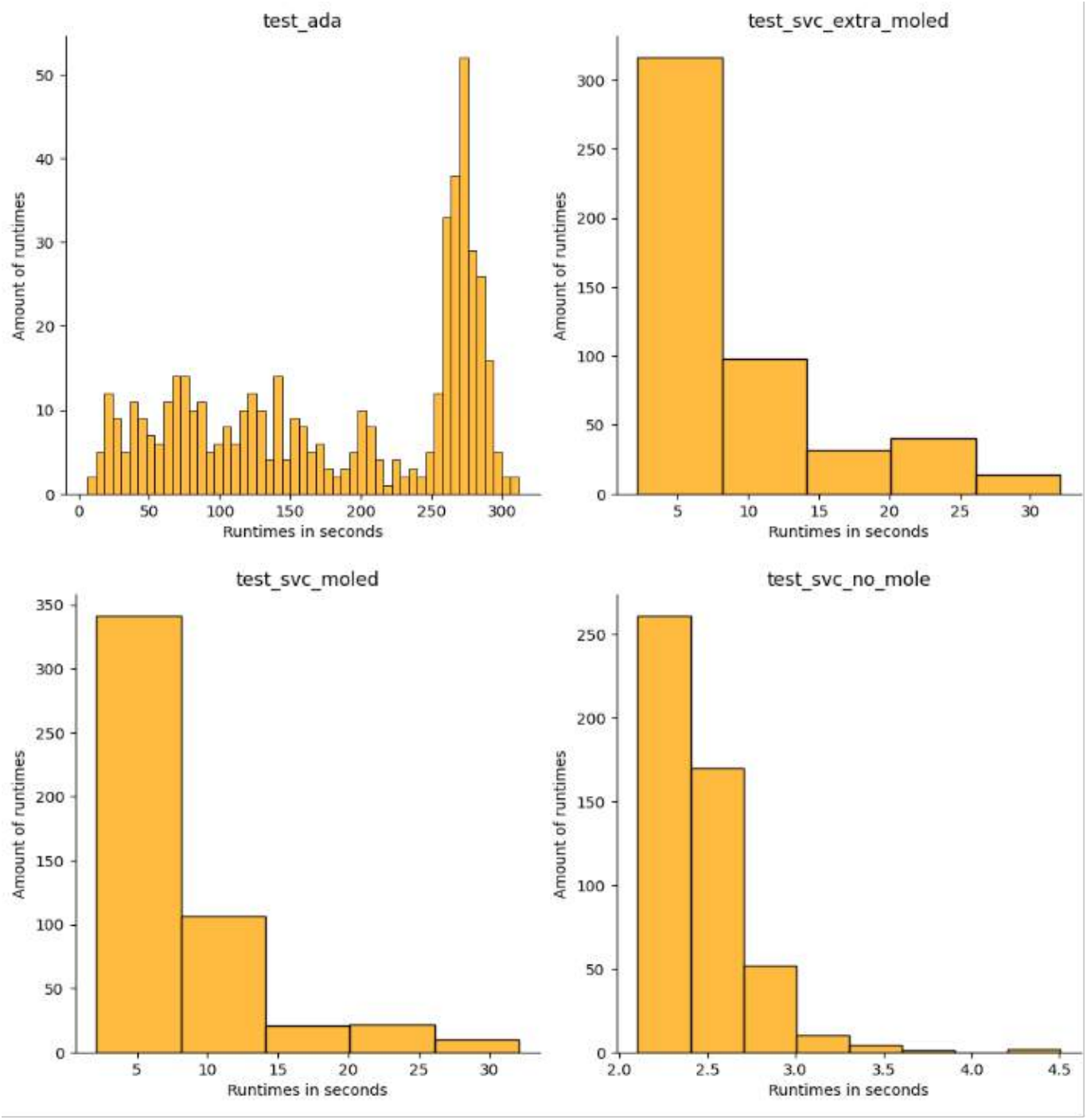


Figura 14 – Box plot de distribuição tempos de execução (em segundos), por modelo

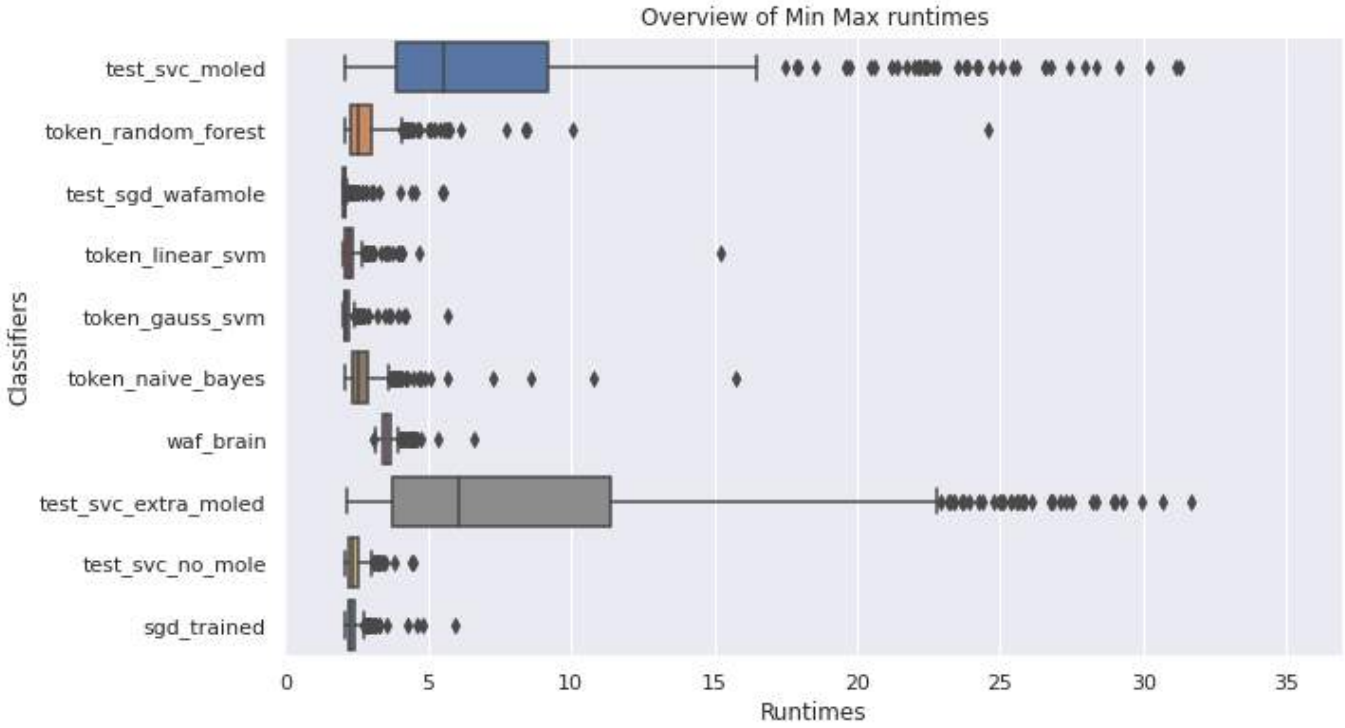


Figura 15 – Box plot de distribuição tempos de execução (em segundos), para modelo AdaBoost

6 CONCLUSÃO

6.1 OBSTÁCULOS ENCONTRADOS

Certamente um obstáculo de destaque que marcou o desenvolvimento da solução *wafamole++* foi a série de limitações impostas aos usuários do plano gratuito na plataforma Google Colaboratory. Uma vez sendo necessário em muitos casos um tempo de execução maior que o limite pré-definido de 12 horas da mesma, fica a reflexão acerca de como seriam os resultados se fosse possível treinar os modelos que exigiam tal poder de processamento com os dados originais do *WAF-A-MoLE*.

Além dessa restrição no plano de uso gratuito da plataforma, o trabalho em torno de dados também trouxe uma série de demandas inesperadas. Foi necessário uma série de funções auxiliares contidas no módulo `sqli_cleaner.py` para tratar os dados originados do Kaggle, que vieram inconsistentes. Após o fornecimento dos dados originais do *WAF-A-MoLE* pelos autores, os resultados anteriormente acumulados com o conjunto de dados do Kaggle se tornaram mais valiosos por darem uma perspectiva nova de comparação na etapa de testes. Também foram relevantes pois tornaram possível treinar os modelos de SVM não linear `svc_mole` e `svc_extra_moled`, uma vez que a execução do treinamento com os dados originais excedeu o tempo limite do Google Colaboratory e recursos locais.

6.2 RESULTADOS

Com esse trabalho foi enfim encontrada uma contribuição para o progresso do campo, culminando no desenvolvimento do *wafamole++*. O mesmo traz não só modelos e operadores de mutação novos em comparação a sua edição original (*WAF-A-MoLE*), mas uma adaptação para um WAF de código aberto (`MLBasedWAF`) baseado em Aprendizado de Máquina que possui uma performance comparável ao `SQLiGOT` (um modelo de ponta, de alta complexidade e robustez). A diferença chave entre ambos, no entanto, é uma implementação mais simples para desenvolvedores e de código aberto ao público.

Nesse contexto, tendo em mente o projeto de testes com milhares de execuções realizadas, pode-se determinar que o reforço do conjunto de dados `SQLiv3.json` com exemplos adversariais do *wafamole++* nos modelos `test_svc_classifier_moled.dump` e `test_svc_classifier_extra_moled.dump` possui um impacto direto na integridade do modelo e do WAF avaliado, progressivamente tornando mais difícil de achar vulnerabilidades neles conforme mais exemplos são adicionados.

Conclui-se nesse âmbito que os modelos baseados no AdaBoost e o SVC reforçado (*extra moled*) se mostram como prova real de que classificadores em *firewalls* desse subcampo podem ser suficientemente reforçados com o *wafamole++*.

6.3 TRABALHOS FUTUROS

Acredita-se a natureza extensível tanto do *WAF-A-MoLE* como do *wafamole++* permite a ambos um potencial de melhoria razoavelmente amplo. Dado que os modelos novos são inteiramente amostrados da biblioteca `scikit-learn`, cabe a reflexão pelos autores de como seriam resultados oriundos de modelos criados em bibliotecas como `Keras` (em uma implementação mais elaborada que a vista no *WAF-Brain*), `PyTorch`, e afins. Não só isso, como há também uma infinidade de operadores de mutação a serem explorados, com os três escolhidos no *wafamole++* sendo apenas uma simples escolha por parte dos autores.

Como também é demonstrado nos iPython Notebooks do repositório final localizados na pasta `models/custom/svc`, a otimização de hiperparâmetros via `GridSearch` poderia ser mais explorada em um ambiente sem restrições de hardware e tempo de uso como o Google Colab, nos modelos com dados originais do *WAF-A-MoLE* que eram mais exigentes.

6.4 CONSIDERAÇÕES FINAIS

O valor acadêmico e profissional de aplicações como as descritas nesse trabalho foi extensivamente discutido, elucidando possíveis direções da área de Segurança da Informação. Foram analisadas diversas propostas e repositórios com ferramentas e soluções de *infosec* no geral, com muitas sendo descartadas para contribuição ou como significativas.

Com o lançamento dessa aplicação, também são bem-vindas contribuições de desenvolvedores interessados, mantendo-se as mesmas diretrizes de expansão do *WAF-A-MoLE* original. Espera-se que especialistas na área possam enriquecer com essa ferramenta seus WAFs, fazendo uso dos exemplos que são gerados por ela. Sua implementação final encontra-se na plataforma GitHub [1].

[1] GitHub - wafamole++: https://github.com/nidnogg/wafamole-plusplus

REFERÊNCIAS

BACH-NUTMAN, M. Understanding the top 10 owasp vulnerabilities. **arXiv preprint arXiv:2012.09960**, 2020.

BEN-HUR, A. et al. Support vector clustering. **Journal of machine learning research**, v. 2, n. Dec, p. 125–137, 2001.

CHAPELLE, O.; HAFFNER, P.; VAPNIK, V. Support vector machines for histogram-based image classification. **IEEE Transactions on Neural Networks**, v. 10, n. 5, p. 1055–1064, 1999.

CONSORTIUM, W. A. S. **Web Application Firewall Evaluation Criteria - Version 1.0**. 2006. (Acessado em: 06/06/2022). Disponível em: http://projects.webappsec.org/f/wasc-wafec-v1.0.pdf.

DANTAS, L. Transformers: Teoria e Viabilização. **FGV EMAp - Trabalhos de Conclusão de Curso**, p. 70, 2021.

FDEZ, S. **WAF-Brain**. 2018. (Acessado em: 06/07/2021). Disponível em: https://github.com/BBVA/waf-brain.

FREUND, Y.; SCHAPIRE, R.; ABE, N. A short introduction to boosting. **Journal-Japanese Society For Artificial Intelligence**, JAPANESE SOC ARTIFICIAL INTELL, v. 14, n. 771-780, p. 1612, 1999. (Acessado em: 08/06/2022).

FUZZING.INFO - History. 2022. (Acessado em: 28/02/2022). Disponível em: https://fuzzinginfo.wordpress.com/history/.

GNU. **gdb Documentation**. 2021. (Acessado em: 02/06/2022). Disponível em: https://sourceware.org/gdb/documentation/.

INVICTI - What is out-of-band SQL injection? 2022. (Acessado em: 28/05/2022). Disponível em: https://www.invicti.com/learn/out-of-band-sql-injection-oob-sqli/.

KAR, D.; PANIGRAHI, S.; SUNDARARAJAN, S. Sqligot: Detecting sql injection attacks using graph of tokens and svm. **Computers & Security**, Elsevier, v. 60, p. 206–225, 2016.

LI, J.; ZHAO, B.; ZHANG, C. Fuzzing: a survey. **Cybersecurity**, SpringerOpen, v. 1, n. 1, p. 1–13, 2018.

LIU, Z. et al. GraphXSS: An efficient XSS payload detection approach based on graph convolutional network. **Computers & Security**, v. 114, p. 102597, mar. 2022. ISSN 01674048. Disponível em: https://linkinghub.elsevier.com/retrieve/pii/S016740482100420X.

MICROSOFT. **windbg Documentation**. 2022. (Acessado em: 02/06/2022). Disponível em: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/getting-started-with-windbg.

MYSQL Union. 2022. (Acessado em: 01/12/2021). Disponível em: https://dev.mysql.com/doc/refman/8.0/en/union.html.

PEREIRA, F.; CROCKER, P.; LEITHARDT, V. R. PADRES: Tool for PrivAcy, Data REgulation and Security. **SoftwareX**, v. 17, p. 100895, jan. 2022. ISSN 23527110. Disponível em: https://linkinghub.elsevier.com/retrieve/pii/S2352711021001515.

PORT Swigger - SQL Injection. 2022. (Acessado em: 17/12/2021). Disponível em: https://portswigger.net/web-security/sql-injection.

RAY Documentation - Distributed Scikit-learn / Joblib. 2022. (Acessado em: 07/03/2022). Disponível em: https://docs.ray.io/en/latest/ray-more-libs/joblib.html.

RAYS hex. **IDA Pro Documentation**. 2022. (Acessado em: 02/06/2022). Disponível em: https://hex-rays.com/documentation/.

RUDER, S. An overview of gradient descent optimization algorithms. **arXiv preprint arXiv:1609.04747**, 2016.

RUDER, S. **An overview of gradient descent optimization algorithms**. 2016. (Acessado em: 11/05/2022). Disponível em: https://ruder.io/optimizing-gradient-descent/.

SAQLAIN, S. **Kaggle - SQL Injection Dataset**. 2021. (Acessado em: 16/06/2021). Disponível em: https://www.kaggle.com/datasets/syedsaqlainhussain/sql-injection-dataset.

SCHAPIRE, R. E. The strength of weak learnability. **Machine learning**, Springer, v. 5, n. 2, p. 197–227, 1990.

SCIKIT-LEARN, Documentation for svm modules. 2022. (Acessado em: 02/05/2022). Disponível em: https://scikit-learn.org/stable/modules/svm.html.

STOJNIC, V. **MLBasedWAF**. 2020. (Acessado em: 13/12/2021). Disponível em: https://github.com/vladan-stojnic/ML-based-WAF.

STUTTARD, D.; PINTO, M. **The Web Application Hacker's Handbook**. [S.l.: s.n.], 2007.

SURIAN, R. U.; RAHMAN, N. A. A.; NATHAN, Y. Nscanner: Vulnerabilities Detection Tool for Web Application. **J. Phys.: Conf. Ser.**, v. 1712, n. 1, p. 012018, dez. 2020. ISSN 1742-6588, 1742-6596. Disponível em: https://iopscience.iop.org/article/10.1088/1742-6596/1712/1/012018.

TENSORFLOW Documentation - Note on Adversarial Examples. 2022. (Acessado em: 30/12/2021). Disponível em: https://www.tensorflow.org/tutorials/generative/adversarial_fgsm.

VALENZA, A. et al. WAF-A-MoLE: An adversarial tool for assessing ML-based WAFs. **SoftwareX**, v. 11, p. 100367, jan. 2020. ISSN 23527110. Disponível em: https://linkinghub.elsevier.com/retrieve/pii/S2352711019302997.

WALLARM. **GoTestWAF - Wallarm**. 2022. (Acessado em: 06/06/2022). Disponível em: https://github.com/wallarm/gotestwaf.

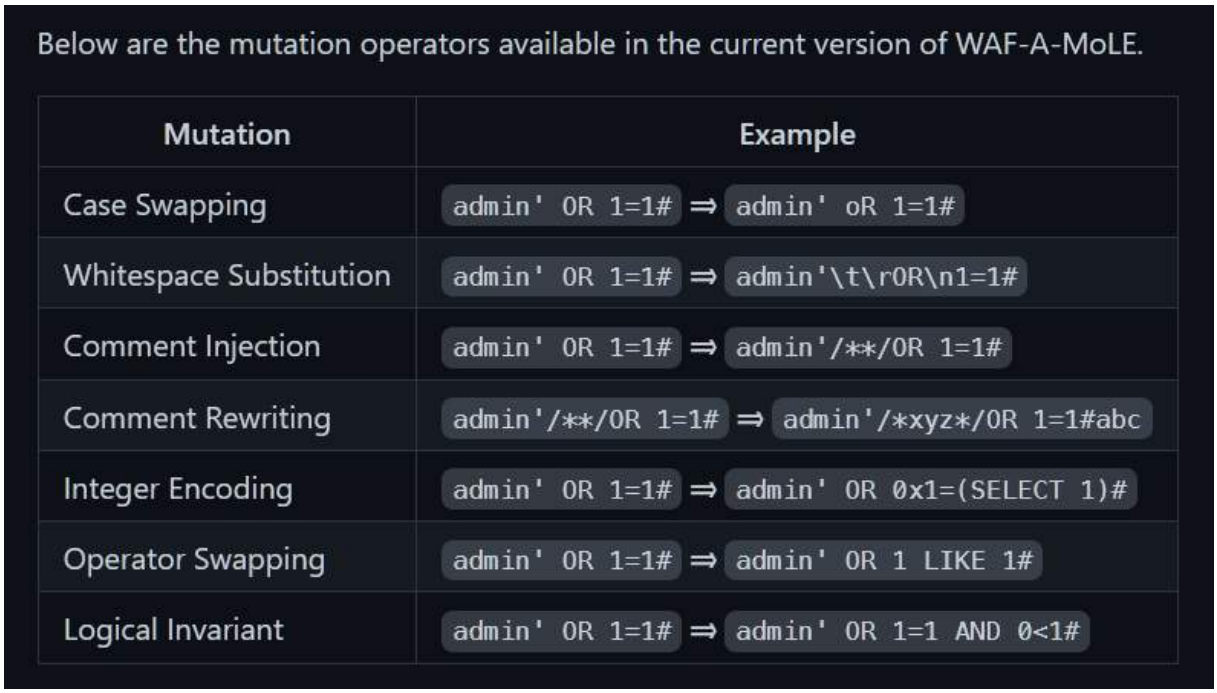
WU, P.; ZHAO, H. Some analysis and research of the adaboost algorithm. In: SPRINGER. **International Conference on Intelligent Computing and Information Science**. [S.l.], 2011. p. 1–5.

ZELLER, A. et al. **The Fuzzing Book**. 2022. (Acessado em: 02/01/2022). Disponível em: https://www.fuzzingbook.org/.

APÊNDICES

APÊNDICE A – LISTA DE OPERADORES DE MUTAÇÃO

Figura 16 – Lista de operadores de mutação WAF-A-MoLE



Below are the mutation operators available in the current version of WAF-A-MoLE.

Mutation	Example
Case Swapping	`admin' OR 1=1#` ⇒ `admin' oR 1=1#`
Whitespace Substitution	`admin' OR 1=1#` ⇒ `admin'\t\rOR\n1=1#`
Comment Injection	`admin' OR 1=1#` ⇒ `admin'/**/OR 1=1#`
Comment Rewriting	`admin'/**/OR 1=1#` ⇒ `admin'/*xyz*/OR 1=1#abc`
Integer Encoding	`admin' OR 1=1#` ⇒ `admin' OR 0x1=(SELECT 1)#`
Operator Swapping	`admin' OR 1=1#` ⇒ `admin' OR 1 LIKE 1#`
Logical Invariant	`admin' OR 1=1#` ⇒ `admin' OR 1=1 AND 0<1#`

Fonte: WAF-A-MoLE GitHub (2022)