UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Caio Augusto Pereira Burgardt

Malware detection in macOS using supervised learning

Recife

2022

Caio Augusto Pereira Burgardt

Malware detection in macOS using supervised learning

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação em 25 de fevereiro de 2022.

Área de Concentração: Redes de Computadores e Sistemas Distribuídos

Orientador: Divanilson Rodrigo de Sousa Campelo

Recife

2022

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

B954m Burgardt, Caio Augusto Pereira
Malware detection in macOS using supervised learning / Caio Augusto Pereira Burgardt. – 2022.
51 f.: il., fig., tab.

Orientador: Divanilson Rodrigo de Sousa Campelo.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2022.
.
Inclui referências.

1. Redes de Computadores. 2. Aprendizagem de máquina. I. Campelo, Divanilson Rodrigo de Sousa (orientador). II. Título.

004.6 CDD (23. ed.) UFPE - CCEN 2022-117

Caio Augusto Pereira Burgardt

“Malware detection in macOS using supervised learning”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Redes de Computadores e Sistemas Distribuídos.

Aprovado em: 25/02/2022.

BANCA EXAMINADORA

_____

Prof. Dr. Daniel Carvalho da Cunha
Centro de Informática / UFPE

_____

Prof. Dr. Rafael Timóteo de Sousa Júnior
Departamento de EngenhariaElétrica / UnB

_____

Prof. Dr. Divanilson Rodrigo de Sousa Campelo
Centro de Informática / UFPE
(**Orientador**)

This thesis is dedicated to my parents, Cristiane and Otávio, for their unconditional love and support throughout my life. I'd also like to thank my brother Victor and my girlfriend Mariana for helping me manage my stress and always being there.

ACKNOWLEDGEMENTS

Firstly, I'd like to thank my Master's advisor Prof. Divanilson Rodrigo Campelo, for his patience, his insight and his friendship in the last 5 years. I would simply not be where I am were it not for him. It's an honor to be part of his team.

Second of all, I'd like to thank his research team, in particular Paulo Freitas Araujo for his help and insights on machine learning. His advice has been very important to this research.

I'd also like to thank the Centro de Informática of UFPE for the opportunity of learning from the best while giving the tools needed to perform the best despite the current dark times.

ABSTRACT

The development of macOS malware has grown significantly in recent years. Attackers have become more sophisticated and more targeted with the emergence of new dangerous malware families for macOS. However, since the malware detection problem is very dependent on the platform, solutions previously proposed for other operating systems cannot be directly used in macOS. Malware detection is one of the main pillars of endpoint security. Unfortunately, there are very few works on macOS endpoint security, which is considered a largely unexplored territory. Currently, the only malware detection mechanism in macOS is a signature-based system with less than 200 rules as of 2021, called XProtect. Recent works that attempted to improve the detection of malwares in macOS have methodology limitations, such as the lack of a large macOS malware dataset and issues that arise with imbalanced datasets. In this work, we bring the malware detection issue to the macOS operating system and evaluate how supervised machine learning algorithms can be used to improve endpoint security in the macOS ecosystem. We create a new and larger dataset of 631 malware and 10,141 benign software using public sources and extracting information from the Mach-O format. We evaluate the performance of seven different machine learning algorithms, two sampling strategies and four feature reduction techniques in the detection of malwares in macOS. As a result, we present models that are better than macOS native protections, with detection rates larger than 90% while maintaining a false alarm rate of less than 1%. The presented models successfully demonstrate that macOS security can be improved by using static characteristics of native executables in combination with common machine learning algorithms.

Keywords: malware; machine learning; macos.

RESUMO

O desenvolvimento de malware para macOS cresceu significativamente nos últimos anos. Os invasores se tornaram mais sofisticados e mais direcionados com o surgimento de novas famílias de malware perigosas para o macOS. No entanto, como o problema de detecção de malware é muito dependente da plataforma, as soluções propostas para outros sistemas operacionais não podem ser usadas diretamente no macOS. A detecção de malware é um dos principais pilares da segurança de endpoints. Infelizmente, existem muito poucos trabalhos sobre a segurança de endpoint do macOS, que é considerada território pouco investigado.Atualmente, o único mecanismo de detecção de malware no macOS é um sistema baseado em assinaturas com menos de 200 regras em 2021, conhecido como XProtect. Trabalhos recentes que tentaram melhorar a detecção de malwares no macOS têm limitações de metodologia, como a falta de um grande conjunto de dados de malware do macOS e problemas que surgem com conjuntos de dados em classes desequilibradas.Nessa dissertação, trazemos o problema de detecção de malware para o sistema operacional macOS e avaliamos como algoritmos de aprendizado de máquina supervisionados podem ser usados para melhorar a segurança de endpoint do ecossistema macOS. Criamos um novo dataset extraindo informações do formato Mach-O de 631 malwares e 10.141 softwares benignos de fontes públicas. Avaliamos o desempenho de sete algoritmos de aprendizagem de máquina em conjunto com duas estratégias de amostragem e quatro técnicas de redução de features para a detecção de malwares no macOS. Como resultado, apresentamos modelos melhores que as proteções nativas do macOS, com taxas de detecção superiores a 90% e taxas de alarmes falsos inferiores a 1%. Os modelos apresentados demonstram com sucesso que a segurança do macOS pode ser aprimorada usando características estáticas de executáveis nativos em combinação com algoritmos populares de aprendizagem de máquina.

Palavras-chaves: malware; aprendizagem de máquina; macos.

LIST OF FIGURES

LIST OF SOURCE CODES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ANOVA	Analysis of Variance
AUC	Area Under Curve
CA	Certificate Authority
DR	Detection rate
ELF	Executable and Linkable Format
FAR	False Alarm Rate
FN	False Negative
FP	False Positive
LIEF	Library to Instrument Executable Formats
MLP	Multilayer Perceptron
MRT	Malware Removal Tool
PCA	Principal Component Analysis
PE	Portable Executable
PUPs	Potentially Unwanted Programs
ROC	Receiver Operating Characteristic
SIP	System Integrity Protection
SMOTE	Synthetic Minority Oversampling Technique
SVM	Support Vector Machine
TCC	Transparency, Consent and Control framework
TN	True Negative
TP	True Positive

CONTENTS

1 INTRODUCTION

Malware is software that performs malicious actions, such as stealing passwords, obtaining confidential files, and getting remote access to a device (SIKORSKI; HONIG, 2012). As an attack technique, malwares can be very versatile and may cause damage to end-users and corporations alike. Even though protection against malwares is a well-discussed subject in the literature, especially for Windows and Android ecosystems, a change in end user's technology results in an adaptation by the attacker.

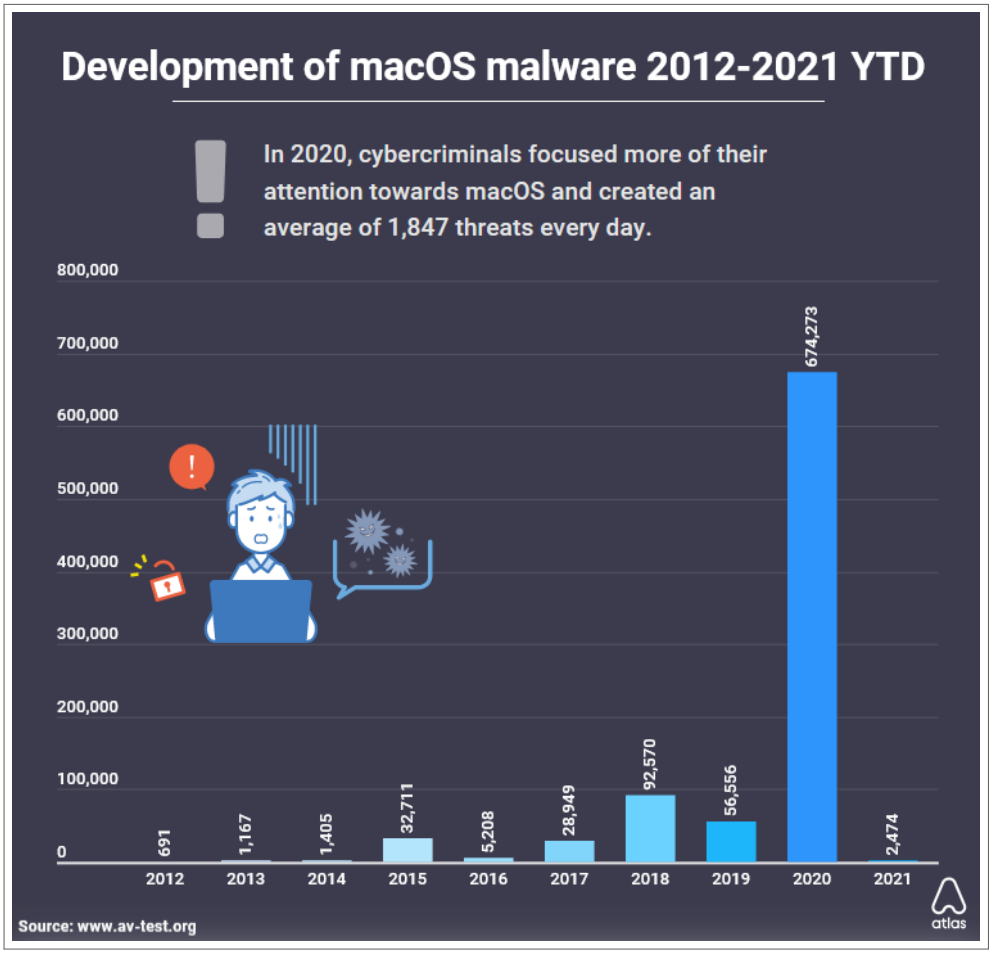
Recently, the macOS operating system has become a more popular target among attackers. The Apple's environment, which used to be claimed in advertisements as a "no viruses" operating system (CLULEY, 2012), has witnessed a major increase in malicious software attacks. As shown in (MCAFFEE, 2017), the detection of malware for macOS skyrocketed in the 2015-2016 period, with a 744% increase. According to (G., 2021), macOS malware development outpaced that of Windows in 2020, with a surge of over 1000% new malwares since 2019, as shown in Figure 1.

As with any modern operating system, macOS has its own embedded security features that create difficulty barriers for attackers. The main malware protection system of the macOS environment is the *Gatekeeper* mechanism. Its main focus is to prevent the execution of malware by verifying the code signature of the executable. The code signature proves that an executable was compiled by an identified developer, therefore more trustworthy (APPLE, 2016b).

Apple has been making it harder for malware developers by creating strict requirements for the OS to run software not distributed by the App Store. In order to allow default execution of applications, the software must not only be signed but also notarized by Apple, which is a process where the app is submitted to Apple so they perform security checks. As of 2021, the actual notarization process is not disclosed to the public. Without notarization, a user that attempts to run an executable downloaded through the Web will be met with a *Gatekeeper* warning that will prevent the execution unless the user manually changes settings in the macOS *System Preferences* (APPLE, 2021c).

Previously, the macOS platform had a separate anti-malware mechanism called *XProtect*, which has been recently integrated into the *Gatekeeper* system. *XProtect* is a simple static

Figure 1 – MacOS malware development ratio over the years until March 2021



Source: AtlasVPN and av-test.org

signature-based malware detection system that uses a small list of YARA[1] rules and a deny list of developer IDs to try and match malicious software before executing them. In the case an attacker can execute malware in the macOS system, there are still different mechanisms that attempt to mitigate damage, such as the System Integrity Protection (SIP), which makes some system files not writable, and the Transparency, Consent and Control framework (TCC), which requires the apps to explicitly ask permission from the user in order to perform actions such as turning on the camera, or using the microphone. However, even with all these mechanisms, malware and adware sometimes are mistakenly notarized by Apple and distributed as benign software, as shown in the 2020 state of malware report from *MalwareBytes* (MALWAREBYTES, 2020).

These weaknesses can be attributed to the lack of diversity in detection techniques embedded in the operating system. While researchers have investigated different techniques to

[1] A popular tool used to identify and classify malware via pattern-matching.

detect malware, such as behavior-based and machine learning-based approaches, malware detection in the macOS itself is only signature-based, which is incapable of detecting unknown malware and is generally trivial to bypass (GANDOTRA; BANSAL; SOFAT, 2016). Furthermore, it seems that signatures are not frequently updated, since we tested current *XProtect* signatures against a dataset of publicly known malware and got a detection rate lower than 10%.

1.1 OBJECTIVES AND GOALS

In this master's thesis, we investigate the usage of supervised machine learning algorithms as detection techniques to classify whether Mach-O binaries are malicious. We built a dataset out of the two most popular software repositories for macOS users and two public available free malware repositories for research. Subsequently, with common machine learning algorithms and only static features, we show that the malware detection rate in macOS can be increased to over 90% with false alarm rates lower than 1%.

1.2 MAIN CONTRIBUTIONS

The primary contributions of this Master's thesis are as follows:

- We highlight the methodology limitations regarding current macOS detection research, mostly related to the use of a small and unbalanced dataset and bad performance metrics. To overcome these limitations, we have built a macOS malware detection dataset from public sources, resulting in the largest public macOS malware detection dataset up to date.
- We have evaluated different algorithms, sampling strategies and feature reduction techniques to mitigate the issues from the dataset, namely the class imbalance and number of malicious samples.
- We have compared the presented models with current macOS signature-based detection and shown how the usage of machine learning in static malware detection for macOS can be a powerful endpoint security ally thanks to the very small false alarm rate in the some models.

1.3 GENERATED WORKS

The results of this thesis generated the paper *"Orange Among Apples: Refining static malware detection in macOS"* with the authors Caio A. P. Burgardt, Victor A. P. Burgardt, and Divanilson R. Campelo, which is currently being finalized to be submitted to a journal.

We also successfully published a paper not directly related to this thesis during my Master's work in the Computer Communications journal. Its title is "Identifying IoT devices and events based on packet length from encrypted traffic", from the authors Antonio J. Pinheiro, Jeandro de M. Bezerra, Caio A.P. Burgardt and Divanilson R. Campelo (PINHEIRO et al., 2019).

1.4 THESIS OVERVIEW AND LAYOUT

The rest of this thesis is described as follows. In Chapter 2 we take a brief look at the malware problem, in specific for the macOS/OS X environment. We also investigate related works on the issue of static malware detection. In Chapter 3, we describe the process of building the database of malware and benign software from where we extracted the static features. Chapter 4 presents the metrics and the machine learning algorithms used in our methodology. In Chapter 5, we present the evaluation of the different machine learning models and refine the models. Finally, we conclude the thesis by discussing the results and pointing to future works and research opportunities in Chapter 6.

2 BACKGROUND

As previously mentioned, malware is an old attack vector, so there has been plenty of research done into both creating protections and evading those protections. In this chapter, we present an introduction to the challenge of the fight against malware.

2.1 MALWARE

According to (SIKORSKI; HONIG, 2012), malware is malicious software that plays a part in most computer intrusion and security incidents. Any software that does something that causes harm to a user, computer, or network can be considered malware, including viruses, trojan horses, worms, rootkits, scareware, and spyware.

2.1.1 Malware types and their behavior

Malware can be as diverse as an average benign software. They can be targeted at a different public, with different objectives and techniques. Predicting malicious behavior is a very difficult ordeal since maliciousness can be very subjective. In order to better facilitate an understanding of malicious behavior in software, we may attempt to classify malware according to its general objective. Unfortunately, there is no academic consensus on which classification scheme we should rely on.

There are many different attempts at classifying malware, generally with each class pointing to a specific action. Because of increasing malware complexity, often malware ends up belonging to multiple different categories. Some malware are pretty straightforward and focus on getting remote access to the attacker while some may be more subtle and focus on spying on the user by logging keystrokes. Some malware may be developed for mass infection and self replicate while some others are designed to run only once and make investigation harder. Malware may have many of these characteristics or none of them.

Because of the subjectivity of defining malicious behavior, there are also types of software with malware-like actions, such as Potentially Unwanted Programs (PUPs). Examples of such applications are toolbars, popups, browser extensions, or other types of Adware with the purpose of aggressively serving advertisements to the victim. With the advent of cryptocurrency,

some "malware" has the objective of borrowing some of the victim's computing power to mine their chosen cryptocurrency.

2.1.2 MacOS as a target

Apple's ecosystem was marketed as a "PC virus-free" system until 2012, implying that the OS X system is more secure because it does not have to deal with the malware issue plaguing Windows-based systems (CLULEY, 2012). While it is true that Windows malware do not generally work in macOS systems, Apple has its own history with malware, spamming long enough that it reaches the 80's (CLULEY, 2011).

In macOS, malware mostly takes the form of PUPs and Adware, as shown by the state of malware report 2021 from MalwareBytes (MALWAREBYTES, 2020), PUPs accounted for 76% of the infections, Adware for 22% and other more malicious forms of software only by 1.5%. However there still is dangerous malware in macOS. In the same year, *ThiefQuest* was discovered, what seemed to be the first macOS ransomware since 2017. Further analysis of *ThiefQuest* showed that the ransomware functionality is a disguise for its actual objective: mass file exfiltration (STOKES, 2020b).

In 2021, macOS malware has taken some alarming directions, while Adware and PUPs are still the most common type of malware seen. Most new malware families that emerged in 2021 focused on espionage and data theft. Attacks have been getting creative and more specific, such as *XCodeSpy*, which is a malicious *XCode* iOS project with a run script that drops a Mach-O executable that is a shell able to steal information from the victim's microphone and camera (STOKES, 2021).

2.2 MACOS DEFENSE MECHANISMS

The macOS system has its own embedded security mechanisms. Apple has built the macOS endpoint security in three layers (APPLE, 2021e):

1. **Prevent** launch or execution of malware.
2. **Block** malware from running on customer systems.
3. **Remediate** malware that has executed.

The first layer is focused on the prevention of the malware from ever launching. In this layer, Apple has implemented a restrictive policy of distribution of software in their platform as an attempt to gate keep malicious software out of the end user's reach.

The second layer's objective is to detect and block already downloaded malware. It is composed of different mechanisms that identify the malware quickly and work as a traditional anti-virus software.

The third layer kicks in when the other two failed and the malware achieved execution. Its purpose is to remediate and mitigate damage caused from the malicious execution.

2.2.1 First layer: AppStore and Notarization

The spread of malware is highly related to the users' difficulty to decide if they can trust the origins of the file they're executing. To overcome this issue, one may use code signing, which is the process of cryptographically signing executables with an author's private key so that the user may verify the binary's integrity using the author's public key (APPLE, 2016a).

Code signatures give the user the mechanism to authenticate the origin of the executable, but they require a trusted Certificate Authority (CA) to vouch for the certificates sent with the executable. Apple has chosen to centralize the trust. To achieve that, developers for the Apple ecosystems require an Apple Developer ID with a membership in the payed Apple Developer Program. With this Developer ID, developers may request for code signing certificates.

With these code signing certificates, applications are linked to their developers and their Developer ID. In case of attackers distributing malware, certificates are revoked and the attackers lose their Developer ID. By default, applications are not executable without a valid Developer ID code signature (APPLE, 2016a).

The main distribution mechanism, the *App Store*, has several publishing requirements to maintain app quality, including security ones (APPLE, 2021a). In case a piece of software is not delivered by the *App Store*, it has to pass the notarization process, in which Apple performs a series of security checks. The notarization process itself is a secret and not publicly known. It is advertised as an automated scan for malicious components (APPLE, 2021d).

As of February of 2020, non Mac App Store apps must pass the app notarization process to be executable by default (APPLE, 2019b). Apps that are not notarized can still be executed by users, although it requires a bypass or disable of the *Gatekeeper* security mechanism.

2.2.2 Second Layer: Gatekeeper and XProtect

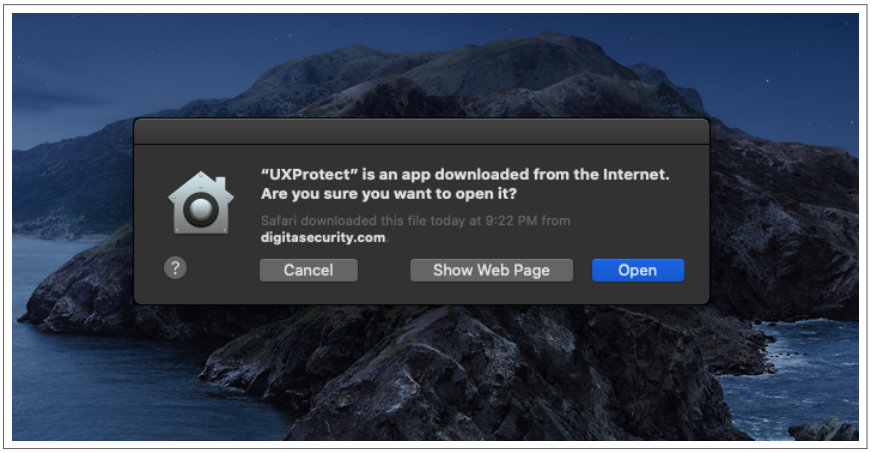
The main endpoint security mechanism on the macOS system is the *Gatekeeper*. When a user downloads a file from the internet, the files are marked with a quarantine extended attribute. This attribute tells the *Gatekeeper* that the file needs to be checked for trustworthiness. A user that executes the marked executable will trigger a warning prompt like the one shown in Figure 2.

The *Gatekeeper* is responsible for verifying the *Developer ID*, code signature and if the software has been notarized by Apple (APPLE, 2021b). *Gatekeeper* can be configured to be more permissive or completely disabled.

In macOS 10.15, the macOS native malware detection mechanism, *XProtect*, has been integrated with *Gatekeeper*. *XProtect* is a signature-based malware detection system based on YARA signatures that triggers in the following scenarios (APPLE, 2021e):

1. Every time a software is first executed.
2. Every time a software is altered or tampered.
3. Every time *XProtect's* signatures are updated.

Figure 2 – Gatekeeper execution warning



Source: The author (2022)

YARA is a tool to help malware researchers to identify and classify malware samples (ALVAREZ, 2021). It gives researchers a language to write rules that define patterns which the YARA engine tries to locate in files. In *XProtect*, the YARA rules are found in the resources of the *XProtect* app bundle. A snippet of the file can be seen in Source Code 1.

Source Code 1 – YARA Rule Examples from XProtect.yara

```
rule BundloreA
{
    meta:
        description = "OSX.Bundlore.A"
    strings:
        $a = {5F 5F 6D 6D 5F 67 65 74 49 6E 6A 65 63 74 65 64 50 61 72 61 6D 73}
        $b = {5F 5F 6D 6D 5F 72 75 6E 53 68 65 6C 6C 53 63 72 69 70 74 41 73 52 6
            F 6F 74}
    condition:
        Macho and ($a and $b)
}

rule GenieoE
{
    meta:
        description = "OSX.Genieo.E"
    strings:
        $a = {47 4E 53 69 6E 67 6C 65 74 6F 6E 47 6C 6F 62 61 6C 43 61 6C 63 75 6
            C 61 74 6F 72}
        $b = {47 4E 46 61 6C 6C 62 61 63 6B 52 65 70 6F 72 74 48 61 6E 64 6C 65
            72}
    condition:
        Macho and ($a and $b)
}
```

Source: The Author (2022).

In Code 1, we see rules for the malware "*OSX.Bundlore.A*" and "*OSX.Genieo.E*". In both rules, the researcher defined strings of bytes that when they are all in the binary, the detection condition becomes true. The condition also requires the binary to be a Mach-O file.

XProtect not only uses YARA signatures, but also holds databases of blacklisted *Team IDs* and *Safari Extensions* (STOKES, 2020a).

2.2.3 Third Layer: Malware Removal Tool

While *XProtect* detects the malware, the Malware Removal Tool (MRT) removes it. The inner workings of the MRT are not published by Apple, but researchers have analysed its behavior (OAKLEY, 2020). MRT is triggered whenever it is updated and it uses a rule file to find and remove different malware families.

2.2.4 Further Protection: Containing damage

When all the layers fail, macOS has mechanisms to mitigate damage as much as possible.

2.2.4.1 App Sandbox

App Sandbox restricts access to system resources and user data in macOS apps to contain damage if an app becomes compromised (APPLE, 2022a). As of 2022, apps must have the sandbox enabled in order to be distributed in the App Store (APPLE, 2022a).

2.2.4.2 Transparency, Consent, and Control (TCC)

TCC offers granular control of privacy by asking whether or not applications may perform privacy-sensitive actions, such as accessing the microphone, camera and reading user files (IFERT-MILLER, 2019). The database which controls the TCC can be found in **/Users/<USER>/Library/Application Support/com.apple.TCC/TCC.db** .

2.2.4.3 System Integrity Protection (SIP)

The SIP is a security system introduced in OS X 10.11 that stops the *root* user from writing to specific system directories (APPLE, 2019a). The idea is to limit the impact of malicious code by protecting the critical parts of the system, such as the following:

- /System
- /usr
- /bin
- /sbin
- /var
- Directories of Apps that are pre-installed with the system.

In order to write to these locations, apps must be signed by Apple and have specific entitlements. Users may also disable the SIP entirely if they wish by rebooting the system in *Recovery mode* and typing specific commands in the terminal (APPLE, 2022b).

2.3 RELATED WORK

2.3.1 Static malware detection

There has been significant progress in the fight against malware. While some of the recent focus turned towards dynamic detection, static detection is constantly revisited by researchers as these techniques allow for malware detection before it actually runs, preventing any damage.

Many researchers have been using supervised learning approaches to tackle the malware detection challenge. Most of them focused on Windows and its native executable format, the Windows Portable Executable (PE) format. Among the first works on the subject, (SCHULTZ et al., 2000) used a Multi-Naive Bayes method using PE headers, strings and byte sequences to create a detection model with accuracy of 97.76%. The authors in (KOLTER; MALOOF, 2004) extracted over 255 million distinct n-grams of benign and malicious software and, by using a boosted decision tree, they achieved 0.996 area under Receiver Operating Characteristic (ROC) curve. These results indicated that it is possible to detect unseen Windows malware through machine learning with a high detection rate.

The authors in (BALDANGOMBO; JAMBALJAV; HORNG, 2013) have used data mining methods to create static malware detection from PE header values and API functions called. The authors used information gain and Principal Component Analysis (PCA) to reduce the dimension of the problem and achieved a detection rate of 99.6% and a false positive rate of 2.7%.

The authors in (LO; PABLO; CARLOS, 2016) achieved an accuracy of 99.60% using only 9 chosen features, static and dynamic, against a Windows malware dataset of 14,902 total samples using a combination of Random Forest, Support Vector Machine (SVM) and Neural Networks. More recently, researchers in (MANAVI; HAMZEH, 2021) used deep learning techniques such as Long Short-Term Memory (LSTM) to detect ransomware samples from only PE header information with a 93.5% accuracy.

Windows has not been the only platform investigated in the literature, as many other authors have also visited the mobile Android platform. In (MILOSEVIC; DEHGHANTANHA; CHOO,

2017), the authors performed two static feature analyses, a source code-based analysis and a permission-based analysis. Both approaches were tackled with ensemble learning models with an F-measure of 95.1% and 89%, respectively, showing that static features can be very effective determinants of maliciousness in Android systems.

2.3.2 MacOS/OSX malware detection

With the recent surge of macOS malware appearances, some researchers investigated the endpoint security issues in macOS. In forensics investigation, the research in (CASE; III, 2015) proposed new methodologies to detect rootkit malware in OS X systems. From the perspective of malware analysis environments, the authors in (PHAM; VU; MASSACCI, 2019) developed a virtualizing OS X platform to help the investigation of macOS malware by extracting static and dynamic features of malware more efficiently than current solutions at the time.

Regarding research with malware detection as the main investigation point, the author in (MIEGHEM, 2016) built a heuristic-based approach to detect malware. The author studied the imported libraries by 21 macOS malware and came up with four patterns that achieved a detection rate of 100%, which as shown in their work, could result in a high false alarm rate of 25%.

The authors in (PAJOUH et al., 2018) attempted a static malware detection approach using machine learning with a smaller dataset composed of only 152 malware and 450 benign software. With the preliminary dataset, the authors achieved a 91% accuracy with a false alarm rate of 3.9%. In light of the size of the dataset, the authors synthetically created samples via Synthetic Minority Oversampling Technique (SMOTE) to make the dataset five times the original size and got a 96% accuracy on the synthetic dataset with a minor hit to false alarm rate reaching 4%. While the metrics may seem positive, the work has some methodology limitations: the usage of accuracy as a performance metric when evaluating results from a dataset with a high degree of class imbalance and using oversampling techniques before the train-test split, effectively detecting malware that do not exist.

In (SAHOO; DHAWAN, 2022), authors handled the same dataset as (PAJOUH et al., 2018), executing SMOTE to completely balance the dataset to a 1:1 benign-malicious ratio and extracting libraries calling in the DYLIB sections via TF-ID based text processing. The authors' model used Logistic Regression and achieved a 96% accuracy and 2.14% false alarm rate. This work has the same methodological limitations as (PAJOUH et al., 2018) regarding the generation

of synthetic samples to fix class imbalance before the train-test split.

Other methods were tested in (GHARGHASHEH; HADAYEGHPARAST, 2022), handling the same dataset as the previously mentioned articles, but with no sampling techniques to deal with the low number of malicious samples. They used Chi-square feature selection to halve the number of features and achieve a 94.7% accuracy using the Subspace KNN ensemble method. A methodological limitation is the usage of accuracy as their main metric for performance, which is not appropriate for dataset with imbalanced classes.

Regarding a contribution from the industry, the authors in (HSIEH; LIU, 2017) used static features with supervised machine learning in order to classify macOS malware using a dataset of over 600 thousand samples from VirusTotal (VIRUSTOTAL, 2021) and 4,000 known malicious samples. The result was not very optimistic, since the best-performed algorithm was a decision tree that achieved a 60% recall score.

3 DATASET

Different operating systems have different native executable binary formats: Linux systems use the Executable and Linkable Format (ELF), Windows uses the PE format and macOS uses the Mach-O format. Even though these file formats may be similar in the fact that they all are executable code with separate memory segments, their loading, linking, and metadata are different. These differences make static malware detection techniques, such as the previously mentioned, very platform-dependent and, as such, models for malware detection in Windows or Android are incompatible with the macOS environment.

In order to properly translate and evaluate known techniques, we need to adapt them to the macOS ecosystem. To the best of our knowledge, as of 2021, there is no public dataset for the macOS malware detection problem. We approached this issue by building a dataset of our own by acquiring malicious and benign Mach-O files and collecting features from the Mach-O format similar to the way features were extracted in previous Windows and Android malware works (ORANGES..., 2021).

3.1 FEATURE EXTRACTION AND THE MACH-O FORMAT

In previously mentioned works, the focus is mostly on Windows malware. Therefore, their feature extraction step takes into account the PE format. Since we tackle the macOS environment, we are forced to work with the Mach-O format and thus with different static features.

For PE files (MICROSOFT, 2021), the static features commonly used are the ones found in the DOS header, the File and Optional headers, the addresses and sizes in the data directory, the resource entries, the DLL imported, the API used and information of important sections. While Mach-O format does not hold the exact type of information, there are some similarities in every executable format, such as separated segments for code and data, and imported libraries (MACH-O..., 2014).

Extracting header information is pretty straightforward in both PE and Mach-O formats. However, the PE format has more metadata on its headers. In fact, it has three different headers: the DOS header, the File header, and the Optional header, which hold data like information about its sections, required Windows version, memory space required, processor architecture, and much more. The Mach-O format starts with the Mach-O header as illustrated

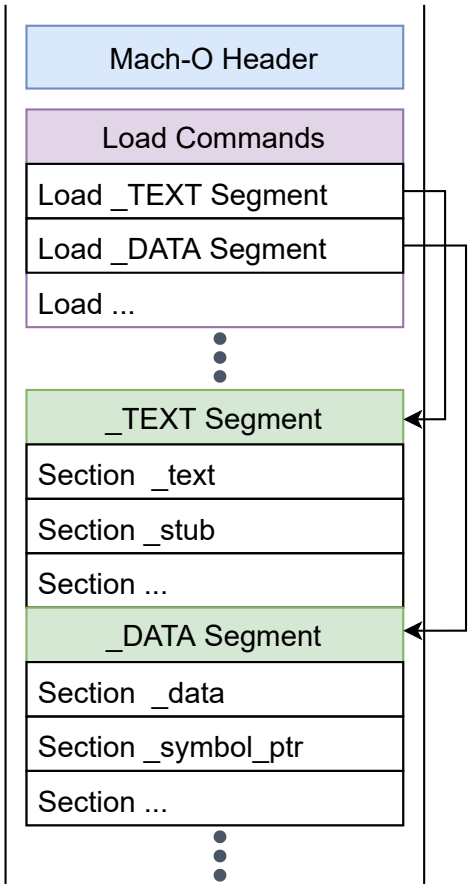
in Fig. 3. The Mach-O Header is only 32 bytes long and holds only the magic number for the Mach-O file, the type of Mach-O executable, architecture, executable feature flags, and the number of load commands.

In the PE file, the Optional Header holds the data directories, which include meta-information about exports and imports, among other things. Right after that, we find the section table holding headers that pertain to the different sections and how each of them should be loaded into the memory.

The Mach-O format does not handle sections, imports, or exports that way. Right after the Mach-O header, there are a number of load commands (specified in the Mach-O header), followed by raw data, as illustrated by Fig. 3. Load commands pose a similar function to the section headers and the PE data directories by pointing to specific chunks in the raw content and how they should be loaded into memory.

There is a variety of load commands for different cases: Special load commands for loading code, data, how libraries should be linked, and even a load command for the code signature.

Figure 3 – The Mach-O format.



Source: Author (2022)

In order to retrieve these features in Mach-O files, we used an open-source project named Library to Instrument Executable Formats (LIEF) (ROMAIN, 2021), which provides an efficient way of playing around with executable formats and extracting interesting information such as the Mach-O header, load commands, section names, exported and imported functions.

Table 1 – Table showing features extracted from the Mach-O File.

Binary resource	Features extracted
Mach-0 header	Target CPU information, Type of file, Number of load commands, size of all load commands, binary flags
Load Segment Commands	Segment name, size of segment, offset, number of sections in segment, flags
Sections	Name, Entropy, Size, Flags
Symbol Command	Number of Symbols, Size of the size string table
Dynamic Symbol Command	Number and index of local and external symbols
Libraries	List of Libraries imported

Source: The author (2022)

3.2 OBTAINING MACH-O SAMPLES

With a proper method of extracting relevant features from Mach-O files, a database of malicious and benign software is required, preferably a database already properly labeled as malicious or not. While there are some public datasets for Windows, unfortunately, the discussion around the detection of macOS malware has not reached the same level of popularity. Therefore it is up to us to build a proper dataset for our tests.

3.2.1 Acquiring Benign Samples

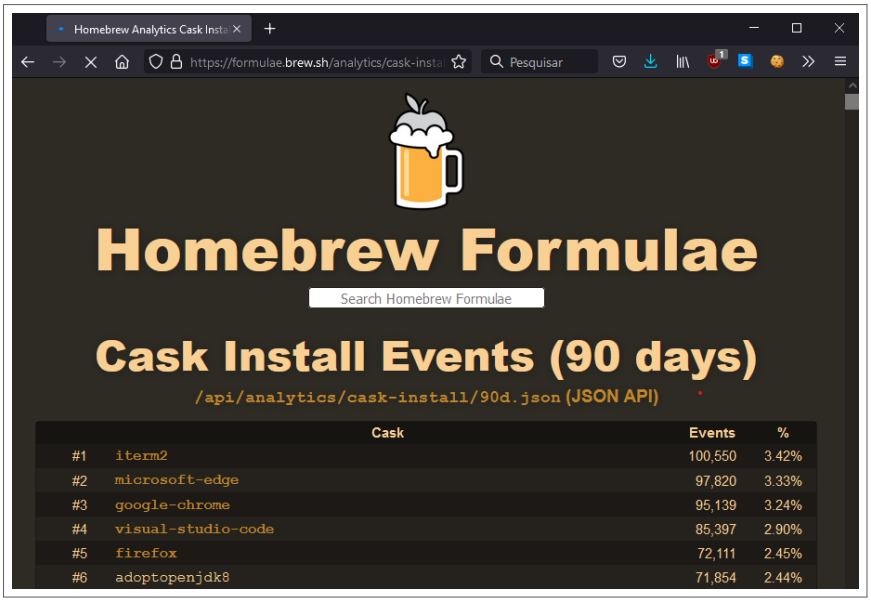
Apple's go-to mechanism for the distribution of applications is its AppStore (APP..., 2021). However, as of 2021, Apple's App Store provides no API for downloading apps automatically, nor does it provide a list of all the applications on it. These barriers greatly complicate a systematic approach to choosing benign software.

In order to build the dataset for our experiments, we manually downloaded the top free apps shown in the AppStore on April 7th and April 15th of 2021, resulting in a total of 215 different applications. Because of the low number of applications downloaded through this method, we

went after a different popular source, a third-party repository for macOS applications known as Homebrew (HOMEBREW, 2021).

Homebrew is widely regarded as the missing macOS package manager (HOMEBREW, 2021), and it provides a repository for Mac applications and Unix tools. Homebrew gives their packages in two different ways: Formulae and Casks. Formulae are mostly open-source software for command-line applications, while Casks are mostly GUI applications, some of them closed source. The Homebrew project also provides analytics of downloads so we can infer the most popular applications (HOMEBREW, 2021), as seen in Figure 4. With that in mind, we created a script to download all the Casks with over a thousand downloads in the last 90 days as of April 14th, 2021, summing up to a total of 246 different applications. After downloading all the applications, we extracted the Mach-O files from those applications and removed duplicates, resulting in 10,141 unique files.

Figure 4 – Homebrew analytics showing top Casks in 90 days.



Source: Author (2022)

3.2.2 Acquiring Malicious Samples

Searching for an extensive free database of macOS malware has proven to be a challenge. The highest-profile macOS malware database is the malware collection of Objective-See, an initiative to research and publish security-related macOS knowledge. As of April of 2021, the collection holds samples for 130 different malware families (MAC..., 2021). The Objective-see

blog not only offers malware samples but also information on how they classify each malware family according to the malware behavior.

Another source was the GitHub repository "Macos-Malware-Samples", which belongs to VirusSamples (MALWARESAMPLES, 2021). The repository provides almost 500 malware samples for use. Unfortunately, there is not much information on their origin, name, or behavior, however, if needed, one can use virus scanners like VirusTotal to try and discover some of this information.

After aggregating all the malwares from these two databases, and filtering out malformed and non-Mach-O samples, we got 178 samples from the Objective-See database and 453 samples from VirusSamples, summing up to a total of 631 malware samples.

Our dataset has a couple of issues we need to take note of when modeling solutions. It currently holds a total of 10,772 Mach-O samples with only 631 being malicious ones, which presents a high-class imbalance, as seen in Table 2. Another possible issue is mislabeling malicious software as benign. We tried to address this issue by using the popularity of the software as a heuristic for what seems to be more trustworthy among macOS users.

Table 2 – Table showing the origin of Mach-O files used in the dataset.

Source	Type	Quantity
AppStore	Benign	4,331
HomeBrewCasks	Benign	5,810
ObjectiveSee	Malicious	178
VirusSamples	Malicious	453

Source: The author (2022)

4 METHODOLOGY

In this chapter, we introduce our methodology by exploring some definitions regarding metrics and the machine learning models used in this thesis. Such models have the objective to classify a given set of features as belonging to malware or benign software. To evaluate the performance of the models, we must choose the best metrics for the scenario.

4.1 MALWARE DETECTION METRICS

The confusion matrix allows us to extract values on how often the model output matched the ground truth. The following metrics can be derived from the output from the confusion matrix:

- True Positive (TP) : The number of correctly classified malware.
- True Negative (TN) : The number of correctly classified benign software.
- False Positive (FP) : The number of benign software misclassified as malware.
- False Negative (FN) : The number of malware misclassified as benign software.
- TP Rate : The rate of correctly classified malware in the test dataset, also known as Detection rate (DR) and recall.
- FP Rate: The rate of benign software misclassified as malware, also called False Alarm Rate (FAR).

Accuracy is another simple commonly used metric, and it represents how often the model gives correct predictions, as shown in Eq. 4.1:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{4.1}$$

Even though accuracy may seem like a good metric to use, it fails to account for dataset imbalances and conveys little information. The ROC curve is another commonly used metric, a chart where the TP rate and the FP rate are plotted for every classification threshold. The Area Under Curve (AUC) is often pointed to as a single metric for evaluation on the ROC curve, where 0.5 would be as good as a random classifier and 1.0 would be a perfect classifier.

In cases of highly imbalanced data, we can use the F1-score, which is the harmonic mean of the precision (shown in Eq. 4.2) and recall, simplified to Eq. 4.3:

$$Precision = \frac{TP}{TP + FP} \tag{4.2}$$

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} = \frac{TP}{TP + \frac{1}{2}(FP + FN)}. \tag{4.3}$$

We may also plot the graph of the Precision-Recall curve, which shows the precision and recall of the model in different classification thresholds. As with the ROC curve, we can take the area under the curve to simplify the curve into a single numeric metric.

4.2 MACHINE LEARNING MODELS

In this work, our experiments take a supervised learning approach to attempt to classify whether a given software is a malware. Supervised machine learning algorithms learn from labeled training datasets that provide examples of what samples in given classifications are like. Samples are represented as combinations of features extracted from instances of objects of classification, in our case, Mach-O binaries.

4.2.1 Naive Bayes

One of the most simple supervised machine learning algorithms is the Naive Bayes algorithm, which uses the Bayes conditional probability equation to classify samples based on how likely it is that a given feature belongs to a class (RUSSELL; NORVIG, 2002). With the *naive* condition that every feature is independent of each other, we get Eq. 4.4

$$\hat{y} = \underset{k \in \{1,\dots,K\}}{argmax}\ p(C_k) \prod_{i=1}^{n} p(x_i | C_k), \tag{4.4}$$

where $\hat{y}$ is the decision, K is the number of classes, n is the number of features and C_k is the kth class label.

4.2.2 Tree-based algorithms

Another simple algorithm is the decision tree, which classifies samples via a flowchart-like system where every node in the tree is a rule on a specific feature. The result of the comparison

dictates which node is the next to be tested until the algorithm reaches the leaves of the tree, which hold the final decision of which classification the sample fits in.

Decision trees are very prone to overfitting due to their own nature, that is, corresponding too closely to the training data and providing an unreliable generalization of the problem (RUSSELL; NORVIG, 2002). Random Forest is a way of mitigating this fact through the creation of an ensemble of different decision trees where a sample is tested against all the trees and the output is the result given by the majority of trees (HASTIE et al., 2009).

4.2.3 K-nearest neighbors

We also experiment with the k-nearest neighbors algorithm, a popular instance-based method that represents samples as vectors in a multidimensional space. The algorithm classifies new samples according to the k nearest samples, whichever class is predominant among these samples is the result of the classification (RUSSELL; NORVIG, 2002).

4.2.4 Logistic Regression

Logistic regression is a statistical analysis method used to predict an event based on prior observations of the dataset. In the logistic regression, we try to fit the logistic function to determine the probability of a feature value belonging to a specific class – in this paper, whether a feature set is more likely to belong to a malware or benign software (RUSSELL; NORVIG, 2002).

4.2.5 Support Vector Machine

SVM is a binary non-probabilistic linear classifier that receives multidimensional data as input and identifies patterns to separate different classes with hyperplanes, which work as decision boundaries that focus on maximizing the distance between the nearest instances in relation to each class (CORTES; VAPNIK, 1995).

4.2.6 Multilayer Perceptron

Perceptron is a learning model inspired by neurons. It is a neural network with only one layer and a linear classifier. When organized in a multilayer feedforward architecture, it is known as a Multilayer Perceptron (MLP).

The perceptrons in MLP are trained via the backpropagation learning algorithm, which uses samples from the data to tune their parameters according to the output. Its organization and learning allows for the classification of non-linear data, an upgrade from the single-layer perceptron (RUSSELL; NORVIG, 2002).

4.3 SAMPLING AND FEATURE REDUCTION

The generated dataset in Chapter 3 has two major issues that might generate problem in the classification results provided by the model: high dimensionality and highly imbalanced classes. In the following, we discuss ways to mitigate these problems in the dataset.

4.3.1 Undersampling and Oversampling

Imbalanced classes in the dataset negatively affect the training of the models. Two main approaches can be used to mitigate this: undersampling the majority class or oversampling the minority class. There are many different ways of going about these two approaches, however, for the sake of simplicity, in this work, we chose to use the random undersampling and SMOTE.

The random undersampling technique is simple like the name suggests, an undersampling tactic that chooses random samples from the majority class on a smaller scale than usual. The SMOTE algorithm generates synthetic samples based on the current samples in the minority class. It does so by selecting samples that are near the feature space, drawing lines between those, and choosing points in these lines to represent the synthetic data (CHAWLA et al., 2002). The SMOTE pseudo-code algorithm is described in Source Code 2.

Source Code 2 – SMOTE original algorithm

```
1 Algorithm SMOTE (T, N, k)
  Input: Number of minority class samples T ; Amount of SMOTE N percentage; Number
      of nearest
3 neighbors k
  Output: (N/100)* T synthetic minority class samples
```

```
 (* If N is less than 100%, randomize the minority class samples as only a random
percent of them will be SMOTEd. *)
if N < 100:
    Randomize the T minority class samples
    T = (N/100) * T
    N = 100

N = (int)(N/100) # The amount of SMOTE is assumed to be in integral multiples of
k = Number of nearest neighbors
numattrs = Number of attributes
Sample[ ][ ]: array for original minority class samples
newindex: keeps a count of number of synthetic samples generated, initialized to
    0
Synthetic[ ][ ]: array for synthetic samples

#Compute k nearest neighbors for each minority class sample only.
for i in 1 to T
    Compute k nearest neighbors for i, and save the indices in the nnarray
    Populate(N , i, nnarray)
endfor

Populate(N, i, nnarray) # Function to generate the synthetic samples.
    while not N == 0:
        Choose a random number between 1 and k, call it nn. This step chooses one
             of the k nearest neighbors of i.
        for attr in 1 to numattrs:
            Compute: dif = Sample[nnarray[nn]][attr] - Sample[i][attr]
            Compute: gap = random number between 0 and 1
            Synthetic[newindex][attr] = Sample[i][attr] + gap * dif
        newindex++
        N = N - 1
    return # End of Populate.
```

Source: Adapted from CHAWLA et al. (2002)

4.3.2 Feature selection

When dealing with high dimensionality, one either uses feature selection or feature projection techniques. In feature selection, we drop features from the dataset that may not be as helpful in the classification as others. In feature projection, one uses algorithms to transform the values of the current dataset into a representation of lower dimension (PUDIL; NOVOVIČOVÁ, 1998).

Three common feature selection techniques are through Analysis of Variance (ANOVA)

with F-test, the Chi-Squared test, and the information gain. F-test is a statistical test that checks whether a given feature variance impacts different classifications (LOMAX; HAHS-VAUGHN, 2013). The F value is calculated as shown in Eq. 4.5:

$$F\ value = \frac{\text{variance between classes}}{\text{variance within class}}. \tag{4.5}$$

The Chi-squared test checks the independence between two different events (COCHRAN, 1952). With this test, we can measure which features have more dependence on the classification (malware or benign), allowing us to select features with higher dependence for our model and discard less dependent ones. The Chi-squared value is calculated from Eq. 4.6, where we divide all observations between n cells that represent all possible configurations between two variables to test their independence. In the equation O is the count of observed values and E is the count of expected values:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i} \tag{4.6}$$

Another way of discarding less useful features is through the information gain. The information gain is a way to measure how much entropy is lost when a given feature is known (KULLBACK; LEIBLER, 1951). A feature with high information gain suggests that it holds more information for the classification than others.

4.3.3 Principal Component Analysis

Principal Component Analysis (PCA) is a tool for exploratory data analysis. It can be used to reveal the internal structure of data by analyzing data patterns in datasets of high dimensionality. As an unsupervised feature reduction algorithm, it finds the directions of maximum variance in high dimensional data and maps them into a space with fewer dimensions (WOLD; ESBENSEN; GELADI, 1987).

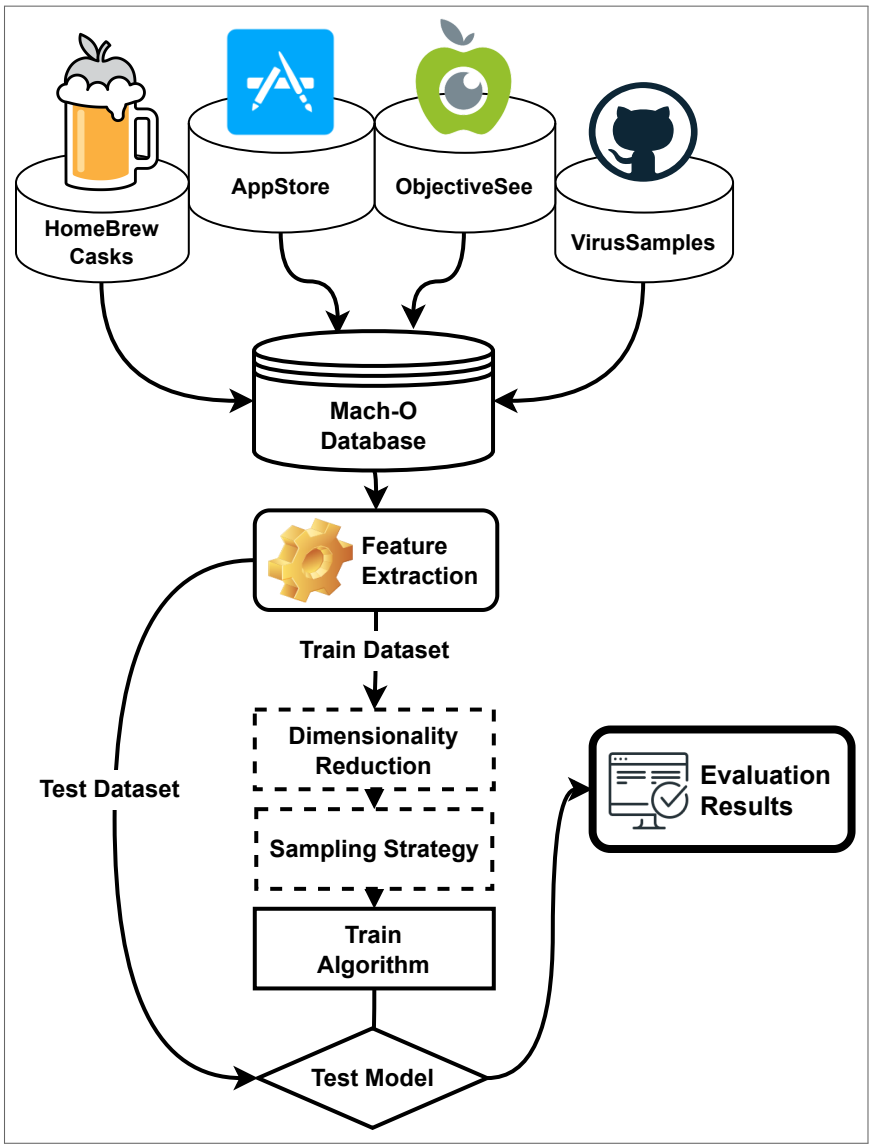
4.4 EXPERIMENT METHODOLOGY

Our methodology is best illustrated by Figure 5. Our first step was to build a dataset with benign and malicious Mach-O files from four different public sources. With the down-

loaded files, we extract interesting features using LIEF to generate the dataset used for the experiments.

After the feature extraction, we use a stratified 5-fold cross-validation technique dividing the data into a train and test dataset. We then apply a dimensionality reduction algorithm to the training dataset, followed by a sampling strategy before feeding the data into one of the machine learning models. Once the model is trained, we use the test dataset to retrieve the performance metrics for the detection model. We repeat the stratified cross-validation ten times to avoid noise and retrieve more reliable metrics.

Figure 5 – Diagram of the methodology used.



Source: The author (2022)

5 EVALUATION

In this chapter we go into details on the experiment's execution. We explain the pre-processing stage and evaluate the impacts of different sampling strategies and feature reduction techniques on the detection models. We present and discuss results when combining the techniques into fully working macOS malware detection mechanisms.

5.1 PRE-PROCESSING

In the pre-processing stage, we evaluate how we are going to feed the features into the model. The raw dataset built using the LIEF tool holds plenty of different meta-information on the binaries, however, some of this information might not be as useful or as easy to work with, such as addresses and offsets. This information can be difficult to represent categorically or numerically, and in general, neither are a good indicator of malicious activity.

The dataset has some missing values due to the fact that some sections and load commands are not present in every binary, therefore, no features can be extracted from these parts on some binaries. Firstly, we completely removed the features with zero variance and those missing 20 percent or more of the available binaries in the dataset.

Some features are represented by string values. In such cases, we used dummy encoding to transform these features into binary-valued columns. Most features were encoded this way, such as load command names, segment and section names, section flags, Mach-O header flags, and imported libraries. After encoding all categorical features, we got a dataset of 10,772 samples with 3,260 features.

5.2 INITIAL EVALUATION

Firstly, we evaluate the performance of the algorithms using 5-fold cross-validation repeated 10 times directly on our dataset. We normalize the input of numerical columns and performed the classification with Decision Tree, Random Forest, SVM, Logistic Regression, K-nearest neighbors, and MLP. Our implementation uses the popular machine learning library sci-kit-learn in Python 3 scripts run in a Windows 10 64-bit system, with 16 GB of RAM and a Ryzen 7 1700 processor. In Table 3, we can see the used parameters for each algorithm implementation.

The parameters were tuned manually from default values to provide better results while not being too time-consuming.

Table 3 – Machine Learning parameters

Algorithm	Parameters
Decision Tree	Gini impurity, no max_depth, default parameters
Naive Bayes	GaussianNB
Random Forest	n_estimators=150, max_depth=25, random_state=0
K-Neareast Neighbors	n_neighbors=3
Logistic Regression	max_iter=2000,
SVM	max_iter=20000
Multi Layer Perceptron	hidden_layer_sizes=(100,), random_state =1, max_iter=1000

Source: The author (2022)

Table 4 – Average scores of 5-Fold repeated 10 times cross validation of different algorithms with an unchanged dataset

Algorithm	Accuracy	ROC AUC	PR AUC	F1
Decision Tree	0.9804	0.9133	0.7058	0.8339
Naive Bayes	0.6171	0.7809	0.1270	0.2282
Random Forest	0.9858	0.9974	0.9685	0.**8652**
K-Neareast Neighbors	0.9831	0.9541	0.8514	0.8484
Logistic Regression	0.9829	0.9920	0.9291	0.8468
Support Vector Machine	0.9850	0.9893	0.9245	0.**8728**
Multi Layer Perceptron	0.9868	0.9922	0.9351	0.**8869**

Source: The author (2022)

In Table 4, we show the results obtained using the metrics F1-score, accuracy, area under the ROC curve, and the area under the Precision-Recall curve. We can see how accuracy as a metric may lead to a false sense of good performance as almost all algorithms exhibited an accuracy over 98%. This is mainly due to the high-class imbalance in the dataset. In this initial evaluation with the raw dataset, the MLP, SVM, and Random Forest classifiers appear to have the best performance when looking at the F1 score.

It's important to note the subpar performance of the Naive Bayes. Its simple algorithm assumes independence among all features which is an unreasonable expectation in this high dimensionality dataset.

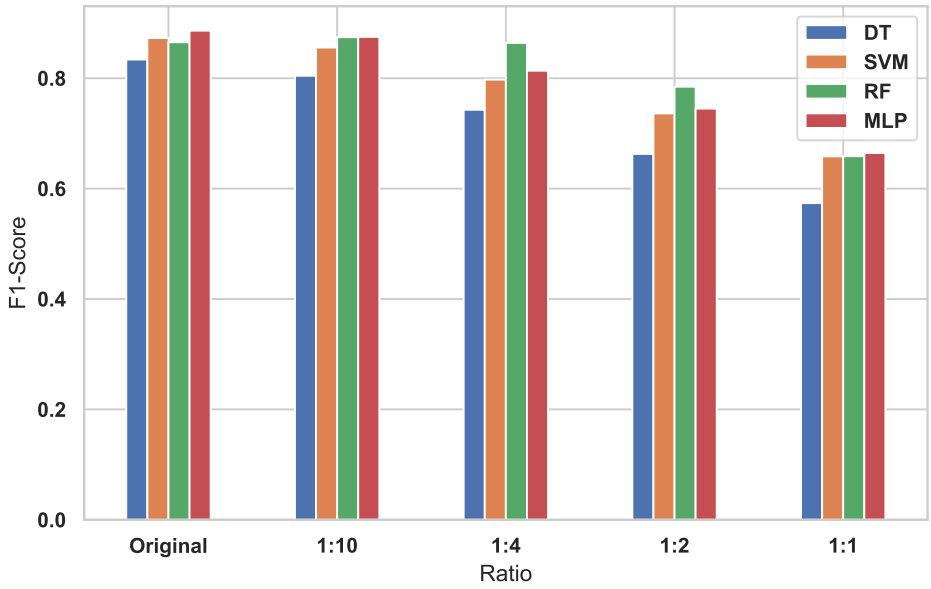
5.3 MITIGATING ISSUES WITH DATASET

Given the highly imbalanced classes and the high dimensionality of the problem caused by a large number of features in contrast to the small number of malicious samples, we must adjust the model accordingly to mitigate the problems caused by those undesired characteristics of our dataset.

5.3.1 Sampling strategy

We chose to tackle the imbalance of classes by evaluating the performance with two different sampling strategies to address the low malicious to benign sample ratio in the dataset. The original dataset has a malware to benign software ratio of approximately 1 to 16. We altered this ratio with random undersampling from the majority class and generating synthetic malware observations with SMOTE.

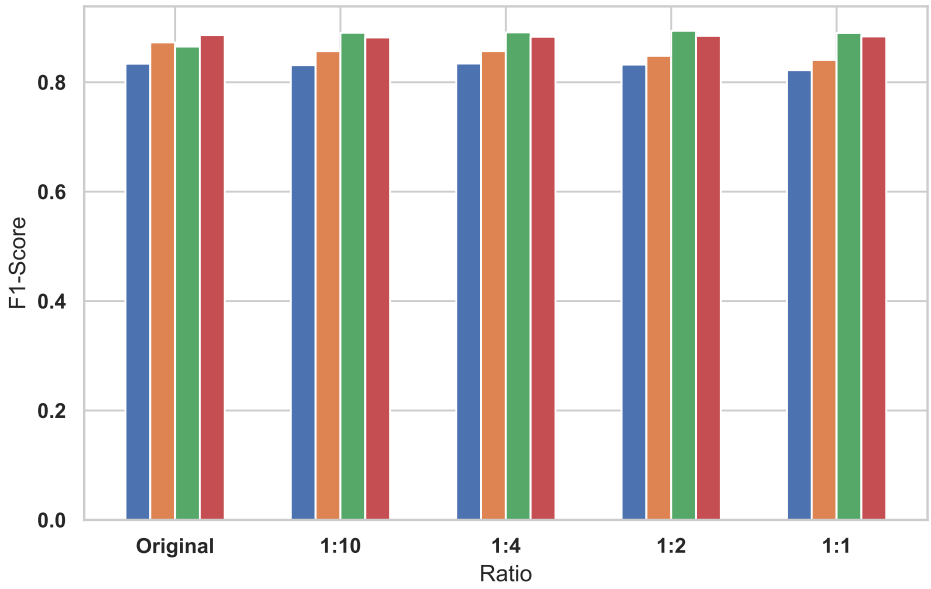
Figure 6 – F1-score of decision tree, random forest, SVM and MLP with the undersampling of the majority class to achieve a specific malware to benign ratio.



Source: The author (2022)

Fig. 6 shows the effect of undersampling the benign software observations on the F1 score. As seen in Figure 6, by changing the malware ratio to 1:10 via random undersampling, the F1-score exhibits little change; in fact, it provides a slight improvement in the Random Forest

Figure 7 – F1-score of decision tree, random forest, SVM and MLP with the oversampling of the minority class to achieve a specific malware to benign ratio.



Source: The author (2022)

algorithm. However, further undersampling appears to decrease the F1 score of every algorithm. As shown in Fig. 7, oversampling using SMOTE appears to not result in many impacts, with minor changes as the ratio increases.

The F1 score has shown to be more sensitive to undersampling tactics as the malicious to benign ratio increases. However, the SMOTE technique has been demonstrated to be more promising since it maintained a relatively constant F1 score regardless of the increasing ratio.

5.3.2 Dimension reduction

In order to deal with the high dimensionality issue, we tried out the dataset with feature selection techniques like Chi-squared, analysis of variation, and information gain, to reduce the number of features. We also used PCA to transform the dataset into a dataset of lower dimension.

Table 5 shows the F1-score for a decreasing number of features using the different dimension reduction techniques. As observed, feature reduction seems to improve the models in comparison to our initial evaluation. As seen in the table, PCA appears to generate the best results with MLP, demonstrated by the F1-score kept relatively stable throughout the

Table 5 – F1-Score of Random Forest, SVM and Multi-Layer perceptron with reduced number of features.

	1000	500	250	100	50	25
ANOVA						
RF	0.8749	0.8811	0.8846	**0.8855**	0.8725	0.8586
SVM	0.8723	0.8664	0.8509	0.7964	0.7592	0.6931
MLP	0.9001	0.9059	0.8957	0.8853	0.8485	0.7752
CHI-2						
RF	0.8760	0.8834	0.8814	**0.8833**	0.8173	0.5237
SVM	0.8713	0.8622	0.8252	0.7845	0.6895	0.3955
MLP	0.8998	0.9040	0.8995	0.8786	0.7660	0.5255
Information Gain						
RF	0.8691	0.8706	0.8724	0.8752	0.8533	0.8055
SVM	0.8515	0.8374	0.8108	0.7850	0.6790	0.4184
MLP	0.8926	0.8902	0.8852	0.8722	0.7927	0.5825
Principal Component Analysis						
RF	0.7745	0.7840	0.8016	0.8174	0.8262	0.8042
SVM	0.8713	0.8642	0.8436	0.7856	0.7518	0.4895
MLP	**0.9017**	**0.9057**	**0.9008**	**0.9039**	**0.8911**	0.8614

Source: The author (2022)

dimension reducing tests down to only 50 features. The Random Forest models appear to improve with Chi-squared and ANOVA with only 100 features. The performance shown by the SVM model did not improve with any of the tested dimension reduction techniques. The information gain feature reduction method also underperformed in comparison to the other techniques.

5.4 RESULTS AND DISCUSSION

After diving into the strategies to mitigate the dataset issues, we can combine them into experiment models. Table 6 shows the models with the best average performances using a repeated stratified 5-fold cross-validation technique.

As seen in Table 6, by evaluating the F1 score the best performances were MLP and Random Forest. Different configurations of sampling and feature reduction techniques provided us with a myriad of statistics of similar performance. While the detection rate was not extremely high, the false alarm rate was extremely low. Both undersampling and oversampling have been shown to be useful tactics, but we must be wary of extreme undersampling. In Table 6, we

Table 6 – Best performance models and their detection rate, false alarm rate and F1-score

Models	DR	FAR	F1
MLP + PCA 500 + SMOTE 1:4	90.6%	0.5%	0.9068
MLP + PCA 250 + SMOTE 1:4	90.1%	0.5%	0.9023
MLP + PCA100 + SMOTE 1:10	89.4%	0.5%	0.9004
MLP + PCA100 + Undersamp. 1:10	90.6%	0.7%	0.8944
MLP + ANOVA 250 + SMOTE 1:10	88.6%	0.5%	0.8973
MLP + CHI2 500 + SMOTE 1:10	89.2%	0.4%	0.9049
RF + ANOVA 250 + SMOTE 1:10	83.6%	0.2%	0.8905
RF + CHI2 250 + Undersamp. 1:10	85.7%	0.4%	0.8893
MLP + PCA100 + Undersamp. 1:2	95.0%	2.9%	0.7827

Source: The author (2022)

can see a scenario of extreme undersampling increasing the perceived detection rate while maintaining a false alarm ratio at acceptable levels, but as indicated by the F1-score, such a result is not reliable because of the low number of malware in the dataset.

A fair direct comparison with other detection approaches is difficult since the used dataset was not the same. In Table 7, we revisit the limitation of recent works on the subject. In this work, we created a large dataset for malware detection, and applied sampling and feature reduction techniques, while addressing the class imbalance of the dataset throughout the experiment to achieve a low false alarm rate and over 90% detection rate.

Table 7 – MacOS malware research limitations

Research	Limitation
(MIEGHEM, 2016)	21 malwares only, high FAR
(PAJOUH et al., 2018)	SMOTE before train-test split
(SAHOO; DHAWAN, 2022)	SMOTE before train-test split
(GHARGHASHEH; HADAYEGHPARAST, 2022)	Imbalance of class in dataset not addressed

Source: The author (2022)

5.4.1 MacOS embedded detection

As previously mentioned, *XProtect* is a simple signature-based detection system that recently became a component of the now larger and more powerful *Gatekeeper* on macOS. As of macOS 12, there are no other native malware detection capabilities.

Most signatures either try to match the hash of the malware file to a specifically known hash or verify if the file has a specific byte string inside it. As of September of 2021, the XProtect signature file has only 160 rules, of which managed to match only 58 malwares out of the 631 in our database, exhibiting a less than 10% detection rate with no false alarms.

While having no false alarms is great in a malware detection system, that fact is generally a common feature among signature-based detection mechanisms with well-written rules. The low detection rate, however, is a serious concern considering there is no other malware detection mechanism in the macOS. In signature-based detection systems, one may increase its detection rate by adding more rules regularly as new malware appears, but that seems not to be the case of *XProtect* as its database holds a very small number of signatures.

6 CONCLUSION

This work addresses the malware detection challenge in the macOS environment. We built a database from samples of malware and benign software publicly available on the internet. We then extracted static features from the Mach-O file, resulting in the largest public dataset for malware detection in macOS to the best of our knowledge[1]. We trained different machine learning models, investigated the impact of sampling tactics and dimension reduction techniques, and in the end achieved a detection model that had its performance compared to other approaches on Windows and Android platforms. The models are also shown to be better than current macOS detection mechanisms, pointing to the fact that Apple may greatly benefit from minor machine learning approaches to endpoint security.

Practically speaking, the shown detection mechanisms can be integrated into either the macOS operating system or into app distribution chains. Thanks to the simplicity of extracting static features from the binary, the machine learning models can be easily embedded into the working Gatekeeper system as an extra check before the software's first execution. And even though the AppStore's app admission process and the notarization process are currently unknown to the public, it is not unreasonable to assert that these detection models could become a part of such processes.

6.1 FUTURE WORK

We investigated and experimented with the use of static features present in the Mach-O format to tackle the malware detection problem in a macOS environment, which opens plenty of new research possibilities. Future works will explore other ignored static features in the code segment to identify malicious patterns in the assembly code. Detection research should also investigate dynamic features. The *EndpointSecurity* framework available since macOS Catalina gives us new opportunities to detect malicious behavior during execution by gathering data from system events originating from processes. The main challenge for such work would be the virtualization of macOS/iOS environments to create experiment testbeds to retrieve dynamic features in large scale.

Regardless of the type of endpoint security technique investigated, further work should consider expanding the dataset to include more malware samples for a better representation

[1] Available at https://github.com/CBurgardt/orange_vs_apples

of the current macOS malware landscape.

REFERENCES

ALVAREZ, V. M. *YARA in a nutshell*. 2021. Accessed: 2021-11-09. Available at: <https://virustotal.github.io/yara/>.

APP Store. 2021. <https://www.apple.com/br/app-store/>. Accessed: 2021-11-20.

APPLE. *Code Signing Tasks*. 2016. Accessed: 2021-11-09. Available at: <https://developer.apple.com/news/?id=12232019a>.

APPLE. *macOS Code Signing In Depth - Documentation*. 2016. <https://developer.apple.com/library/archive/technotes/tn2206/>. Accessed: 2021-11-09.

APPLE. *About System Integrity Protection on your Mac*. 2019. Accessed: 2021-11-09. Available at: <https://support.apple.com/en-us/HT204899>.

APPLE. *Update to Notarization Prerequisites*. 2019. Accessed: 2021-11-09. Available at: <https://developer.apple.com/news/?id=12232019a>.

APPLE. *App Store Review Guidelines*. 2021. Accessed: 2021-11-09. Available at: <https://developer.apple.com/app-store/review/guidelines/>.

APPLE. *Gatekeeper and runtime protection in macOS*. 2021. Accessed: 2021-11-09. Available at: <https://support.apple.com/guide/security/gatekeeper-and-runtime-protection-sec5599b66df/web>.

APPLE. *Notarizing macOS Software Before Distribution*. 2021. <https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution>. Accessed: 2021-11-05.

APPLE. *Notarizing macOS Software Before Distribution*. 2021. Accessed: 2021-11-09. Available at: <https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution/>.

APPLE. *Protecting against malware in macos*. 2021. Accessed: 2021-11-09. Available at: <https://support.apple.com/guide/security/protecting-against-malware-sec469d47bd8/web>.

APPLE. *App Sandbox*. 2022. Accessed: 2021-11-09. Available at: <https://developer.apple.com/documentation/security/app_sandbox>.

APPLE. *Disabling and Enabling System Integrity Protection*. 2022. Accessed: 2021-11-09. Available at: <https://developer.apple.com/documentation/security/disabling_and_enabling_system_integrity_protection>.

BALDANGOMBO, U.; JAMBALJAV, N.; HORNG, S.-J. A static malware detection system using data mining methods. *arXiv preprint arXiv:1308.2831*, 2013.

CASE, A.; III, G. G. R. Advancing macOS X rootkit detection. *Digital Investigation*, Elsevier, v. 14, p. S25–S33, 2015.

CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, v. 16, p. 321–357, 2002.

CLULEY, G. *History of Mac malware: 1982 – 2011*. 2011. <https://nakedsecurity.sophos.com/2011/10/03/mac-malware-history/>. Accessed: 2021-11-09.

CLULEY, G. *Macs and malware – See how Apple has changed its marketing message*. 2012. <https://nakedsecurity.sophos.com/2012/06/14/mac-malware-apple-marketing-message/>. Accessed: 2021-11-09.

COCHRAN, W. G. The $\chi 2$ test of goodness of fit. *The Annals of mathematical statistics*, JSTOR, p. 315–345, 1952.

CORTES, C.; VAPNIK, V. Support-vector networks. *Machine learning*, Springer, v. 20, n. 3, p. 273–297, 1995.

G., E. *macOS malware development surged by over 1,000% in 2020*. 2021. <https://atlasvpn.com/blog/macos-malware-development-surged-by-over-1-000-in-2020>. Accessed: 2021-11-09.

GANDOTRA, E.; BANSAL, D.; SOFAT, S. Zero-day malware detection. In: IEEE. *2016 Sixth international symposium on embedded computing and system design (ISED)*. [S.l.], 2016. p. 171–175.

GHARGHASHEH, S. E.; HADAYEGHPARAST, S. Mac os x malware detection with supervised machine learning algorithms. In: *Handbook of Big Data Analytics and Forensics*. [S.l.]: Springer, 2022. p. 193–208.

HASTIE, T.; TIBSHIRANI, R.; FRIEDMAN, J. H.; FRIEDMAN, J. H. *The elements of statistical learning: data mining, inference, and prediction*. [S.l.]: Springer, 2009.

HOMEBREW. 2021. <https://brew.sh/>. Accessed: 2021-11-20.

HOMEBREW. *Homebrew Analytics Data*. 2021. Accessed: 2021-04-04. Available at: <https://formulae.brew.sh/analytics>.

HSIEH, S.; LIU, H. Automatic classifying of macOS X samples. In: . [S.l.: s.n.], 2017.

IFERT-MILLER, F. *macOS Catalina osquery*. 2019. Accessed: 2021-11-09. Available at: <https://blog.kolide.com/macos-catalina-osquery-a6753dc3c35c>.

KOLTER, J. Z.; MALOOF, M. A. Learning to detect malicious executables in the wild. In: *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. [S.l.: s.n.], 2004. p. 470–478.

KULLBACK, S.; LEIBLER, R. A. On information and sufficiency. *The annals of mathematical statistics*, JSTOR, v. 22, n. 1, p. 79–86, 1951.

LO, C. T. D.; PABLO, O.; CARLOS, C. M. Towards an effective and efficient malware detection system. In: IEEE. *2016 IEEE International Conference on Big Data (Big Data)*. [S.l.], 2016. p. 3648–3655.

LOMAX, R. G.; HAHS-VAUGHN, D. L. *Statistical Concepts-A Second Course*. [S.l.]: Routledge, 2013.

MAC Malware Collection. 2021. Accessed: 2021-06-04. Available at: <https://objective-see.com/malware.html>.

MACH-O Format Overview. 2014. <https://developer.apple.com/library/archive/documentation/Performance/Conceptual/CodeFootprint/Articles/MachOOverview.html>. Accessed: 2021-11-09.

MALWAREBYTES. *2020 State of Malware Report*. 2020. <https://www.malwarebytes.com/resources/files/2020/02/2020_state-of-malware-report.pdf>. Accessed: 2021-11-09.

MALWARESAMPLES. *MacOS Malware Samples*. 2021. <https://github.com/MalwareSamples/Macos-Malware-Samples>. Accessed: 2021-06-04.

MANAVI, F.; HAMZEH, A. Static detection of ransomware using LSTM network and PE header. In: *2021 26th International Computer Conference, Computer Society of Iran (CSICC)*. [S.l.: s.n.], 2021. p. 1–5.

MCAFFEE. *McAfee Labs Threats Report*. 2017. Https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-mar-2017.pdf.

MICROSOFT. *PE Format - Microsoft Docs*. 2021. <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format>. Accessed: 2021-11-09.

MIEGHEM, V. *Detecting malicious behaviour using system calls*. Phd Thesis (PhD Thesis) — Master thesis, TU Delft, 2016.

MILOSEVIC, N.; DEHGHANTANHA, A.; CHOO, K.-K. R. Machine learning aided Android malware classification. *Computers & Electrical Engineering*, Elsevier, v. 61, p. 266–274, 2017.

OAKLEY, H. *MRT: what do we know about it?* . 2020. Accessed: 2021-11-09. Available at: <https://eclecticlight.co/2020/10/23/mrt-what-do-we-know-about-it/>.

ORANGES vs Apples. 2021. <https://github.com/CBurgardt/orange_vs_apples>. Accessed: 2022-01-01.

PAJOUH, H. H.; DEHGHANTANHA, A.; KHAYAMI, R.; CHOO, K.-K. R. Intelligent OS x malware threat detection with code inspection. *Journal of Computer Virology and Hacking Techniques*, Springer, v. 14, n. 3, p. 213–223, 2018.

PHAM, D.-P.; VU, D.-L.; MASSACCI, F. Mac-a-mal: macOS malware analysis framework resistant to anti evasion techniques. *Journal of Computer Virology and Hacking Techniques*, Springer, v. 15, n. 4, p. 249–257, 2019.

PINHEIRO, A. J.; BEZERRA, J. d. M.; BURGARDT, C. A.; CAMPELO, D. R. Identifying iot devices and events based on packet length from encrypted traffic. *Computer Communications*, Elsevier, v. 144, p. 8–17, 2019.

PUDIL, P.; NOVOVIČOVÁ, J. Novel methods for feature subset selection with respect to problem knowledge. In: *Feature extraction, construction and selection*. [S.l.]: Springer, 1998. p. 101–116.

ROMAIN, T. *LIEF: Library to Instrument Executable Formats*. 2021. <https://lief-project.github.io/>. Accessed: 2021-04-05.

RUSSELL, S.; NORVIG, P. Artificial intelligence: a modern approach. 2002.

SAHOO, D.; DHAWAN, Y. Evaluation of supervised and unsupervised machine learning classifiers for mac os malware detection. In: *Handbook of Big Data Analytics and Forensics*. [S.l.]: Springer, 2022. p. 159–175.

SCHULTZ, M. G.; ESKIN, E.; ZADOK, F.; STOLFO, S. J. Data mining methods for detection of new malicious executables. In: IEEE. *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. [S.l.], 2000. p. 38–49.

SIKORSKI, M.; HONIG, A. *Practical malware analysis: the hands-on guide to dissecting malicious software*. [S.l.]: no starch press, 2012.

STOKES, P. *macOS Security Updates Part 3*. 2020. Accessed: 2021-11-09. Available at: <https://www.sentinelone.com/blog/macos-security-updates-part-3-apples-whitelists-blacklists-and-yara-rules/>.

STOKES, P. *"EvilQuest" Rolls Ransomware, Spyware Data Theft Into One*. 2020. <https://www.sentinelone.com/blog/evilquest-a-new-macos-malware-rolls-ransomware-spyware-and-data-theft-into-one/>. Accessed: 2021-11-09.

STOKES, P. *Top 10 macOS Malware Discoveries in 2021 | A Guide To Prevention Detection*. 2021. <https://www.sentinelone.com/blog/top-10-macos-malware-discoveries-in-2021-a-guide-to-prevention-detection/>. Accessed: 2021-11-09.

VIRUSTOTAL. *VirusTotal*. 2021. Accessed: 2022-01-05. Available at: <https://www.virustotal.com>.

WOLD, S.; ESBENSEN, K.; GELADI, P. Principal component analysis. *Chemometrics and intelligent laboratory systems*, Elsevier, v. 2, n. 1-3, p. 37–52, 1987.