iOSDBuilder: Uma Ferramenta de Construção de Datasets para Detecção de Malwares iOS

João Guilherme Freitas[1], **Vanderson Rocha**[1], **Eduardo Feitosa**[1]

[1]Universidade Federal do Amazonas (UFAM)
{joaoguicf, vanderson, efeitosa}@icomp.ufam.edu.br

Abstract. *The lack of datasets to train and test solutions for identifying and classifying malicious applications in the iOS operating system is a reality. While on the Android platform, there are dozens, there is no public dataset to facilitate the training and testing of detection solutions. To solve this problem, iOSDBuilder was developed to build datasets capable of analyzing and detecting iOS malware. iOSDBuilder is separated into four independent modules with different features and tools for building up-to-date datasets. As a result, it was possible to generate a dataset with 176 applications, of which 9 were classified as possibly malicious by the VirusTotal scanners.*

Resumo. *A falta de conjuntos de dados para treinar e testar soluções para identificação e classificação de aplicativos maliciosos no sistema operacional iOS é uma realidade. Enquanto na plataforma Android existem dezenas, não existe um dataset público para facilitar o treino e teste de soluções de detecção. Visando resolver este problema, o iOSDBuilder foi desenvolvido para construir datasets capazes de serem empregados na análise e detecção de malwares iOS. O iOSDBuilder é separado em quatro módulos independentes com características e ferramentas diferentes para construção de datasets atualizados. Como resultado, foi possível gerar um dataset com 176 aplicativos, dos quais 9 foram classificados como possivelmente maliciosos pelos scanners do VirusTotal.*

1. Introdução

A identificação, classificação e mitigação de aplicativos com *malwares* vem crescendo cada vez mais, especialmente com o emprego de modelos de inteligência artificial capazes de classificar e rotular tais aplicativos. Observando o cenário atual de soluções e ferramentas para detecção de *malwares* em dispositivos móveis, percebemos que grande parte é voltada para plataforma Android[Kouliaridis et al. 2020]. Existem poucas soluções para plataforma *iOS*, principalmente devido a pequena quantidade de amostras disponíveis de aplicativos maliciosos [Felt et al. 2011]. Embora a *Apple* tenha grande preocupação com questões de seguranças e métodos de análise durante a submissão de novos aplicativos em sua loja, é estranho notar essa falta de insumos para pesquisar nessa área (quase segurança por obscuridade).

Mas o que falta então? O problema não está na falta de modelos de aprendizado de máquina. Trabalhos como [Cimitile et al. 2017, Husainiamer et al. 2021, Garg and Baliyan 2021, Yixiang and Kang 2017] mostram o emprego de diferentes métodos e classificadores para identificar *malwares* em aplicativos *iOS*. O problema

está na falta de conjuntos de dados (*datasets*) para estas análises, isto é, há uma escassez de insumos relacionados à aplicativos *iOS*. Enquanto o sistema Android existem diversos repositórios (AndroZoo[1], Google Play, 1mobile[2], entre outros) com tal finalidade, no iOS o que existem são sites não confiáveis que distribuem aplicativos, muitas vezes pirateados, para serem baixados por dispositivos desbloqueados, como IpaSpot[3], IpaFile[4], AppAdict[5], IpaLibrary[6], entre outros. Também encontramos sites que disponibilizam aplicativos antigos. O trabalho de [Ciaramella et al. 2022] retrata que, para treinar o método proposto por eles, foi necessário fazer o download manual em três fontes diferentes de repositórios de aplicativos *iOS*. Por sinal, uma delas (Contagio Dump[7]) contém uma lista de aplicativos muito desatualizada.

Uma vez que solucionar essa escassez é necessário e importante na detecção de *malwares* em aplicativos *iOS*, a saída é a geração de um *dataset* confiável, com informações atualizadas, construído de forma automatizada (ou semi-automatizada), incluindo informações sobre o processo de rotulação do aplicativo como malicioso ou não.

Tendo como base o artigo de [Vilanova et al. 2022], que propôs e implementou uma solução para construção de *datasets* de aplicativos benignos e maliciosos para o sistema Android, este artigo apresenta o iOSDBuilder[8], uma solução modular, composta por quatro blocos funcionais autônomos que, quando combinados, viabilizam a criação automatizada e estruturada de *datasets* atualizados de aplicativos *iOS*. Os *datasets* criados podem ser utilizados para treinar modelos de aprendizado de máquina aplicados na detecção de *malwares* direcionados à plataforma *iOS*.

O restante deste trabalho está estruturado da seguinte forma: nas Seções 2 e 3, são apresentados a arquitetura e a implementação do iOSDBuilder. Na Seção 4, são descritas as metodologias utilizadas e a análise dos resultados da avaliação do programa. Por fim, na Seção 5, são apresentadas informações básicas sobre a demonstração e as considerações finais.

2. Arquitetura

Arquiteturalmente, o iOSDBuilder é composto por quatro módulos bem definidos: (i) **download** de IPA (*iOS App Store package)*, (ii) **rotulação** de amostras, (iii) **extração** de características e (iv) **geração** do *dataset*. A arquitetura da ferramenta iOSDBuilder é apresentada na Figura 1.

O módulo de **download** executa duas ações relacionadas a coleta de aplicativos iOS. A primeira é usar uma lista de sites, recebida como entrada, para coletar os possíveis IPAs existentes através de um processo de *scraping* no site e consequentemente realizando o download deles. A segunda, executada em paralelo, é realizar

[1] `https://androzoo.uni.lu`
[2] `http://www.1mobile.com/`
[3] `https://ipaspot.app`
[4] `https://ipafile.com`
[5] `https://appaddict.org`
[6] `https://app.ipalibrary.net`
[7] `http://contagiodump.blogspot.com/`
[8] `https://github.com/guicfreitas/ios-apps-dataset-builder`

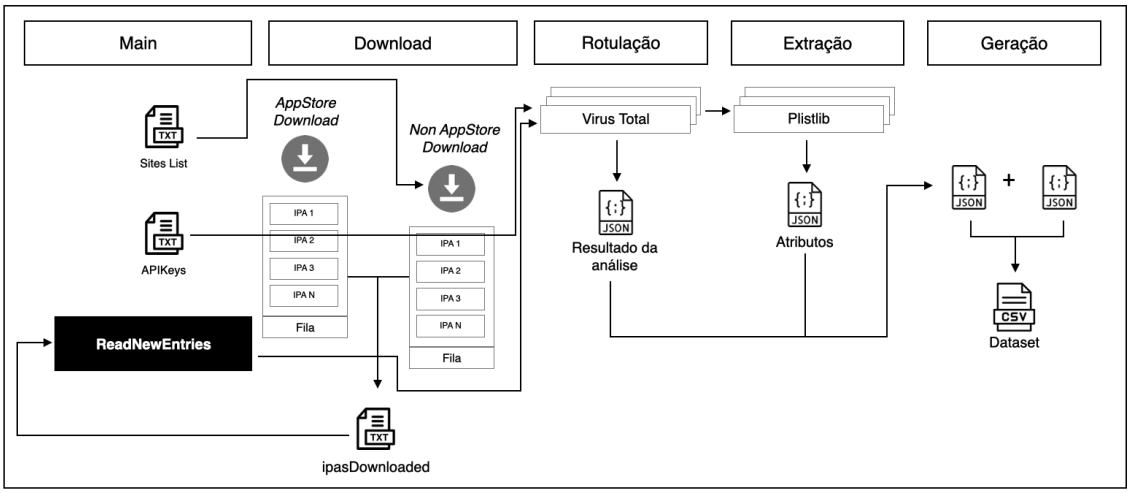

Figura 1. Visão geral das ferramenta iOSDBuilder

o download de aplicativos diretamente da AppStore, onde é permitido coletar os 50 aplicativos mais populares (baixados) de todos os países do mundo. Todos os aplicativos coletados com sucesso, através das duas ações, são registrados em um arquivo de controle para serem usados em outros módulos.

O segundo módulo é o de **rotulação**, que realiza chamadas à serviços públicos de classificação (rotulação) de aplicativos, informando o aplicativo a ser analisado e recebendo como resposta a rotulação de malicioso ou não. Atualmente, utilizamos o VirusTotal[9] por ter sido o mesmo empregado no trabalho de [Vilanova et al. 2022] e por ser bastante utilizado em soluções de detecção, conforme descrito em [Sharma and Rattan 2021]. Vale destacar que qualquer outro serviço, como, por exemplo, polyswarm[10], metadefender[11], Cisco Talos Intelligence [12], entre outros, poderia ser utilizado. Por sinal, na iOSDBuilder não existe qualquer impedimento de empregar dois ou mais serviços de classificação de aplicativos. Tal uso acrescenta redundância ao processo, ao evitar possíveis indisponibilidades de algum serviço, bem como aumentaria a qualidade das classificações ao permitir a comparação entre as rotulações.

De posse do resultado da análise do aplicativo, o terceiro módulo (**extração**) em ação. Este módulo inicia da mesma forma que o módulo anterior, utilizando o nome do aplicativo, extraído do arquivo de controle. Este módulo realiza a abertura do arquivo IPA em formato ZIP, a fim de ser possível a navegação dentro das pastas e conteúdos do aplicativo. Ao realizar a conversão, é necessário buscar dentro dos diretórios do aplicativo o arquivo *InfoPlist*, que contém informações relevantes para análise como: permissões que o usuário deve conceder para usar o aplicativo, chaves de APIs, configurações de funcionamento da própria aplicação, entre outras informações. Para realizar a extração dessas informações, foi necessário usar a biblioteca do Python chamada *plistlib*, que faz a interpretação desse arquivo para um

[9]`https://www.virustotal.com`
[10]`https://polyswarm.network/scan`
[11]`https://metadefender.opswat.com/`
[12]`https://talosintelligence.com/talos_file_reputation`

formato JSON. Este módulo necessita de maior capacidade de processamento que os outros devido ser necessário navegar por quase todo o diretório da aplicação para a busca do arquivo, e também para interpretar todo o arquivo em formato JSON.

O último módulo é de **geração**. Seu funcionamento é bem simples. Este módulo entra em ação recebendo o resultado dos dois módulos anteriores e o nome da aplicação, adicionando uma nova linha ao *dataset* que foi criado ao iniciar a execução do iOSDBuilder. Este módulo não usa tanto poder de processamento, pois apenas faz tarefas de leitura e registro em um único arquivo na raiz do projeto. Após a conclusão deste módulo, todo o ciclo, a partir do módulo de rotulação, se inicia novamente para o próximo aplicativo, até que todos os aplicativos da lista de controle sejam analisados.

É importante salientar que toda a aplicação é controlada por um módulo principal, que faz a paralelização do download de aplicativos oficiais e não oficiais. Ele fica sempre procurando por novas entradas no arquivo de controle de aplicativos baixados, a fim de delegar para uma *pool* de processos a rotulação, extração e geração dos mesmos. Assim, ele fica procurando por novas aplicações, para obter seus resultados, até que os processos de download de aplicação tenham terminado. Após isso o *script* se encerra.

3. Implementação

Esta seção descreve as tecnologias e a implementação de cada um dos módulos.

3.1. Tecnologias

Todas as implementações utilizaram a linguagem Python, versão 3.9.6, e as bibliotecas *BeautifulSoup*[13], versão 4.12.2; *plistlib*, versão 3.12; *zipfile*, versão 3.15.0; *tempfile*, versão 3.15.0; e *multiprocessing*, versão 2.6.2. Além disso, foi utilizada a ferramenta IPATool[14], versão 2.0.3.

3.2. Módulos

O módulo de **download** foi implementado instanciando dois processos, que são executados paralelamente: *downloadAppStore* e *downloadNonAppStoreIpa*.

O processo *downloadAppStore* obtém uma lista de 167 países onde existem lojas da *Apple* e, para cada país, realiza uma chamada do tipo GET no serviço de publicidade da AppStore[15]. Essa chamada retorna uma lista dos 50 aplicativos mais baixados em cada país. Para se obter cada um dos aplicativos retornados, utiliza-se a ferramenta IPATool, que realiza o download direto do aplicativo da *App Store* sem a necessidade de um dispositivo físico. O uso da IPATool se deve ao fato de que, pelos termos de uso da *App Store*, não é possível fazer o download de aplicações sem uma conta fixa, nem disponibilizar APIs ou downloads diretos do site da Apple.

A Figura 2 ilustra um trecho de código onde a IPATool faz a busca pelos 50 aplicativos de cada país, retornando um JSON com diversas informações e incluindo

[13] `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`

[14] `https://github.com/majd/ipatool`

[15] `https://rss.applemarketingtools.com/api/v2/{country}/apps/top-free/50/apps.json`

o *Principal Bundle Identifier*, um identificador da aplicação dentro da *App Store*. Essa identificação é salva em um arquivo de controle para futuros downloads. Após salvar o ID da aplicação, é feita a "compra" da licença da aplicação, necessária para o download mesmo que o aplicativo seja gratuito. Com as licenças adquiridas é feito o download da aplicação. Quando o download é concluído, o arquivo .ipa é salvo no diretório */apps*.

```
try:
        result = subprocess.check_output(["ipatool", "purchase", "-b", bundleIdentifier])
    except subprocess.CalledProcessError as e:
        print("Liscença já obtida ou falha na liscença")

    try:
        result = subprocess.check_output(["ipatool", "download", "-b", bundleIdentifier, "-o",
                                          "./apps", "--format", "json", "--non-interactive"])
        saveBundleIdentifier(bundleIdentifier)
        terminalOutput = json.loads(result)

        filePath = terminalOutput["output"]
        appname = os.path.basename(filePath)
        saveDownloadedList(appname)

    except:
        print("Falha no download da aplicação")
```

Figura 2. Uso do IPATool no iOSDBuilder

O outro processo do módulo de download é o *downloadNonAppStoreIpa*, que realiza a coleta de aplicativos a partir de sites que disponibilizam aplicativos pagos gratuitamente, inclusive jogos com modificações para obter vantagens sobre os outros. A execução da *downloadNonAppStoreIpa* é ilustrada na Figura 3, que obtém a lista de sites para download de aplicativos do arquivo *sitesToScrap.txt*.

```
def downloadNonAppStoreIpa():

    sitesToScrap = open("sitesToScrap.txt", "r")
    urls = []

    for line in sitesToScrap:
        line = line.strip()
        urls.append(line)

    sitesToScrap.close

    for url in urls:
        print("Fazendo scraping na url: " + url)
        print("\n")

        links = list(filter(lambda x: x is not None, getLinks(url)))

        if len(links) > 0:
            for homePageLink in links:
                scrapeIpaFile(homePageLink, url)

                secondaryLinks = getLinks(homePageLink)
                if len(secondaryLinks) > 0:
                    for link in secondaryLinks:
                        if link:
                            scrapeIpaFile(link, homePageLink)
```

Figura 3. Implementação da função downloadNonAppStore

Para cada entrada recuperada do arquivo, a biblioteca *BeautifulSoup* é utilizda para fazer o *Scrapping* da URL, visando encontrar arquivos com a extensão .ipa. Utilizamos a requisição para baixar toda a página e procuramos por todos os links que levam a outras páginas. Em seguida, verificamos cada elemento "a" do HTML e se o atributo "href" desse elemento possui um arquivo .ipa. Quando encontramos o arquivo para download, o processo é iniciado. Esse algoritmo percorre cada link da página principal até a segunda página de cada link encontrado na primeira página, para evitar a complexidade do crescimento do algoritmo. No módulo de download, cada função trata as possíveis exceções. Caso ocorra algum erro durante o processo, uma mensagem é impressa no terminal e o próximo download é iniciado.

É importante mencionar que, no código principal da ferramenta, existe a função *readNewEntries*, executada assim que os processos de download são colocados em execução. Essa função (Figura 4) realiza a leitura do arquivo de controle, que contém o nome de todos os aplicativos .ipa que estão sendo coletados pelos dois processos de download executados. Toda vez que ela encontra novas linhas adicionadas (novos aplicativos baixados), cria uma *pool* de processos com essas novas linhas e chama o módulo de rotulação para realizar sua tarefa.

```python
def readNewEntries(file_path, downloadProcess1: multiprocessing.Process, downloadProcess2: multiprocessing.Process):
    last_position = 0

    while downloadProcess1.is_alive() or downloadProcess2.is_alive():
        apiKeyIndex = 0
        with open(file_path, 'r') as file:
            file.seek(last_position)
            lines = file.readlines()
            last_position = file.tell()

            pool = multiprocessing.Pool()
            for line in lines:
                pool.apply_async(processFile, args=(line, apiKeyIndex))
                if apiKeyIndex != len(rotulationModule.apiKeys) - 1:
                    apiKeyIndex += 1
                else:
                    apiKeyIndex = 0
            pool.close()
            pool.join()
```

Figura 4. Implementação função readNewEntries iOSDBuilder

Para cada processo adicionado na *pool*, a função *processFile* parte ainda do módulo principal, recebe como parâmetros o nome do arquivo a ser processado e um número inteiro relacionado ao índice de um vetor das chaves de API, faz um pré-processamento no nome do arquivo, removendo o caractere de quebra de linha, e, em seguida, chamando da função *prepareForRotulation*. Onde é feito uma espera de 15 segundos para evitar extrapolar o limite de quatro chamadas por minuto no serviço do VirusTotal. É essa função que de fato realiza a chamada dos módulos de rotulação, extração e geração, respectivamente (Figura 5).

No módulo de **rotulação** inicia suas atividades através da função *scanIpaFile*, que recebe o nome do arquivo e a primeira ou única $APIKey$ que foi inserida via parâmetro no início de da execução, a fim de solicitar ao VirusTotal a verificação da presença de *malwares* no referido arquivo. A função *scanIpaFile*, primeiramente, verifica o tamanho do arquivo, pois o VirusTotal só aceita arquivos de até 600 megabytes. Se o arquivo tiver mais de 32 megabytes, é necessário fazer uma chamada adicional para obter um link único de *upload* para o VirusTotal. Antes de fazer

essa primeira verificação, aguardamos 15 segundos para evitar muitas chamadas simultâneas, pois, de acordo com a documentação da API, só podemos fazer 4 requisições por minuto. Após essa espera, usamos a função *uploadFile*, onde é feita uma requisição do tipo POST contendo a $APIKey$ no cabeçalho e o próprio arquivo a ser analisado no corpo da requisição.

```
def prepareForRotulation(fileName, apiKeyIndex):

    rotulationResult = rotulationModule.scanIpaFile(fileName, apiKeyIndex)
    extractionResult = extractionModule.extractInfoPlist(fileName)
    generationModule.generateCSV(fileName, extractionResult, rotulationResult)
    if os.path.exists("./apps/" + fileName):
        os.remove("./apps/" + fileName)

def processFile(line, apiKeyIndex):
    fileName = line.replace("\n", "")
    prepareForRotulation(fileName, apiKeyIndex)
    time.sleep(15)
```

Figura 5. Implementação função processFile e prepareForRotulation iOSDBuilder

A análise do VirusTotal ocorre em mais de 70 antivírus e mais de 10 *sandboxes* de análise dinâmica. Os aplicativos a serem analisados são obtidos do arquivo de controle gerado pelo módulo de **download**. Usamos o *endpoint* de *upload* de arquivos devido ao fato de que não temos uma padronização nos nomes e versões de aplicativos iOS, podendo ficar livre para o desenvolvedor ou para o site que distribui o IPA nomear a aplicação, então não conseguimos usar o *endpoint* do VirusTotal de análise por *hash* que é o mais comum, justamente por não termos essa padronização, o que pode acarretar na geração de um *dataset* menos confiável. Este módulo pode vir a consumir um pouco mais de tempo caso o resultado da análise não seja retornado em 3 minutos.

A resposta dessa requisição é um ID de análise (no formato (`Y2NkZTRkYZE4NZkzYZRiNZYZNj95OWRiODgyYzhkMiM6MTY4ODMyODIZMW=-`) que será usado na próxima etapa para coletar o resultado da análise feita pelo VirusTotal. Na sequência, a função *getAnalysis* (Figura 6)coleta o resultado da análise.

Nessa função, um *loop* mantém a execução funcionado até que uma interrupção seja solicitada e retorne o resultado da análise. Primeiramente, adicionamos um intervalo de espera de 3 minutos para evitar que sejam feitas muitas requisições para verificar se a análise está pronta. Verificamos se ela está pronta quando o campo "status", dentro da resposta da API, tiver o valor "completed". Vale ressaltar que, em todas as requisições do módulo de rotulação, se o código de resposta for 429, a cota de requisições da $APIKey$ está esgotada. Assim, buscamos uma nova $APIKey$, incrementando o índice do vetor de $APIKeys$.

Quando todo o fluxo de chamadas do módulo de rotulação é concluído e temos o resultado das análises, o módulo de **extração** entra em ação, obtendo características importantes do código do aplicativo, como permissões, intenções, chamadas de API, atividades, serviços, provedores, receptores e *opcodes*. No entanto, realizar a extração de todas essas características é um processo complexo e que

dependente de fatores externos. Para que a extração funcione, seria necessário, inicialmente, ter um iPhone com *Jailbreak* (método de desbloqueio do dispositivo) para que fosse possível utilizar ferramentas (bibliotecas) como *Frida*[16], *Cycript*[17], *Radare2*[18], *iRET*[19], *Clutch*[20] e *GDB*[21]. Tais ferramentas nos permitiriam analisar o código dos aplicativos em tempo de execução e, assim, realizar a extração das informações mencionadas anteriormente.

```python
def getAnalysis(analysisId, apiKeyIndex):
    currentApiKeyIndex = apiKeyIndex
    url = "https://www.virustotal.com/api/v3/analyses/" + analysisId

    while True:
        try:
            time.sleep(90)
            headers = {"x-apikey": apiKeys[currentApiKeyIndex]}
            response = requests.get(url, headers=headers)
            if response.json()["data"]["attributes"]["status"] == "completed":
                return json.dumps(response.json()["data"]["attributes"]["results"], indent=4)
                break
        except:
            if response.status_code == 429:

                if currentApiKeyIndex < len(apiKeys) - 1:
                    currentApiKeyIndex += 1
                    return getAnalysis(analysisId, currentApiKeyIndex)
                else:
                    return "Get Analysis failed Quota Exceeded"
            else:
                return "Analysis failed"
                break
```

Figura 6. Trecho da função *getAnalysis*

Como não é viável depender de um iPhone, a solução foi extrair todas as informações listadas no arquivo $Info.plist$ de cada aplicativo, que contém basicamente as permissões que o usuário concede para usar a aplicação e outras configurações do próprio aplicativo que os próprios desenvolvedores introduzem. De acordo com a documentação da Apple, o $Info.plist$ é um arquivo especial que contém metadados usados pelo sistema para identificar o aplicativo e os tipos de documentos que ele suporta.

Para executar a extração, usamos a função *extractInfoPlist*, que recebe o nome do arquivo como parâmetro. Primeiramente, é preciso abrir o arquivo .ipa, para isso criamos um diretório temporário usando a biblioteca *Tempfile* e, em seguida, abrimos o arquivo no formato zip, usando a biblioteca *Zipfile*, e extraímos todo o conteúdo. De posse do arquivo $Info.plist$, usamos a biblioteca *Plistlib* para ler corretamente todos os dados do arquivo, e a função retorna todos os dados no formato JSON. Qualquer tipo de falha será registrada no arquivo CSV final.

Por fim, o módulo de **geração** é acionado sempre que o resultado da rotulação e extração estiverem prontos. A função *generateCSV* recebe o nome do arquivo, o

[16] https://frida.re/docs/home/
[17] http://www.cycript.org
[18] https://rada.re/n/
[19] https://github.com/S3Jensen/iRET
[20] https://github.com/KJCracks/Clutch
[21] https://github.com/swigger/gdb-ios

resultado da extração e da rotulação como parâmetros e adiciona uma linha no arquivo previamente criado intitulado *result.csv* cujo o conteúdo está na Figura 7.

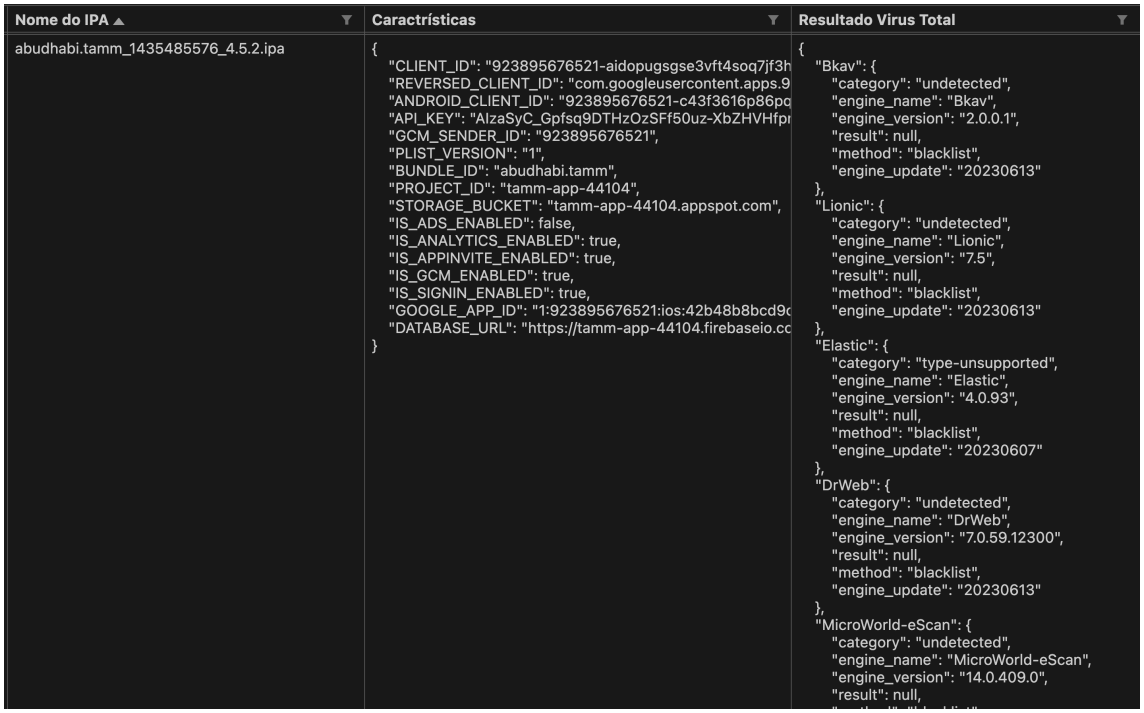

Figura 7. Resultado do arquivo result.csv

Em nossos testes, o processo completo, após o download, de rotulação, extração e geração para cada aplicação leva em média de 3 a 5 minutos, de acordo com o tempo de resposta dos resultado dos *scanners* do VirusTotal. O processo da seguimento para outra aplicação a cada 15 segundos para evitar problemas com as requisições ao VirusTotal.

4. Resultados

Nesta seção apresentamos e discutimos os resultados da ferramenta, incluindo o ambiente de experimentação, sua execução e uma descrição do *dataset* gerado.

4.1. Ambiente

Os testes foram executados em um ambiente com as seguintes configurações: processador Intel® Core™ i7-9750H 2.60GHz 6 Core; Placa de vídeo AMD Radeon Pro 5300M 4 GB; Memoria Ram de 16 GB 2667 MHz DDR4 e; sistema operacional MacOS Ventura 13.4.

4.2. Experimentos

A Tabela 1 apresenta os resultados da execução do iOSDBuilder em três conjuntos distintos contendo 100 arquivos IPA, divididos por grupo de aplicativos de menor ou iguais a 5 MB, até 50 MB e maiores ou iguais a 100MB. As métricas coletadas foram o tempo médio (em segundos), uso de CPU (em porcentagem) e uso de memória RAM (em KiB).

Analisando o módulo de **rotulação**, nota-se que é o que mais consome tempo. Isso se deve ao simples fato de que toda rotulação tem uma espera de no mínimo

Tabela 1. Resultados dos Experimentos.

Testes (média)		Módulos		
Métrica	**Tamanho**	**Rotulação**	**Extração**	**Geração**
Tempo (s)	$\leq$ 5 MB	144,63	0,05	0,001
	50 MB	200,91	0,40	0,001
	$\geq$ 100 MB	234,87	2,21	0,001
CPU (%)	$\leq$ 5 MB	7,34	10,44	0,96
	50 MB	6,98	14,26	5,60
	$\geq$ 100 MB	6,21	12,90	3,51
RAM (KiB)	$\leq$ 5 MB	792,40	0,40	0,00
	50 MB	5.533,57	2,04	0,00
	$\geq$ 100 MB	16.799,60	1,71	0,00

três minutos para aguardar o resultado da rotulação. O tempo médio vária de dois a quatro minutos, mas quanto maior o arquivo mais tempo o VirusTotal demora para devolver o resultado. Em relação ao consumo de processamento, é possível notar que existe uma constância na média de uso. Isso acontece porque o módulo de **rotulação** executa em um laço de repetição, onde verifica se existe resultado de rotulação a cada 3 minutos. Já sobre o uso de memória, podemos observar que quanto maior o aplicativo mais memória é consumida. Isso ocorre porque o *upload* do arquivo, que utiliza a biblioteca *requests* do Python, usa a memória RAM para armazenar o arquivo que vai ser mandado via a requisição POST.

Sobre o módulo de **extração**, podemos associar os resultados ao tamanho das aplicações em análise. O uso de CPU, em comparação aos outros módulos, é muito maior, pois este módulo, além tratar com a extração de arquivo, navega pelo conteúdo extraído em busca do arquivo $InfoPlist$ e transforma o próprio arquivo em um texto JSON. Notamos também que o tempo aumenta conforme o tamanho do arquivo. Por fim o módulo de **geração** é o que menos consome poder computacional, devido ao método apenas possuir tarefas simples de leitura e gravação em arquivos. Podemos ver que sua execução é muito rápida e não depende do tamanho dos arquivos.

A Tabela 2 apresenta os testes para o módulo de **download** (duas funções) que tiveram uma abordagem diferente. Monitoramos tempo de execução, consumo de CPU, uso de memória e tráfego de download e upload para 100 aplicativos.

Tabela 2. Resultados dos Experimentos.

Métricas:	**Tempo (m)**	**CPU (%)**	**RAM (KiB)**	**Download (Mb/s)**	**Upload (Mb/s)**
DownloadAppStore	81,98	0,4	2912,00	12260,53	80,45
DownloadNonAppStore	10,97	15,4	276872,00	5876,58	19,10

Podemos observar que na função *DownloadAppStore*, o tempo levado para concluir o download de 100 aplicativos é muito superior a função *DownloadNonAppStore*. Essa diferença é devido ao uso da ferramenta *IPATool*, o que inclui a verificação da licença do aplicativo antes do download, o que aumenta o tempo. O uso de CPU e memória é menor, uma vez que a nosso código apenas chama a IPATool, que é otimizada para esse função.

Já a função *DownloadNonAppStore*, que apresenta um tempo menor, uma vez que apenas coleta os aplicativos de sites, consome mais CPU, dado todo o processo de análise e verificação da estrutura da página web, ou seja, identificação de todos os possíveis links que possam conter arquivos IPA. Também nota-se que o uso de memória é bem mais elevado que na outra função. O motivo é o processamento interno da página HTML.

Comparando as duas funções, percebemos que elas variam conforme a disponibilidade dos serviços, velocidade da internet, tamanho da aplicação e o tempo de resposta dos sites. Tendo isso em vista, e usando a mesma máquina que usamos para rodar os testes, percebemos que o tempo total de download de 100 aplicativos com as duas funções rodando em paralelo é de **17,27 minutos**. A partir destes dados podemos estimar que levaria **34,54 minutos** para baixar 200 aplicativos, **2,8 horas** para baixar 1.000 aplicativos e **1 dia e 2 horas** para 10.000 aplicativos.

4.3. Dataset Exemplo

Para validar a ideia desta pesquisa, montamos um *dataset* composto pelo processamento de 176 aplicativos coletados, rotulados e extraídos, sendo 48 extraídos da *App Store* e 128 de diferentes sites. De acordo com os *scanners* do VirusTotal, nove (9) aplicativos receberam o rótulo de *malicious*, sendo dois deles com dois ou mais *scanners* acusando-os de maliciosos e os outros sete (7) com apenas um *scanner*. Dessas 9 ocorrências, cinco (5) eram de aplicativos coletados por sites fora da *App Store*. Todas as 5 ocorrências foram identificadas pelo antivírus **ClamAV**. Outro ponto interessante é que mesmo aplicativos baixados diretamente da *App Store* foram detectados como maliciosos pelos scanners do VirusTotal. Foi o caso do aplicativo *Fonts*, cujo *bundleID* é *fonts.app_1480383173_3.44.ipa*, que teve dois *scanners* identificando-o como malicioso, o antivírus do **Google** e do **Ikarus**.

Sobre as características extraídas de cada aplicativo, podemos informar que alguns apresentam grandes quantidade de permissões, como no caso do **whatappwatusi.ipa**, baixado fora da *App store*. Esta aplicação solicita ao todo 17 permissões, incluindo uso do calendário, da câmera, acesso aos contatos, uso do *FaceID*, acesso a internet, acesso a dados relacionados a localização, uso do microfone, acesso a biblioteca de fotos, entre outros. Outros aplicativos apenas possuem dados relacionados a variáveis globais, como *ApiKeys*, endereço do banco de dados, ID do *Firebase*, entre outros. O ponto é que não existe uma definição formal de quantas são as permissões que um aplicativo pode ter. Este *dataset* pode ser consultado no projeto no seguinte diretório: *ios-apps-dataset-builder/Tests/DatasetExemplo.csv*

5. Considerações Finais

Este artigo apresentou a iOSDBuilder, uma ferramenta voltada para geração de *datasets* atualizados capazes de serem empregados na detecção de *malwares* em aplicativos iOS. A iOSDBuilder é composta por quatro módulos independentes, rodando de maneira paralela e automatizada. Até a escrita deste artigo, em diferentes testes, a ferramenta foi capaz de baixar mais de 2400 aplicativos e trazer o resultado dos *scanners* do VirusTotal de aproximadamente 800 aplicativos por dia. Em um teste que durou 11 horas, foram coletados 176 aplicativos, onde 33 constaram nos *scanners* do VirusTotal como maliciosos. Vale destacar que o tempo total de execução

pode diminuir ou aumentar conforme a quantidade de links de sites inseridos para serem escaneados. Em uma hora de execução, a iOSDBuilder é capaz de devolver resultado de até 70 aplicativos.

Acreditamos que esta ferramenta será de grande ajuda para comunidade devido à escassez de informações e *datasets* para serem alimentados em modelos de identificação de *malwares* voltados para iOS.

Atualmente o projeto está configurado para ser executado uma única vez. Um trabalho futuro é aprimorar sua execução, para que o *script* fique executando sempre a fim de coletar os aplicativos disponíveis e permanecer buscando por novos aplicativos. Outra melhoria é o aprimoramento da extração de características, para ser possível extrair o máximo de informações de dentro dos aplicativos.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Este trabalho foi parcialmente financiado pela Fundação de Amparo à Pesquisa do Estado do Amazonas – FAPEAM – por meio do projeto POSGRAD.

Referências

Ciaramella, G., Iadarola, G., Martinelli, F., Mercaldo, F., and Santone, A. (2022). A model checking-based approach to malicious family detection in ios environment. volume 207.

Cimitile, A., Martinelli, F., and Mercaldo, F. (2017). Machine learning meets ios malware: Identifying malicious applications on apple environment. volume 2017-January.

Felt, A. P., Finifter, M., Chin, E., Hanna, S., and Wagner, D. (2011). A survey of mobile malware in the wild.

Garg, S. and Baliyan, N. (2021). Comparative analysis of android and ios from security viewpoint.

Husainiamer, M. A., Saudi, M. M., Ahmad, A., and Syafiq, A. S. M. (2021). Mobile malware classification for ios inspired by phylogenetics. *International Journal of Advanced Computer Science and Applications*, 12.

Kouliaridis, V., Barmpatsalou, K., Kambourakis, G., and Chen, S. (2020). A survey on mobile malware detection techniques. *IEICE Transactions on Information and Systems*, E103D.

Sharma, T. and Rattan, D. (2021). Malicious application detection in android - a systematic literature review.

Vilanova, L., Kreutz, D., Assolin, J., Quincozes, V., Miers, C., Mansilha, R., and Feitosa, E. (2022). Adbuilder: uma ferramenta de construção de datasets para detecção de malwares android.

Yixiang, Z. and Kang, Z. (2017). Review of ios malware analysis.