The Role of Domain-Specific Features in Malware Detection: A macOS Case Study

Biagio Montaruli
montarul@eurecom.fr
EURECOM & SAP
France

Andrea Oliveri
oliveri@eurecom.fr
EURECOM
France

Savino Dambra
savino.dambra@gendigital.com
GenDigital
France

Davide Balzarotti
balzarot@eurecom.fr
EURECOM
France

Abstract

Despite the growing popularity of macOS among end users and enterprise systems, malware research has primarily focused on Windows and Android operating systems, leaving the problem of macOS malware detection relatively unexplored. Indeed, the specificity of the operating system and the unique characteristics of the Mach-O file format can play a fundamental role in the classification of unknown samples, drastically increasing the detection rate.

In this work, for the first time in the literature, we employ new domain-specific features, i.e., static features specific to macOS binaries, such as embedded certificates, entitlements, persistence techniques and key system APIs, to train a machine learning malware detector. We perform a comprehensive experimental evaluation on a novel dataset of 41,129 samples, comprising 11,413 benign and 29,716 malicious executables, and demonstrate that our solution achieves state-of-the-art detection performance (98.50%), outperforming all existing approaches, with an average improvement of 16% in terms of detection rate. We also provide an in-depth analysis of the importance of the individual features, showing that our detector effectively leverages the new domain-specific features. Then, in order to evaluate the generalization capabilities of our detector over time, we perform a real-world evaluation on a new dataset of 9,000 fresh macOS executables. The results show that (i) our detector maintains a very high detection rate (99.50%), (ii) outperforms the state-of-the-art by 50%, and (iii) the domain-specific features are crucial for generalizing to novel malware samples, as their removal leads to a 15.92% drop in detection performance.

Finally, in the spirit of open science, we release our dataset to the research community.

CCS Concepts

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies → Machine learning**.

Keywords

machine learning, macos, malware

ACM Reference Format:
Biagio Montaruli, Andrea Oliveri, Savino Dambra, and Davide Balzarotti. 2026. The Role of Domain-Specific Features in Malware Detection: A macOS Case Study. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '26), June 1–5, 2026, Bangalore, India.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779208.3785392

This work is licensed under a Creative Commons Attribution 4.0 International License.
ASIA CCS '26, Bangalore, India
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2356-8/26/06
https://doi.org/10.1145/3779208.3785392

1 Introduction

In recent years, the adoption of macOS among enterprises has significantly increased, and it is now part of 76% of large US businesses. If this trend continues, macOS is expected to become the dominant operating system for enterprise endpoints by 2030 [22, 62]. Along with its growing popularity, macOS has faced many security challenges in recent years. As reported by Kaspersky [28], in 2023, a total of 17 zero-day vulnerabilities targeting macOS were discovered, including over a dozen classified as high-risk and one as critical. The threat landscape is further exacerbated by the growing number of malware samples designed for macOS. Indeed, as reported by Moonlock Lab [40], 2024 has been characterized by a noteworthy increase in macOS malware, with a significant growth in the variety and sophistication of infostealers. In addition, malware authors are starting to target the `arm64` architecture of new Macs, either by building `arm64` binaries or by using `fat` file format that supports multiple architectures [52].

The increasing maturity of the malware ecosystem has prompted researchers to port existing malware detection techniques, mainly based on machine learning, to the macOS operating system. This provides a unique opportunity to observe the evolution of these techniques and to study how the specificity of the OS affects the features and accuracy of detectors. In fact, while general techniques remain the same across operating systems, the role they play in the detection of malicious samples may vary from one system to another. For instance, the features commonly adopted to detect malicious samples in Android [10] differ significantly from those used for detecting Windows malware [20].

At the time of writing, existing approaches proposed in the literature still rely solely on generic static features, such as strings, byte N-grams, and the number of sections [17, 24, 42, 48, 53]. However, the unique characteristics of the Mach-O binary format, the container for executables and libraries on macOS, and its built-in security mechanisms (such as embedded code-signing certificates and entitlements) have been largely overlooked. Although analogous features exist on other platforms, for example, certificates in the Windows Portable Executable (PE) [29] and permissions in Android [10], no prior work in macOS malware detection has systematically investigated how these macOS-specific features can be leveraged to identify malicious software and enhance the detection performance. Additionally, it is important to understand whether the use of these features can provide additional benefits, such as enhanced generalizability or robustness to data drift and the introduction of new variants.

Contributions – In this work, we take advantage of the emerging and rapidly evolving landscape of malware defenses for macOS to investigate the role and impact of macOS-specific features on malware detection. In particular, we designed a number of experiments to show how unique features of the Mach-O file format can enhance the performance of a machine learning-based detector.

We also observed that the absence of a large, up-to-date, publicly available dataset of benign and malicious macOS samples has significantly limited the development of machine learning-based solutions. Therefore, as a key contribution of this work, we collected a new dataset containing 41,129 samples (11,413 benign and 29,716 malicious), spanning three architectures: `x86-64`, `arm64`, and `fat`. This represents the largest dataset of macOS binaries used in the literature for malware detection. Indeed, in comparison, the most commonly used public dataset in this area contained only 152 binaries [42]. To support future experiments and in the spirit of open science, we publicly release the dataset [39].

The results of our experiments show that our detector, trained on macOS-specific static features, significantly outperforms existing solutions, with an improvement of over 16% over the state-of-the-art solutions [17, 24, 42, 48, 53]. Previous research works [25, 44] have also investigated deep learning methods such as MalConv [45] that automatically extract significant patterns from raw byte sequences rather than relying on manual feature engineering. However, our experiments show that MalConv is not as effective as feature-based approaches in the context of macOS malware detection. This interesting finding highlights the importance of leveraging semantically-rich features in macOS malware detection, as they can provide more meaningful insights into the characteristics of malicious samples compared to raw byte sequences.

An in-depth analysis of the feature importance reveals two interesting results. On the one hand, the distinctive features of macOS binaries are consistently ranked as the most important for the classification task. On the other hand, when these features are removed, the detector is still able to achieve comparable results by relying on other, generic features. This seems to suggest that while these specific features provide a more direct way to classify samples compared with the less semantically-rich headers information or byte N-grams, the latter may still be sufficient to detect *known* samples.

To evaluate our detector on *fresh* data, i.e., temporally newer samples, we conducted a real-world assessment using 9,000 macOS binaries collected from the VirusTotal feed over a three-month period (September to November 2024). A key finding from this second set of experiments is that macOS-specific features generalize much better to new malware samples. In fact, while generic features were still sufficient to reliably detect known samples, performance on new data dropped by over 15% when specific features were removed, hence demonstrating their key generalization role.

Finally, in the real-world experiment, our detector based on the full feature set continued to achieve a high detection rate, i.e., 99.50% at a 1% false positive rate (FPR), even on new samples, and continued to outperform state-of-the-art detectors, with a remarkable 50.03% improvement in detection rate at 1% FPR.

2 Background

Each operating system is defined by key characteristics and unique file formats that reflect its underlying architecture and design. By leveraging this platform-specific information, we can gain critical insights into the structure and attributes of executables, enabling a more precise distinction between benign and malicious samples. For example, researchers have leveraged the *Portable Executable (PE) Rich Header* to detect Windows malware [63], anomalies in the *ELF* file format to flag binaries designed for Linux platforms [19], and the sequence of requested permissions as a sign of suspicious behavior for Android [10]. In this paper, we focus on macOS malware, an emerging field in which researchers have so far only relied on generic features [17, 24, 42, 48, 53], such as the number of sections and strings, but have not investigated any macOS-specific features tied to its unique file formats and architectures.

2.1 macOS Applications and Mach-O Binaries

macOS applications (apps) are packaged as *app bundles* [6, 42], composed of a directory tree containing various files and sub-directories. Notably, each bundle contains the main executable of the app in the Mach-O format, an `Info.plist` file with metadata (e.g., name, version, supported platforms by the program), a set of directories dedicated to `assets` (e.g., images, configurations), required `Frameworks` (e.g., dynamic libraries used by the app), and optional `Plugins`.

In this work, we focus on Mach-O executables rather than entire app bundles to extract macOS-specific indicators and build our classifiers. This choice is motivated by recent research [17, 24, 42, 48, 53], which highlights Mach-O executables as the most significant artifacts for macOS malware detection. Additionally, individual binaries are the primary granularity at which analysts commonly share malicious macOS software, emphasizing the need to design features and classifiers that specifically target executables rather than app bundles. In support of this approach, we conducted a preliminary analysis of the VirusTotal (VT) [58] feed over a five-month period (July–November 2023) and found that only 6% of malware Mach-O executables included metadata linking them to their parent app bundles.

> It is crucial to highlight that having access only to the Mach-O file of an app bundle, a very common situation for samples shared on VirusTotal, makes dynamic analysis nearly impossible and static analysis significantly harder. While malicious PE and ELF executables are typically self-contained, Mach-O binaries often depend on external files within the app's bundle, a unique characteristic that introduces a significant challenge for the analyst. The absence of these files (e.g., dynamic libraries) can prevent the sample from executing entirely or limit the analyst's ability to fully assess its capabilities, even during static analysis. For instance, if the Mach-O relies on external scripts to perform system operations, the analyst may struggle to fully understand its behavior.

As mentioned, the main executable of a macOS app is stored in a Mach-O file [49, 61]. A Mach-O file is a container that can represent not only executables but also object code, shared libraries, and core dumps. It consists of three main parts (see Figure 1): header, load

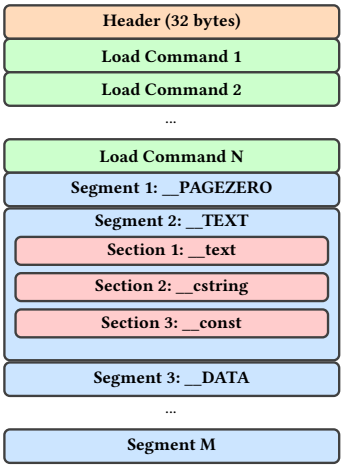

Figure 1: Mach-O file format.

commands, and data. Every Mach-O file starts with a header that includes a magic number and a set of additional information that instructs the loader on how to interpret the remaining part of the file, such as the CPU architecture for which the binary has been compiled. After the header, each Mach-O file includes a series of load commands that describe its internal organization and instruct the loader on how to map the file contents into memory. The file is organized into segments, which represent logical regions of code or data required by the app. Each segment is further divided into sections, which contain specific types of code or data, such as executable instructions, read-only constants, or writable variables. These sections provide fine-grained organization within the broader structure of segments, allowing the operating system to handle and protect different types of data appropriately.

Comparison with Windows Executables.

The Mach-O file format differs from the Windows PE format [36] in several aspects. Mach-O binaries contain a single Mach header, while PE files begin with an MS-DOS stub, followed by the PE header, which includes a COFF (Common Object File Format) header and an optional header. Moreover, unlike the PE format in which data and code are organized into sections, the Mach-O format adopts a more complex structure consisting of segments, which are, in turn, divided into sections. However, the most distinctive feature of macOS executables is their support for `fat` binaries, a.k.a. *universal binaries*, which include code for multiple CPU architectures in the same file, thus simplifying distribution compatibility without requiring separate binaries for each architecture. A `fat` executable starts with a `fat` header, which includes metadata for each architecture supported by the executable. Following the header, the file contains a separate Mach-O binary for each architecture, each with its own header, load commands, segments, and sections. The loader uses the `fat` header to select the Mach-O binary matching the system's CPU, ensuring a seamless execution across architectures.

2.2 macOS Security

macOS implements several security features to protect the system from malware and unauthorized software [5]. In our work, we focus on the techniques that the operating system employs to validate, load, and execute binary files. We describe such hardening practices in the following sections and report how they enabled the extraction of a specific set of features in Section **??**.

2.2.1 Gatekeeper, Code Signing and Notarization. They are integral components of macOS's security infrastructure, working in concert to protect users from untrusted or malicious software. *Code signing* [2] is the first step in this process, requiring developers to sign their applications with a Developer ID certificate issued by Apple. This signature ensures the authenticity of every component of the bundle and ensures that its code has not been tampered with.

Notarization adds an additional layer of security, particularly for apps distributed outside the Mac App Store. After a Mach-O executable is code-signed, developers can submit the entire app bundle that contains it to Apple's Notarization service, where it is scanned for security threats. It is important to note that notarization is not applicable to a single Mach-O executable but only to a complete and correctly formatted app bundle that contains it. If the app bundle passes the scan, Apple issues a ticket stapled to the app bundle's global signature, indicating that the app has been notarized.

Finally, *Gatekeeper* [8] is a security mechanism to verify that any downloaded app or plugin is signed with a valid Developer ID certificate, is notarized by Apple, and has not been tampered with since it was signed. It permits only notarized app bundles to be executed by default. However, the user can voluntarily allow any non-notarized app to run on the system (or be deceived into doing so by the creator of the malicious software).

2.2.2 Entitlements. They are essentially permissions that an app needs in order to access privileged system resources or user-sensitive data, perform certain operations, or disable specific security measures [7]. These permissions are encoded as key-value pairs and are embedded within the Mach-O code signature. Apple provides a broad range of entitlements [7], which are organized into categories based on the types of resources or operations they control, such as *Security*, *Privacy*, and *Notifications*. Many of them fall under the *Security* category, covering important components like the Hardened Runtime and App Sandbox. However, not all entitlements are available to developers, as some are restricted to Apple's own applications.

2.2.3 App Sandbox. It isolates applications from critical system resources and other apps, providing an additional layer of protection against potential security breaches [3]. By default, the App Sandbox restricts access to sensitive system areas, such as files, hardware, and network services, ensuring that an app can only access the specific resources it needs to function. This containment minimizes the impact of security vulnerabilities within the app by preventing malicious or compromised applications from interfering with the rest of the system. Apps that require access to restricted resources must request dedicated entitlements to access the user's documents, camera, location, etc.

2.2.4 Hardened Runtime. It is a security feature designed to provide strong protection against exploitation by enforcing strict security policies [9]. These policies include restrictions on dynamic code

Table 1: Feature categories proposed in this work. N: numeric, B: boolean. Green highlighted: macOS-specific.

Category	Features	Type	Size
Structural	Sample Total Size	N	1
	Sample Virtual Size	N	1
	Header Flags	B	26
	Load Commands Histogram	N	50
	Min/Max/Avg Load Commands Size	N	3
	Min/Max/Avg Load Commands Entropy	N	3
	Min/Max/Avg Sections Size	N	3
	Min/Max/Avg Sections Entropy	N	3
	Min/Max/Avg Segments Size	N	3
	Min/Max/Avg Segments Virtual Size	N	3
	Min/Max/Avg Segments Entropy	N	3
	Segments Histogram	N	6
	Sections Histogram	N	11
Byte	Byte Histogram	N	256
	Bytes N-grams	N	128
String	Count of strings representing common IoC	N	5
Packing	Presence of sections with suspicious names	B	1
	Presence of segments (> 20%) with high-entropy data	B	1
API	Presence of common macOS API	B	4009
Entitlements	Presence of common and security-related entitlements	B	55
Persistence	Check for login and launch items	B	2
Certificates	Status of certificate chain	N	5

generation, loading of unsigned libraries, and preventing unauthorized debugging or code injection. By default, the Hardened Runtime is enabled, and apps that wish to perform potentially dangerous operations, which could undermine security, must request specific entitlements to bypass some of these protections. For example, an app may request entitlements for Just-In-Time (JIT) compilation or other operations that the Hardened Runtime typically blocks.

In our detector, each Mach-O executable is represented by a 4,578-dimensional feature vector. As summarized in Table 1, we group each feature into eight main categories, defined according to the source used to extract them: file structure, bytes, strings, packing, APIs, entitlements, persistence techniques, and certificates. For each feature, the table shows the data type, i.e., numeric (float) or boolean, as well as its size, i.e., the number of values used to represent the feature. These features have been designed to cover the set of features commonly adopted for Windows malware detection [1, 20], and are complemented with platform-dependent features dedicated to capture the peculiarities of the Mach-O file format, such as entitlements, certificates, and those related to persistence. It is worth noting that, even though the features that are exclusively specific to macOS binaries represent only 62 out of 4,578 features, in the experiments we will show that they are valuable for the classification of macOS samples (see Subsection 4.3). The remainder of this section describes in more detail how we adapted existing indicators to macOS binaries and how we computed macOS-specific features.

Structural features. This set includes commonly used features associated with structural properties of a binary file, which have been extended from the literature on Windows malware detection [1, 20]. Specifically, in addition to common features such as total and virtual size of the sample, we consider 50 of the most common load commands that can be present in a Mach-O file [49], and count how many times each of them is included in the binary (*Load Commands Histogram* in Table 1). Furthermore, we compute additional statistical features, such as max, min, and average size and entropy of sections, load commands, and segments present in the binary. Moreover, we count the number of common segments, namely `__TEXT`, `__DATA`, `__PAGEZERO`, `__OBJC`, `__IMPORT`, `__LINKEDIT`, and sections, i.e., `__text`, `__data`, `__bss`, `__dylb`, `__cstring`, `__const`, `__la_symbol_ptr`, `__literal4`, `__literal8`, `__jump_table`, and `__pointers`. These features are summarized by the *Segments Histogram* and *Sections Histogram* in Table 1. Finally, to support `fat` binaries, we compute the average of the numeric features among the executables contained in the `fat` binary, while for boolean features (such as *Header Flags*) we apply the logical *OR* operation, i.e., if at least one of the binaries has the feature, the resulting feature is set.

Byte-based features. These features represent statistical properties of the byte sequence that composes the binary. We compute them by respectively counting the occurrence of each byte (i.e., byte histogram) and by extracting byte N-grams (N=2 in our experiments) with a hashing technique applied to map these patterns into a fixed-dimensional space of 128 features for efficient representation, following the approach in EMBER [1].

String-based features. This category captures the presence of common strings associated with Indicators of Compromise (IoC). Specifically, we extract five classes of strings that are more likely to be relevant for malware detection, namely network resources (IPs and URLs), filesystem paths, base64-encoded strings, imported libraries, and functions' names. As for the strings related to network resources, filesystem paths, and base64-encoded strings, we leverage regular expressions to identify them. On the other hand, libraries and functions' names have been extracted using the LIEF library [54]. To reduce the noise introduced by the high number and variety of strings in the binaries, we compute the number of strings belonging to each class and obtain five numerical features.

Packing-based features. Packing is a widely used technique by malware authors to obfuscate malicious code and hinder static analysis efforts, and previous research has shown that macOS malware authors also leverage it for the same purpose [62]. Indeed, some well-known samples, such as *oRat*, *IPStorm*, *ZuRu*, *Coldroot* and *OceanLotus*, use common packers such as `UPX` [62]. Our work is the first to incorporate packing-based features into a macOS malware detector, in particular by using two features that reflect the presence of packing. The first checks for the presence of sections with names related to the `UPX` packer, such as `__XHDR`, `UPX_DATA`, and `upxTEXT`. We focused on `UPX` because it is one of the most common packers for Mach-O binaries used by macOS malware authors. Indeed, researchers from MoonLock Labs [40] found that 26% of the packed malware samples analyzed in 2024 were packed with `UPX`, making it the primary choice for Mach-O compression. The second feature checks if the binary includes more than 20% of segments with high-entropy data (i.e., if the entropy is higher than 7), which indicates that the binary is likely packed or compressed. This feature is inspired by the approach of `pefile` [15] to detect packing in Windows malware samples and is also adopted in [62] for macOS binaries.

API-based features. This category of features aims to capture how the samples use common macOS APIs. Notably, while macOS APIs have previously been identified as good indicators to guide manual analysis of macOS malware [34, 61], none of the existing works have used them as features for machine learning-based detection. In this work, we show that API calls are effective for macOS malware detection when used as features (see Subsection 4.3), thereby supporting analogous findings in the Windows malware detection literature [30], where APIs have proven effective in capturing the malicious behavior. Specifically, we first leverage the official Apple documentation [4] to identify the most common frameworks used in macOS applications, such as `AppKit`, `CoreFoundation`, `SystemConfiguration`, `Kernel`, as well as those related to security and privacy, such as `Security`. To this end, we identified a total of 35 frameworks. Next, we remove all the APIs that are included in fewer than 10 samples (i.e., 0.02% of the samples) in the dataset since they are not very representative of the common behavior of the samples and generally represent outliers obtained when extracting the imported APIs using the LIEF library [54]. For each framework, we identify the 400 most common APIs[1], setting this threshold based on our finding that the average number of APIs per framework in our dataset is around 400. This process results in a total of 4,009 unique APIs, each represented by a boolean feature.

Entitlements-based features. This category, specific to macOS, captures the presence of common entitlements. Specifically, we create an initial list of entitlements by leveraging the official Apple documentation [7] to identify all the main security-relevant entitlements, such as those related to App Sandbox and Hardened Runtime. Then, to complement the above list with common entitlements used by the samples in the dataset, we select those present in at least 1% of the samples (i.e., 411) and add those not already in the initial list. Finally, we compute a boolean feature for each of the resulting 55 entitlements that we obtained through our selection process.

Persistence-based features. We leverage two boolean features to capture two common persistence techniques used in macOS: login items and launch items.

Login items are applications that start automatically at user login and run within the user's desktop session by inheriting the user's permissions. Based on previous research [61, 62], we check for the usage of common APIs such as `LSSharedFileListCreate`, `LSSharedFileListInsertItemURL`, `SMLoginItemSetEnabled`, and `registerAndReturnError`, specifically designed to manage and interact with applications that are set to launch automatically during user login. This persistence mechanism has been observed in several macOS malware families, such as *Kitm*, *NetWire*, and *WindTail* [61].

Launch items are, instead, persistence mechanisms designed for service executables, such as software updaters, background processes, and daemons. They can be classified into launch agents and launch daemons. While the former run once after login with standard user permissions and may interact with the user session, the latter are non-interactive daemons launched before user login and run with `root` permissions. As explained in [61, 62], a common way to achieve persistence by malware samples is to create at runtime a property list (*.plist*) file in `/Library/LaunchAgents` or `/LaunchAgents` for launch agents, or `/Library/LaunchDaemons` for launch daemons, with the `<key>RunAtLoad</key>` set to `true`, which tells the macOS system to start the launch item automatically. This technique has been observed in several macOS malware families, such as *AppleJeus*, *DazzleSpy*, and *EvilQuest* [61]. To detect this persistence method in our samples, we check for the presence of the `<key>RunAtLoad</key>` string within the executable. However, it is worth noting that some malware, such as *EvilQuest*, can use obfuscation techniques to hide the presence of the embedded *.plist* file [61], bypassing our simple feature extraction technique.

Certificates-based features. This set of features aims to summarize the status of the certificate chain included in the binary (if any). To achieve this, we map five boolean features to the potential states in which a certificate chain embedded in a binary may be found (see Table 1):

- *Certificate chain found:* true if the binary includes a certificate chain (i.e., if the certificates are included in the *Code Signature* load command).
- *Certificate chain expired:* true if at least one of the certificates in the chain is expired. In this case, we consider a certificate expired if the date of the analysis (i.e., 2nd December 2024) is after the expiration date of the certificate. Despite being time-dependent, this feature is included both for completeness and because it is generally required for certificate validation.
- *Certificate chain self-signed:* true if at least one of the certificates in the chain is self-signed.
- *Certificate chain revoked:* true if at least one of the certificates in the chain is revoked.
- *Certificate chain validated:* true if the root certificate is signed by the Apple Root Certification Authority (CA).

3 Dataset

As discussed in Section 1, one of the main obstacles that has severely limited macOS malware research is the lack of a large, up-to-date, and publicly available dataset of macOS binaries. What researchers have previously created for their experiments is too small to effectively train machine learning models. For instance, the dataset created and published by Pajouh *et al.* [42] in 2018 and used in subsequent studies [17, 24, 48] until 2022 contains only 152 binaries. Other datasets, such as those curated by Walkup *et al.* [59] in 2014 and Thaeler *et al.* [53] in 2024, have never been publicly released, further limiting access to comprehensive resources for macOS malware analysis.

To tackle this challenge and support the training and evaluation of our machine learning models, we assembled a new dataset of macOS executables by combining samples obtained from a variety of sources. Our new dataset includes 41,139 Mach-O executables, with 11,413 goodware samples and 29,716 malware samples, and all samples were carefully processed to ensure its suitability for large-scale analysis and machine learning applications. To further support the research community, we make our dataset publicly available [39].

[1] If a framework has fewer than 400 APIs, we include all available ones.

3.1 Sample Sources

Goodware samples were collected from the following three sources.

- **macOS installation.** We extracted 1,239 Mach-O executables from a macOS Sonoma 14.5 installation on an Apple M1 MacBook Pro. Specifically, we collected preinstalled system binaries and applications from the default installation paths, including `/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, `/Applications`, and `~/Applications`.
- **Homebrew.** We extracted 9,843 executables from macOS applications available through the Homebrew package manager [35], filtering out those whose status is either *deprecated*, *disabled*, or *outdated*.
- **Open-source macOS apps.** We collected the remaining 1,045 executables from several open-source macOS applications available on GitHub [32].

Malware samples were collected from a variety of sources.

- **Objective-See dataset.** A curated dataset of macOS malware samples [41] maintained by Patrick Wardle and also used in previous research studies [42, 53], which included 180 samples at the time we accessed it.
- **MalwareBazaar.** We downloaded all the samples from MalwareBazaar [33] tagged with `macho` and detected as malicious by at least 5 antivirus (AV) engines. At the time of the download, we found 90 samples.
- **Virus Samples Team dataset.** This dataset includes 103 samples that we downloaded from their GitHub repository [56].
- **VirusShare.** We used 2,910 malware samples from VirusShare [57]. Since the platform does not provide any API to filter for macOS samples, we leveraged the reports and information provided in the MalDICT [27] paper by selecting all samples available on VirusShare tagged as `macos`. Moreover, for each selected sample, we collected the related report to filter out all samples detected by less than 5 AV engines.
- **VirusTotal.** We monitored the VT feed from July 1st to November 13th 2023 and retained samples with the `mac` or `macho` tag, which were detected by at least 5 AV engines. This resulted in 28,380 additional macOS malware samples.

3.2 Sample Extraction

After collecting goodware and malware samples from different sources, we processed them to create a homogeneous dataset. Since the samples can be distributed using different archive file formats, such as `.dmg` and `.zip`, we implemented a fully automated pipeline that extracts the contents of the archive based on the file format and checks if it contains any Mach-O executable. After the extraction, we filtered out all binaries that are not Mach-O executables (e.g., dynamic libraries) and those compiled for a CPU architecture other than `x86-64` or `arm64` (e.g., `PowerPC`). We also excluded all the Mach-O samples intended for iOS and WatchOS by checking the *platform* field in the *Build Version* load command (`LC_BUILD_VERSION`). Finally, we processed each sample using a LIEF-based script [54] and removed those that could not be properly parsed and those lacking load commands or sections.

As for the goodware samples, we implemented an extra validation step by using VirusTotal to verify that none were detected as malicious by any of the AV engines available on the platform.

Table 2: Samples distribution by CPU architecture. Percentages in the *Total* row refer to all samples, while the others are computed relative to each label.

Label	x86-64	arm64	fat	Total
Malware	27,339 (92%)	1,812 (6%)	565 (2%)	29,716
Goodware	3,813 (33%)	2,104 (18%)	5,496 (48%)	11,413
Total	31,152 (76%)	3,916 (9%)	6,061 (15%)	41,129

Table 3: Families containing at list 1% of the malware samples.

bundlore	adload	pirrit	jailbreak	evilquest	lador	genieo	stealer
9,484 (33.84%)	7,131 (25.44%)	2,934 (10.47%)	582 (2.08%)	464 (1.66%)	454 (1.62%)	438 (1.56%)	415 (1.48%)

As a result of these processing steps, our dataset consists of 41,129 samples, including 11,413 goodware and 29,716 malware samples.

3.3 Dataset Analysis

Table 2 shows the distribution of the samples by CPU architecture: 76% of the samples are compiled for `x86-64`, 9% for `arm64` CPUs, and 15% are `fat` binaries, i.e., they support both the `x86-64` and `arm64` architectures. It is noteworthy how goodware and malware samples are distributed differently across the target architectures. In fact, the majority of the benign samples (5,496, corresponding to 48% of the goodware dataset) are `fat` binaries, while the distribution of malware samples is heavily skewed towards the `x86-64` architecture, with 27,344 samples (92% of all malicious files). This result confirms that malware authors continue to target the `x86-64` architecture [47, 51], suggesting that `x86-64` remains prevalent in the desktop Apple ecosystem.

Table 3 shows the distribution of malicious samples by family according to the most common label provided by VT. We considered only the families containing at least 1% of the total number of malware samples in the dataset. This results in eight families, which cover about 80% of all malware samples. The family distribution of our dataset, which mainly reflects the prevalence of real-world families collected in the wild, is highly imbalanced. Indeed, the top three families, namely `bundlore` [37] (33.84%), `adload` [50, 51] (25.44%), and `pirrit` [23] (10.47%), account for 70% of the malware samples.

Due to the low number of significant families and the high imbalance of the remaining ones, in this work we focused on binary classification. Nevertheless, we also performed experiments with family classification and included the results in Appendix A.

3.3.1 Presence of packing. Our dataset also allowed us to perform a large-scale analysis of packing in macOS malware, which has never been studied before. To identify packed samples, we used two static features commonly associated with packing [62]: the presence of suspicious section names related to UPX and high-entropy data sections. As shown in Table 4, we found 4,362 (~2% of the total) potentially packed samples, of which 4,069 are malicious (~14% of the malware set) and 293 (~2% of the goodware set) are benign. Among the potentially packed malware samples, the majority (3,938) are

Table 4: Samples distribution for packing-based features.

Label	Suspicious Section Names			High Entropy Sections		
	x86-64	arm64	fat	x86-64	arm64	fat
Malware	21	0	0	3917	122	9
Goodware	1	0	0	138	50	104
Total	22 (0.05%)	0 (0%)	0 (0%)	4055 (9.85%)	172 (0.42%)	113 (0.27%)

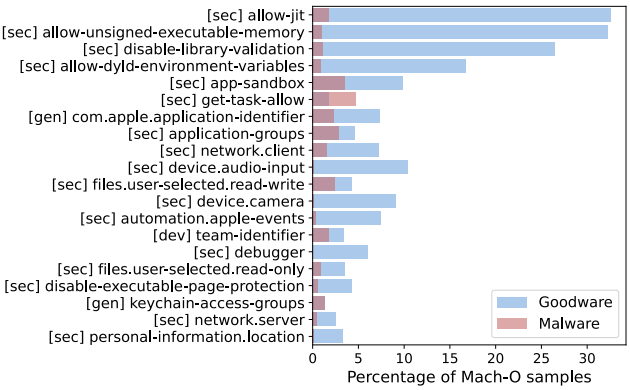

Figure 2: Top-20 entitlements. [sec], [dev] and [gen] prefixes represent the entitlements' category: security-related, developer-related and generic one, respectively.

x86-64 binaries, followed by 122 arm64 binaries and 9 fat binaries. As for the potentially packed goodware samples, we found that most of them (139) are x86-64 binaries, while 50 are arm64 and 104 are fat binaries.

> Packing is not very prevalent among macOS malware executables, with around 10% of the samples in our dataset showing clear signs of packing. As is the case for other operating systems, we also identified potentially packed benign software. Moreover, it is interesting to observe that the proportion of potentially packed samples is considerably higher for x86-64 binaries, likely because common packing techniques are more mature for this architecture.

3.3.2 Differences in macOS APIs usage. To further analyze the API usage by the samples in the dataset, we considered the top-4 frameworks containing the highest number of APIs: Kernel (including the C standard and BSD libraries), AppKit & Foundation (referred to as AppKit hereafter), CoreFoundation (including CFNetwork), and Security. Figure 3 shows the distribution of the top-20 imported APIs per framework, ordered by their total usage across both goodware and malware. The bars report the percentage of goodware and malware samples that import each API, computed over the total number of goodware and malware samples, respectively. For the Kernel and AppKit frameworks, we observed that the top APIs are more frequently used by goodware than malware. This is expected, given their role in standard system functionality and user interface development. In contrast, the APIs belonging to the CoreFoundation and Security frameworks, which are respectively adopted for low-level system and cryptographic operations, are used more frequently by malware.

> Malware samples tend to use a higher number of low-level APIs from CoreFoundation, while goodware ones tend to rely on high-level APIs. In addition, malware samples employ more security-related APIs, especially those that access the keychain (e.g., SecItemExport) or perform cryptographic operations (e.g., SecEncryptTransformCreate).

Table 5: Samples distributions for persistence mechanisms.

Label	Login Items			Launch Items		
	x86-64	arm64	fat	x86-64	arm64	fat
Malware	262	20	29	432	120	36
Goodware	103	10	269	8	8	25
Total	365	30	298	440	128	61

3.3.3 Entitlements distribution. In Figure 2, we show the distribution of the top-20 entitlements and the category of the framework containing them to assess potential differences between benign and malicious samples. Seventeen of the top-20 entitlements are security-related and are far more commonly used by goodware rather than malware. For instance, the most common entitlement, allow-jit, which is related to the Hardened Runtime and allows the binary to execute JIT-compiled code, is used by 32.6% of goodware but only 1.8% of malware. Moreover, among arm64 samples, 99.13% of goodware samples use this entitlement, compared to just 0.87% of malware samples. This further confirms that this entitlement seems to be a distinctive feature of arm64 goodware samples. Similarly, the second most common entitlement, allow-unsigned-executable-memory, is used by 32.3% of goodware and only 1.1% of malware.

> The presence of security-related entitlements such as allow-jit is a very distinctive feature of goodware samples. As we will unveil in our results, the presence of such entitlements is an important feature for distinguishing between goodware and malware.

3.3.4 Differences in persistence mechanisms. We found 1,322 samples (~3% of the dataset) that leverage persistence techniques based on the persistence-related features described in Section **??**: 899 are malicious (3% of malware) and 423 are benign (3.7% of goodware). Table 5 presents their distribution by persistence mechanism (login items vs. launch items) and architecture. As for login items, no major difference is observed between goodware and malware overall. However, looking at the single architecture distributions, we can see that this persistence technique is tightly correlated to the CPU architecture: it is used more often by malware samples based on the x86-64 architecture (262 samples), while it is more commonly used by goodware samples that are fat binaries (269 samples). On the other hand, launch items have a different trend. Indeed, this persistence mechanism is used more often by malware (588) than by goodware (41) across all architectures. Specifically, it is widely

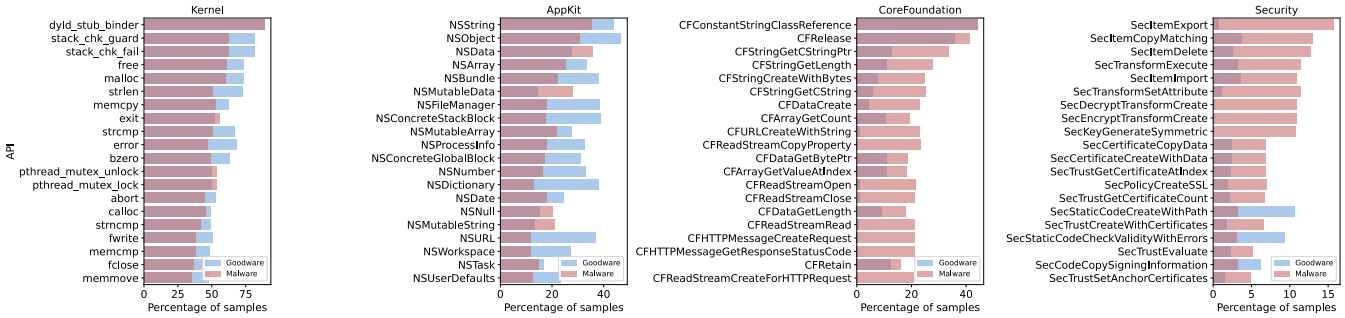

Figure 3: Analysis of the most common macOS APIs.

used by malware samples based on the `x86-64` architecture (432 samples) and less frequently by the other architectures (120 and 36 samples for `arm64` and `fat` binaries, respectively).

Persistence is not widely used in the dataset, with only 3% of samples employing such techniques. The presence of login items depends on both the sample's nature and its architecture, while the presence of launch items is more indicative of malicious behavior, being more common in malware samples regardless of the architecture.

3.3.5 Analysis of the certificate status. According to their certificate chain status, we divided the samples into the following nine groups:

- `nocert`: no certificate chain found.
- `novalid`: certificate chain found but not validated by Apple.
- `revoked valid`: certificate chain found and validated, but at least one certificate is revoked.
- `self-signed`: certificate chain found, with at least one certificate being self-signed.
- `self-signed expired`: certificate chain found, with at least one certificate being both self-signed and expired.
- `expired novalid`: certificate chain found, not validated, and with at least one certificate expired.
- `expired valid`: certificate chain found and validated, with at least one certificate expired.
- `expired revoked valid`: certificate chain found and validated, with at least one certificate expired and at least one that is revoked.
- `valid`: certificate chain found and validated by Apple.

The results, reported in Table 6, show that the majority (~83%) of malware samples do not include a certificate chain, while most goodware samples (~65%) include a certificate chain validated by Apple. Interestingly, roughly 3% of malware samples include a certificate chain validated by Apple, which is neither expired nor revoked. The presence of self-signed certificates is very low in both goodware (0.26%) and malware (0.19%), even when considering samples with certificate chains that are both expired and self-signed (0.13% for goodware and 0.04% for malware).

As we will reveal in our results, the status of the certificate chain, in particular the presence of certificates and whether it is validated by Apple, is a key feature for distinguishing between goodware and malware samples. In addition, security analysts can easily interpret this information, making it highly valuable for explainability and for identifying the root cause of a detection.

4 Experiments & Results

In this section, we introduce the experimental setup, describe the experiments conducted to train and test our detector based on the proposed features, and compare it with other state-of-the-art approaches. This is followed by a discussion of the results. Specifically, the evaluation aims to address the following research questions:

(RQ.1) Feature Effectiveness – Do the proposed macOS-specific features enhance detection performance when compared to existing state-of-the-art approaches?

(RQ.2) Feature Importance – Are the proposed macOS-specific features effectively leveraged by our detector?

(RQ.3) Feature Robustness – Do the macOS-specific features generalize to new variants? Which role do they play in the detection of new samples collected in the wild?

(RQ.4) Real-World Accuracy – How does a detector equipped with our features compare with state-of-the-art macOS models in a real-world assessment?

4.1 Experimental Setup

All experiments were conducted on an Ubuntu 22.04.6 LTS server equipped with an Intel Xeon Platinum 8160 CPU @ 2.10 GHz (64 cores) and 256 GB of RAM.

We performed a comprehensive benchmarking by comparing our set of features with those proposed in the literature by [53] and [42], the latter of which has been widely adopted in subsequent works [17, 24, 48]. Hence, we refer to all these as *Pajouh-based* in the following. We evaluated several tree-based machine learning models, namely Decision Tree (DT), Random Forest (RF) [13], and XGBoost [18], since they are widely adopted in the literature dedicated to Windows malware detection [1, 20, 25], have been proven to outperform deep learning models on tabular data, as in our case, [26], and offer many advantages such as robustness to outliers and explainability [38]. All the tree-based models were

Table 6: Percentages of sample with different certificate status.

Label	nocert	revoked valid	self-signed	self-signed expired	expired novalid	expired valid	expired revoked valid	novalid	valid
Malware	83.44	4.32	0.19	0.04	0.01	5.66	3.43	0.03	2.87
Goodware	14.65	0.12	0.26	0.13	0.00	19.59	0.07	0.00	65.17

Table 7: TPR @ 1% FPR of the evaluated models. In bold the best results for each feature set, used to compute the average improvement as the average of the relative improvements of our best model (XGBOOST) compared to the best-performing model for the other feature sets as (our − other) / other × 100.

Features	Model	x86-64	arm64	fat	multi-arch
Our	DT	39.11	81.40	77.35	85.90
	RF	92.28	95.09	89.38	96.66
	XGBOOST	**96.87**	**97.07**	**90.80**	**98.50**
Pajouh-based	DT	37.38	66.43	66.01	62.81
	RF	87.68	80.91	77.18	88.62
	XGBOOST	**88.44**	**82.86**	**78.05**	**90.90**
Thaeler	DT	38.15	40.77	65.48	72.61
	RF	81.26	69.14	71.93	81.36
	XGBOOST	**86.78**	**69.25**	**72.57**	**90.23**
MalConv	-	**82.04**	**79.81**	**81.77**	**89.06**
Avg. Improv.		+13.07%	**+26.31%**	+17.49%	+9.39%

implemented and trained by using the following Python packages: `scikit-learn` v1.5.0 [43], `xgboost` v2.1.0, and `crepes` v0.7.1 [12] for model calibration. To tune the models' hyper-parameters (see Table 13 in Appendix B) in line with the state-of-the-art [55], we performed a grid search based on 5-fold cross-validation (CV) on the training set to ensure a fair evaluation [46]. Since this resulted in five different models, all the reported results are the mean values across the five models, each one evaluated on its corresponding test set obtained from the CV split. We additionally tested the models' generalization capabilities on a temporally newer sample distribution consisting of a collection of real-world macOS benign and malicious samples collected over a 3-month period. In addition, we evaluated MalConv [45], an end-to-end deep learning model for raw byte sequences classification, widely adopted for Windows malware detection [25, 31, 44]. It was trained using the default architecture and hyper-parameters described in [45]. We evaluated the performance of all the models for all the target CPU architectures, as well as in the architecture-agnostic scenario. To this end, based on our analysis in Section 3, we split our dataset by CPU architecture and created four different *benchmarking datasets* (see Table 2): `x86-64`, `arm64`, `fat`, and `multi-arch`, the latter of which includes all the binaries regardless of the CPU architecture. For completeness, the family classification performance of our solution is reported in Appendix A.

4.2 Comparison with State-of-the-Art Detectors

The results are presented in Table 7, which shows the True Positive Rate (TPR) at 1% False Positive Rate (FPR) for all the feature sets, models, and architectures. They highlight several interesting takeaways.

First and foremost, our feature set consistently outperforms the others in all the considered scenarios. Specifically, considering the best-performing model for each feature set, our detector (based on XGBoost) achieves an average improvement of 13.07%, 26.31%, 17.49%, and 9.39% for the `x86-64`, `arm64`, `fat`, and `multi-arch` datasets, respectively. On average, across all benchmarking datasets, our detector achieves a relative improvement of 16.56% compared to the state-of-the-art detectors. To clarify, the average improvement is computed as the mean of the relative improvements of our best detector (XGBoost) over the best-performing detector for each of the other feature sets, as (our − other)/other × 100. We used the same formula to compute the average improvement (or decrease) for all the comparisons reported below.

Notably, the DT model achieved low performance (< 50% TPR) in all scenarios, suggesting that effective classification requires capturing complex feature interactions that can only be modeled by more advanced tree-based approaches such as Random Forest and XGBoost. Nevertheless, the detectors were evaluated at a very low FPR (e.g., 1%), which represents a particularly challenging operating point for malware detection systems.

Also, it is worth remarking that our detector performs especially well on the `arm64` dataset (26.31%), confirming the effectiveness of the proposed features for emerging CPU architectures.

Furthermore, the MalConv model, which uses raw byte sequences of executables as input, is the worst-performing detector in all the considered scenarios, highlighting the importance of feature engineering in macOS malware detection.

Finally, across all the proposed detectors, the performance on the `multi-arch` dataset is higher than on the other datasets. This may be due to the fact that training on a more diverse and larger dataset enables the detectors to learn distinguishing characteristics from multiple architectures.

Finally, for completeness in Appendix C we also report Figures 5 and 6 to further evaluate the performance of the target models and feature sets. The former (i.e., Figure 5) shows the ROC curves of the machine learning models evaluated in this work on the proposed features, namely Decision Tree (`DT`), Random Forest (`RF`), and XGBoost (`XGBOOST`). This figure confirms that the XGBoost model achieves the best performance among the evaluated models, especially at very low false positive rates.

The latter (i.e., Figure 6) shows the ROC curves of the XGBoost model (the best among the evaluated models) trained on the state-of-the-art feature sets evaluated in this work, i.e., ours (`our`), the features used in Pajouh *et al.* [42] (`pajouh`), those proposed by Thaeler *et al.* [53] (`thaeler`), as well as MalConv (`malconv`). These results clearly demonstrate that the XGBoost model based on our features consistently outperforms the state-of-the-art solutions at all false positive rates, hence confirming the effectiveness of the proposed features for macOS malware detection.

Table 8: TPR at 1% FPR of our detector (XGBOOST) for feature importance analysis based on *feature assessment*.

Features	x86-64	arm64	fat	multi-arch
all	**96.87**	**97.07**	**90.80**	**98.50**
generic	95.16	92.60	86.49	96.68
specific	95.72	94.88	90.44	97.96
generic vs all	-1.76%	-4.60%	-4.74%	-1.85%

Feature Effectiveness – Thanks to the use of the new proposed features, our detector achieves an **average improvement of 16.56%** over state-of-the-art solutions trained on generic features (**RQ.1**). Our experiments also show that the detectors trained on the entire multi-arch dataset provide better results, highlighting the benefits of using larger and more diverse training samples.

4.3 Effectiveness of Domain-Specific Features

In this section, we conduct two studies to evaluate the effectiveness of the macOS-specific features proposed in this work. First, we evaluate their importance, i.e., if our detector (based on the XGBoost model) effectively leverages the macOS-specific features for classification. Then, we assess the impact of removing these features on the detection performance.

Features Importance. Figure 4 shows, for each CPU architecture, the top-10 most important features based on the *total gain*, a metric commonly used to identify the features that most significantly influence XGBoost predictions [14]. Although feature importance varies across the benchmarking datasets, in all the scenarios the macOS-specific features (emphasized in bold in the figure) play a key role in the classification task. Notably, whether the certificate is validated (cert_validated) and the presence of the certificate chain (cert_present) are the most important features for the multi-arch and fat datasets. This finding aligns with our preliminary analysis in Section 3, which emphasizes their relevance in distinguishing goodware from malware samples.

As for the x86-64 dataset, the most important feature is the presence of the allow-unsigned-executable-memory entitlement, while for arm64 the presence of the allow-jit entitlement is the most important one, which has a huge impact on the model's predictions. This is in accordance with the analysis conducted in Section 3 regarding the entitlements, which shows that the presence of allow-jit is a strong indicator of a sample being goodware, especially for arm64 samples.

Finally, it is worth noting that some low-level APIs, such as fputc and system, are among the most important features for the x86-64, fat, and multi-arch datasets. To this end, by analyzing the frequency of fputc, we identified it as the most distinctive low-level API call for goodware samples, as it exhibits the largest normalized difference between the number of goodware and malware samples that utilize it. This likely explains why the model picks up this feature as a strong indicator of goodness. As for the system API, instead, we found that it is widely used by malware (33.78%) but very rarely by goodware (1.02%) in our dataset. This is somehow expected, since by analyzing some reports of macOS malware samples available on VT, we found that the system API is often used to directly execute system commands.

Feature Assessment. To further investigate the effectiveness of the macOS-specific features, we divided the proposed features into two subsets: generic and specific. The former (generic) includes only the baseline features commonly used in malware analysis, such as structural, byte-based, string-based, and packing-based features. The latter (specific) includes the macOS-specific features, i.e., certificate chain status, entitlements, persistence, and APIs.

We re-trained the XGBoost model on these two subsets by using the same experimental setup described above. This allows us to assess the impact of removing the macOS-specific features. The results, reported in Table 8, show that the detector trained on all features (all) performs better than those trained on only a subset of them. When used in isolation, the specific features outperform the generic ones, particularly for arm64 and fat. However, on the full dataset (multi-arch), removing the specific features results in only a 1.85% drop in performance (3.23% on average across all datasets).

Feature importance – Our results show that when the macOS-specific features are available, our detector considers them very important. In particular, features based on the certificate status, entitlements, and system-related APIs are among the most impactful (**RQ.2**). However, when these specific features are removed, our detector is still capable of achieving similar results by relying only on generic features.

4.4 Real-world Assessment

We now provide a real-world assessment of the proposed detector by evaluating its effectiveness in the wild and its generalization capabilities over time. To this end, we created a new *fresh* dataset of 9,000 samples, including 4,500 goodware and 4,500 malware samples, extracted from the VT feed over a period of 3 months (from September to November 2024) according to the following processing steps.

Samples processing. We selected only samples that are Mach-O executables.

Samples were sorted first using the first_submission_ date field of the VT report to select the most recent samples by submission date, and then using the last_analysis_date field to select the most recent samples by analysis date. As for the malware samples, we selected only those detected by at least 5 antivirus engines, while for the goodware samples we selected only those that have zero detections. Finally, for both goodware and malware samples, we took 4,500 samples in a stratified way, i.e. proportionally to the number of samples for each architecture in the VT feed.

Dataset Distribution. Table 9 shows the distribution of the samples for each CPU architecture in the *fresh* dataset.

Similarly to the original dataset, the x86-64 architecture is the most represented one, with 51% of the samples, followed by the fat architecture with 32% of the samples, and the arm64 architecture with 17% of the samples. Moreover, compared to the main dataset (see Table 2), the percentages related to the malware distribution highlight an increasing number of both arm64 (16% vs. 6%) and fat

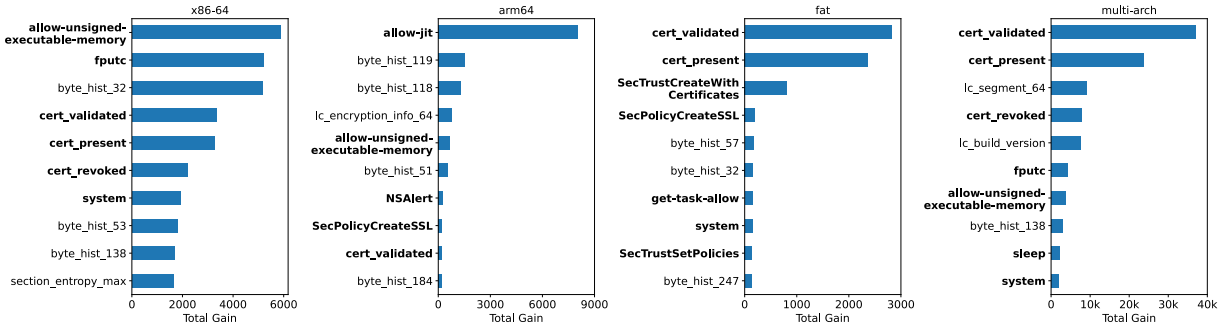

Figure 4: Feature importance based on total gain for our detector. The macOS-specific features are highlighted in bold.

Table 9: Samples distribution by CPU architecture for the *fresh* dataset. Percentages in the *Total* row refer to all samples, while the others are computed relative to each label.

Label	x86-64	arm64	fat
Malware	3,090 (68%)	715 (16%)	695 (15%)
Goodware	1,491 (33%)	801 (18%)	2,208 (49%)
Total	4,581 (51%)	1,516 (17%)	2,903 (32%)

Table 10: TPR at 1% FPR of the detectors evaluated for the real-world assessment. The average improvement w.r.t. the s.o.t.a. detectors is based on the `specific` feature set.

Detector	x86-64	arm64	fat	multi-arch
our – all	**98.20**	73.47	97.42	99.50
our – generic	92.01	58.82	69.64	90.62
our – specific	97.74	**77.55**	**97.71**	**99.50**
generic vs all	-6.30%	-19.94%	-28.52%	-8.92%
specific vs all	-0.47%	+5.55%	+0.30%	+0.00%
Pajouh-based	71.98	54.43	60.86	68.38
Thaeler	70.14	53.77	57.49	67.72
MalConv	65.42	51.10	61.22	64.38
Avg. Improv.	+37.56%	+43.34%	+65.24%	+46.21%

Table 11: F1-score at 1% FPR of the detectors evaluated for the real-world assessment.

Detector	x86-64	arm64	fat	multi-arch
our – all	94.92	75.91	96.41	96.74
our – generic	92.38	64.93	80.00	90.15
our – specific	**97.30**	**80.43**	**96.52**	**98.50**
generic vs all	-2.68%	-14.45%	-17.02%	-6.81%
specific vs all	+2.51%	+5.95%	+0.11%	+1.82%
Pajouh-based	81.83	60.12	64.10	75.21
Thaeler	79.52	56.75	62.84	72.85
MalConv	75.27	54.48	66.80	70.55
Avg. Improv.	+20.62%	+37.75%	+52.08%	+33.08%

(15% vs. 2%) samples, which confirms the trend of malware authors to target these architectures [52].

Evaluation. We used the *fresh* dataset to evaluate the generalization performance of our detector in the wild. For completeness, we also evaluate the detectors based on the features proposed by [42] and [53], as well as MalConv [45]. Moreover, as already done in the previous experiments, we evaluated different combinations of the proposed features, namely `all`, `generic`, and `specific`.

All the detectors are trained on the main dataset. Furthermore, the classification threshold was computed on the test set of the main dataset and then used to evaluate the detectors on the *fresh* dataset. The results are reported in Table 10 and Table 11, which show the detection rate (i.e., TPR) and F1-score at 1% FPR, respectively.

The tables highlight several interesting takeaways.

First, unlike the *feature assessment* experiment, in this case, removing the `specific` features leads to a significant drop in detection performance (15.92% on average). This is particularly evident for the `arm64` and `fat` datasets, where the relative performance drop is 19.94% and 28.52%, respectively. This confirms that the `specific` features generalize significantly better than the `generic` ones on novel malware samples.

Second, compared to the `all` feature set, the `specific` one achieves better performance in terms of F1-score across all the benchmarking datasets (2.60% on average), as well as a slight improvement in the TPR (1.34% on average).

The higher improvement in terms of F1-score is because the `specific` features generates less false positives, hence increasing the precision of the model. This can be explained by the fact that the additional `generic` features in the `all` feature set lead to overfitting and, consequently, a performance drop due to the curse of dimensionality [11].

Third, the detector based on our (`specific`) features consistently outperforms the state-of-the-art solutions (i.e., the detectors based on the features proposed by Pajouh *et al.* [42], Thaeler *et al.* [53], and MalConv [45]) across all the considered scenarios, with a stunning average improvement of 50.03% and 37.34% in terms of TPR and F1-score, respectively.

Finally, despite the dominance of the `x86-64` architecture, our detector is still able to effectively detect malware samples belonging to the other architectures too (see `multi-arch` scenario). However, we also noticed that all the detectors experience a performance drop when trained only on the `arm64` dataset. A possible explanation is that such drop is due to (i) the difference in the data distribution between the main and real-world dataset (6% vs 16% underlining the increasing trend of arm64 samples) and the (ii) data drift (arm64 samples in the real-world dataset differ from those in the main one).

As for this second point, we analyzed the distribution of the most important features for arm64 and found that while in the main dataset the presence of `allow-jit` entitlement (most important feature) effectively discriminates goodware from malware, this is no longer the case in the real-world dataset, where only 5% of goodware use this feature, while 0.05% of malware samples have it. Because of this, the model can no longer rely on this feature, leading to a decrease in detection performance. To mitigate this issue, we could leverage continual learning techniques [60] to continuously adapt the model to the evolving data distribution.

To further evaluate the generalization robustness of our detector, we finally performed three additional experiments. First, we assessed its performance on goodware samples whose certificate chain has not been validated by Apple and obtained a FPR of 2.4%, showing solid performance even when signing information is unavailable. Second, we evaluated the impact of packing on detection performance and we found that our detector achieved 97.5% accuracy on packed malware, a scenario that remains relatively uncommon in our real-world dataset (only ∼7% of all malware). Third, we evaluated our detector on all the signed malware included in the real-world dataset and obtained a detection rate of 96.25%, confirming its effectiveness even against signed malware.

Feature Robustness – While on the main dataset, the removal of the macOS-specific features leads to a limited decrease (3.23%) in detection performance, on newly-collected samples the performance of the generic features significantly drops (15.92%). This is due to the nature of the generic features (such as N-grams), which provide a powerful way to build "signatures" of existing malware, but tend to poorly generalize to new samples.
On the other hand, the macOS-specific features are more semantically-rich and thus can maintain their efficacy even in presence of new variants (**RQ.3**).
This is evident when comparing our detector with state-of-the-art solutions based on generic features: when tested on novel samples, our detector consistently outperforms the state-of-the-art with a remarkable average improvement of 50.03% (**RQ.4**).

5 Related Work

The detection of macOS malware through machine learning has become an area of growing interest, yet existing research exhibits notable limitations in dataset scale, adopted features, and methodological approaches (see Table 14 in Appendix D).

In one of the earliest works, Pajouh *et al.* [42] leveraged a dataset of 602 samples, including 152 malware and 450 goodware, to experiment with several machine learning models such as Naive Bayes, Support Vector Machine (SVM), and Decision Tree (DT). They extracted only structural features from the Mach-O file format, such as the number of load commands, segments, symbols, and imported libraries. Due to the limited dataset size, they employed the Synthetic Minority Over-sampling Technique (SMOTE) [16], which improved accuracy but also increased the false positive rate. The same dataset was reused by subsequent works [17, 24, 48], which further experimented with additional models such as Logistic Regression (LR) and Random Forest (RF).

In the most recent work to date, Thaeler *et al.* [53] collected a new dataset comprising 852 malware and 32,333 goodware samples, and employed a combination of structural (e.g., entropy, file size, number of load commands) and string-based (e.g., presence of suspicious strings) features to train a variety of machine learning models, including SVM, DT, and RF.

Prior studies share common limitations. They rely on small, proprietary datasets, which limits the generalizability of their findings. Their solutions are also limited to a narrow set of structural and string-based features, overlooking critical macOS-specific characteristics such as entitlements, persistence techniques, and embedded certificate status, which, as demonstrated in this work, are essential for achieving both high detection performance and human interpretability. Moreover, unlike this study, they do not rigorously assess feature importance, which is key to understanding the detector's decisions, nor do they evaluate how performance evolves over time by testing on a fresh dataset collected after model training.

6 Conclusions

In the vast majority of cases, attackers must rely on OS-specific services and functionalities to carry out harmful actions, making such features a valuable indicator for effective malware classification. This holds true for macOS as well, where malicious binaries often leverage low-level APIs to interact with system frameworks and achieve unauthorized persistence. Identifying these essential traits is fundamental to improve detection accuracy, as they directly reflect the intent and capabilities of the malware. In this work, we focused on extracting and analyzing macOS-specific features, being the first to conduct a dedicated study in this area. We presented a novel machine learning-based macOS malware detector leveraging static features derived from the Mach-O file format and macOS domain knowledge, such as embedded certificates, entitlements, and persistence techniques. Trained on a large-scale dataset of 41,129 samples, including 11,413 goodware and 29,716 malware, our detector achieves state-of-the-art detection capabilities (98.50% TPR at 1% FPR), outperforming existing approaches by 16.56%. Our detailed feature importance analysis highlights the key role of macOS-specific features, while real-world evaluation on a *fresh* dataset of temporally newer samples confirms its stunning detection performance (99.50%), corresponding to a 50.03% improvement over the state-of-the-art, and demonstrates the outstanding generalization capabilities of macOS-specific features (15.92% drop in detection rate if excluded) compared to generic ones.

Overall, we strongly believe that our work represents a significant step forward in macOS malware detection, providing the first concrete example of how domain knowledge about macOS binaries can be harnessed to build a machine learning-based detector with state-of-the-art detection capabilities.

As future work, since malware authors are continuously evolving their techniques to evade detection, we plan to investigate the adversarial robustness of our detector by exploring both current [21] and novel attacks specifically targeting the proposed macOS features.

7 Acknowledgments

This work has benefited from a government grant managed by the National Research Agency under France 2030 with reference "ANR-22-PECY-0007".

References

[1] Hyrum S. Anderson and Phil Roth. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints* (April 2018). arXiv:1804.04637

[2] Apple. 2021. App code signing process in macOS. https://support.apple.com/en-gb/guide/security/sec3ad8e6e53/web Accessed: December 16, 2025.

[3] Apple. 2025. App Sandbox. https://developer.apple.com/documentation/security/app-sandbox Accessed: December 16, 2025.

[4] Apple. 2025. Apple Documentation. https://developer.apple.com/documentation/ Accessed: December 16, 2025.

[5] Apple. 2025. Apple Platform Security. https://support.apple.com/en-us/102149 Accessed: December 16, 2025.

[6] Apple. 2025. Bundle Programming Guide. https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFBundles/BundleTypes/BundleTypes.html#//apple_ref/doc/uid/10000123i-CH101-SW1 Accessed: December 16, 2025.

[7] Apple. 2025. Entitlements. https://developer.apple.com/documentation/bundleresources/entitlements Accessed: December 16, 2025.

[8] Apple. 2025. Gatekeeper and runtime protection in macOS. https://support.apple.com/en-gb/guide/security/sec5599b66df/web Accessed: December 16, 2025.

[9] Apple. 2025. Hardened Runtime. https://developer.apple.com/documentation/security/hardened-runtime Accessed: December 16, 2025.

[10] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *NDSS*, Vol. 14. The Internet Society, 23–26.

[11] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

[12] Henrik Boström. 2022. crepes: a Python Package for Generating Conformal Regressors and Predictive Systems. In *Proceedings of the Eleventh Symposium on Conformal and Probabilistic Prediction and Applications (Proceedings of Machine Learning Research, Vol. 179)*, Ulf Johansson, Henrik Boström, Khuong An Nguyen, Zhiyuan Luo, and Lars Carlsson (Eds.). PMLR.

[13] Leo Breiman. 2001. Random forests. *Machine learning* 45 (2001), 5–32.

[14] Jason Brownlee. 2024. XGBoost Best Feature Importance Score. https://xgboosting.com/xgboost-best-feature-importance-score/

[15] Ero Carrera. 2025. Multi-platform Python module to parse and work with Portable Executable (PE) files. https://github.com/erocarrera/pefile Accessed: December 16, 2025.

[16] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.

[17] Alex Chenxingyu Chen and Kenneth Wulff. 2022. *Machine Learning for OSX Malware Detection*. Springer International Publishing, Cham, 209–222. https://doi.org/10.1007/978-3-030-74753-4_14

[18] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[19] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding linux malware. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 161–175.

[20] Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. 2023. Decoding the Secrets of Machine Learning in Malware Classification: A Deep Dive into Datasets, Feature Extraction, and Model Performance. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 60–74. https://doi.org/10.1145/3576915.3616589

[21] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. 2021. Functionality-Preserving Black-Box Optimization of Adversarial Windows Malware. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3469–3478. doi:10.1109/TIFS.2021.3082330

[22] Jonny Evans. 2023. Three-quarters of large US firms now using more Apple devices – survey. https://www.computerworld.com/article/1634358/three-quarters-of-large-us-firms-now-using-more-apple-devices-survey.html

[23] Tom Fakterman. 2023. Through the Cortex XDR Lens: macOS Pirrit Adware. https://www.paloaltonetworks.com/blog/security-operations/through-the-cortex-xdr-lens-macos-pirrit-adware/ Accessed: December 16, 2025.

[24] Samira Eisaloo Gharghasheh and Shahrzad Hadayeghparast. 2022. *Mac OS X Malware Detection with Supervised Machine Learning Algorithms*. Springer International Publishing, Cham, 193–208. https://doi.org/10.1007/978-3-030-74753-4_13

[25] Daniel Gibert. 2025. *Machine Learning for Windows Malware Detection and Classification: Methods, Challenges, and Ongoing Research*. Springer Nature Switzerland, 143–173. doi:10.1007/978-3-031-66245-4_6

[26] Leo Grinsztajn, Edouard Oyallon, and Gael Varoquaux. 2022. Why do tree-based models still outperform deep learning on typical tabular data?. In *Advances in Neural Information Processing Systems*, Vol. 35. Curran Associates, Inc., 507–520.

[27] Robert J. Joyce, Edward Raff, Charles Nicholas, and James Holt. 2023. MalDICT: Benchmark Datasets on Malware Behaviors, Platforms, Exploitation, and Packers. arXiv:2310.11706 [cs.CR]

[28] Kaspersky Team. 2023. Are Macs safe? Threats to macOS users. https://www.kaspersky.com/blog/macos-users-cyberthreats-2023/50018/

[29] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraş. 2017. Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1435–1448. doi:10.1145/3133956.3133958

[30] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. 2016. Deep Learning for Classification of Malware System Call Sequences. In *AI 2016: Advances in Artificial Intelligence*, Byeong Ho Kang and Quan Bai (Eds.). Springer International Publishing, 137–149.

[31] Matous Kozak, Luca Demetrio, Dmitrijs Trizna, and Fabio Roli. 2024. Updating Windows Malware Detectors: Balancing Robustness and Regression against Adversarial EXEmples. https://arxiv.org/abs/2405.02646

[32] Serhii Londar. 2025. Awesome macOS open source applications. https://github.com/serhii-londar/open-source-mac-os-apps Accessed: July 30, 2024.

[33] MalwareBazaar Team. 2025. MalwareBazaar. https://bazaar.abuse.ch Accessed: July 30, 2024.

[34] Modhuparna Manna, Andrew Case, Aisha Ali-Gombe, and Golden G. Richard. 2021. Modern macOS userland runtime analysis. *Forensic Science International: Digital Investigation* 38 (2021), 301221. doi:10.1016/j.fsidi.2021.301221

[35] Howell Max. 2025. The Missing Package Manager for macOS (or Linux). https://brew.sh/ Accessed: July 30, 2024.

[36] Microsoft. 2025. PE Format. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format Accessed: December 16, 2025.

[37] MITRE. 2025. Bundlore. https://attack.mitre.org/versions/v15/software/S0482/ Accessed: December 16, 2025.

[38] Christoph Molnar. 2022. *Interpretable Machine Learning* (2 ed.). Lulu.com. https://christophm.github.io/interpretable-ml-book

[39] Biagio Montaruli, Andrea Oliveri, Savino Dambra, and Davide Balzarotti. 2025. The Role of Domain-Specific Features in Malware Detection: A macOS Case Study – Dataset. https://github.com/eurecom-s3/macos-malware-dataset

[40] Moonlock Lab Team. 2024. Moonlock's 2024 macOS threat report. https://moonlock.com/moonlock-2024-macos-threat-report

[41] Objective-See Foundation. 2025. macOS Malware Collection. https://github.com/objective-see/Malware Accessed: July 30, 2024.

[42] Hamed Haddad Pajouh, Ali Dehghantanha, Raouf Khayami, and Kim-Kwang Raymond Choo. 2018. Intelligent OS X malware threat detection with code inspection. *Journal of Computer Virology and Hacking Techniques* 14, 3 (2018), 213–223.

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[44] Andrea Ponte, Dmitrijs Trizna, Luca Demetrio, Battista Biggio, Ivan Tesfai Ogbu, and Fabio Roli. 2024. SLIFER: Investigating Performance and Robustness of Malware Detection Pipelines. https://arxiv.org/abs/2405.14478

[45] Edward Raff, William Fleshman, Richard Zak, Hyrum S. Anderson, Bobby Filar, and Mark McLean. 2021. Classifying Sequences of Extreme Length with Constant Memory Applied to Malware Detection. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 11 (May 2021), 9386–9394. doi:10.1609/aaai.v35i11.17131

[46] Sebastian Raschka. 2018. Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning. http://arxiv.org/abs/1811.12808

[47] Raffaele Sabato, Phil Stokes, and Tom Hegel. 2025. BlueNoroff Hidden Risk | Threat Actor Targets Macs with Fake Crypto News and Novel Persistence. https://www.sentinelone.com/labs/bluenoroff-hidden-risk-threat-actor-targets-macs-with-fake-crypto-news-and-novel-persistence/ Accessed: December 16, 2025.

[48] Dilip Sahoo and Yash Dhawan. 2022. *Evaluation of Supervised and Unsupervised Machine Learning Classifiers for Mac OS Malware Detection*. Springer International Publishing, Cham, 159–175. https://doi.org/10.1007/978-3-030-74753-4_11

[49] Aidan Steele. 2025. OS X ABI Mach-O File Format Reference. https://github.com/aidansteele/osx-abi-macho-file-format-reference Accessed: December 16, 2025.

[50] Phil Stokes. 2021. Massive New AdLoad Campaign Goes Entirely Undetected By Apple's XProtect. https://www.sentinelone.com/labs/massive-new-adload-campaign-goes-entirely-undetected-by-apples-xprotect/ Accessed: December 16, 2025.

[51] Phil Stokes. 2025. macOS Adload: Prolific Adware Pivots Just Days After Apple's XProtect Clampdown. https://www.sentinelone.com/blog/macos-adload-prolific-adware-pivots-just-days-after-apples-xprotect-clampdown/ Accessed: December 16, 2025.

[52] Phil Stokes. 2025. macOS Cuckoo Stealer | Ensuring Detection and Defense as New Samples Rapidly Emerge. https://www.sentinelone.com/blog/macos-cuckoo-

stealer-ensuring-detection-and-defense-as-new-samples-rapidly-emerge/ Accessed: December 16, 2025.
[53] Andrew Thaeler, Yagmur Yigit, Leandros Maglaras, William J Buchanan, Naghmeh Moradpoor, and Gordon Russell. 2023. Enhancing Mac OS Malware Detection through Machine Learning and Mach-O File Analysis. In *2023 IEEE 28th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. 170–175. doi:10.1109/CAMAD59638.2023.10478430
[54] Romain Thomas. 2017. LIEF - Library to Instrument Executable Formats. https://lief.quarkslab.com/
[55] Dmitrijs Trizna, Luca Demetrio, Battista Biggio, and Fabio Roli. 2024. Nebula: Self-Attention for Dynamic Malware Analysis. *IEEE Transactions on Information Forensics and Security* 19 (2024), 6155–6167. doi:10.1109/TIFS.2024.3409083
[56] VirusSamples Team. 2025. MacOS Malware Samples - A Collection of MacOS Malware Binaries. https://github.com/MalwareSamples/Macos-Malware-Samples Accessed: July 30, 2024.
[57] VirusShare Team. 2025. VirusShare.com - Because Sharing is Caring. https://virusshare.com Accessed: December 16, 2025.
[58] VirusTotal. 2025. VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/ Accessed: December 16, 2025.
[59] Elizabeth Walkup. 2014. Mac Malware Detection via Static File Structure Analysis. https://cs229.stanford.edu/proj2014/Elizabeth%20Walkup,%20MacMalware.pdf
[60] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. 2024. A Comprehensive Survey of Continual Learning: Theory, Method and Application. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46, 8 (2024), 5362–5383. doi:10.1109/TPAMI.2024.3367329
[61] Patrick Wardle. 2022. *The Art of Mac Malware: The Guide to Analyzing Malicious Software.* No Starch Press. https://taomm.org/vol1/read.html
[62] Patrick Wardle. 2025. *The Art of Mac Malware, Volume 2: Detecting Malicious Software.* No Starch Press. https://taomm.org/vol2/read.html
[63] George D Webster, Bojan Kolosnjaji, Christian von Pentz, Julian Kirsch, Zachary D Hanif, Apostolis Zarras, and Claudia Eckert. 2017. Finding the needle: A study of the pe32 rich header and respective malware triage. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*. Springer, 119–138.

Appendix

A Family Classification

We further evaluated the effectiveness of our detector (i.e., the XGBoost model based on the proposed features) for family classification. To this end, we built a multi-class dataset consisting of all the goodware samples (11,413) from the original dataset, while for the malware samples, we considered the eight most common malware families in our dataset (see Table 3), i.e., `bundlore`, `adload`, `pirrit`, `jailbreak`, `evilquest`, `lador`, `genieo`, and `stealer`, for a total of 21,679 malware samples. Hence, in total, the multi-class dataset consists of 33,097 samples and 9 classes (8 malware families and 1 goodware class). To split the dataset into training and test sets, we also adopted a stratified approach to ensure that the distribution of the classes is the same in both the training and test sets. The experimental results are reported in Table 12, which shows the detection rate (i.e., the TPR) for each category, computed using the one-vs-all strategy, i.e., we turned the multi-class problem into multiple binary classification problems, where we trained a binary classifier for each target class and then computed the TPR at 1% FPR for each class. The results demonstrate that the proposed detector can also effectively distinguish between the main macOS malware families, achieving an average detection rate of 99.53%, with the best-performing families being `evilquest` and `lador`, which achieve a perfect detection rate of 100.00%, while the worst-performing family is `stealer`, which achieves a detection rate of 98.50%, possibly due to the low number of samples available in the dataset.

Table 12: TPR at 1% FPR of our detector trained for family-classification.

bundlore	adload	pirrit	jailbreak	evilquest
99.95	99.09	99.93	99.25	100.00
lador	**genieo**	**stealer**	**goodware**	
100.00	99.52	98.52	99.96	

Table 13: Hyper-parameters of the machine learning models evaluated in this work.

Model	Feature	Range
Decision Tree	`max_depth`	[2, 10]
	`max_features`	[`sqrt`, `log2`, `None`]
	`criterion`	[`gini`, `entropy`, `log_loss`]
	`min_samples_leaf`	[4, 8]
	`min_samples_split`	[2, 10]
Random Forest	`n_estimators`	[5, 100]
	`max_depth`	[2, 10]
	`max_features`	[`sqrt`, `log2`, `None`]
	`criterion`	[`gini`, `entropy`, `log_loss`]
	`min_samples_leaf`	[4, 8]
	`min_samples_split`	[2, 10]
XGBoost	`n_estimators`	[5, 100]
	`max_depth`	[2, 10]
	`eta`	[1e-5, 1e-1]
	`min_child_weight`	[8, 16]
	`colsample_bytree`	[0.4, 1.0]

B Hyper-parameters

Table 13 shows the hyper-parameters of the tree-based machine learning models evaluated in this work, namely Decision Tree (`DT`), Random Forest (`RF`), and XGBoost (`XGBOOST`). The hyper-parameters are tuned using a grid search approach, where we evaluate all the possible combinations of the hyper-parameters in the specified ranges.

C ROC Curves

To further support the experimental results presented in Section 4, we provide additional results in Figure 5 and Figure 6. The former (Figure 5) reports the ROC curves of the machine learning models evaluated in this work on the proposed features, namely Decision Tree (`DT`), Random Forest (`RF`), and XGBoost (`XGBOOST`). It shows that the XGBoost model achieves the best performance among the evaluated models, especially at very low false positive rates.

The latter (Figure 6) shows the ROC curves of the XGBoost model (the best among the evaluated models) trained on the state-of-the-art feature sets evaluated in this work, i.e., ours (`our`), the features used in Pajouh *et al.* [42] (`pajouh`), those proposed in Thaeler *et al.* [53] (`thaeler`), as well as MalConv (`malconv`).

These results clearly demonstrate that the XGBoost model based on our features consistently outperforms the state-of-the-art solutions at all false positive rates, hence confirming the effectiveness of the proposed features for macOS malware detection.

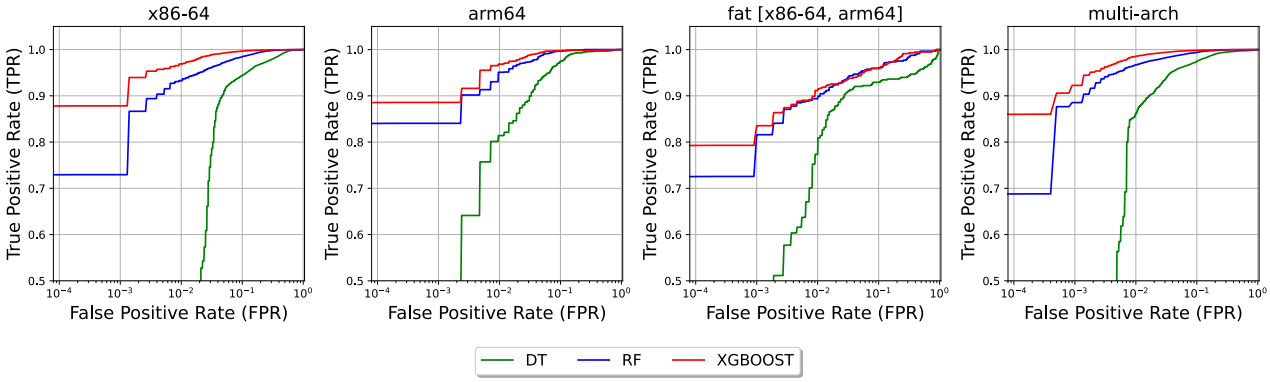

Figure 5: ROC curves of the machine learning models trained on the features proposed in this work, namely Decision Tree (`DT`), Random Forest (`RF`), and XGBoost (`XGBOOST`).

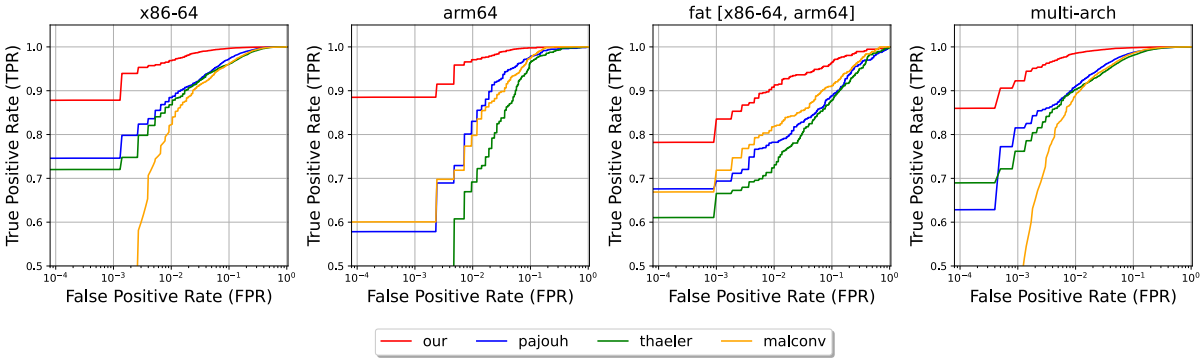

Figure 6: ROC curves of the XGBoost model trained on the state-of-the-art features sets, namely ours (`our`), the features used in Pajouh *et al.* [42] (`pajouh`), those proposed in Thaeler *et al.* [53] (`thaeler`), as well as MalConv (`malconv`)

Table 14: Related work on machine learning for macOS malware detection. Features: S structural, T: string-based, V: various (also macOS-specific)

Work	Year	Dataset			Features
		Malware	Goodware	Available	
Pajouh *et al.* [42]	2018	152	450	✗	S,T
Sahoo *et al.* [48]	2022	152	450	✗	S,T
Chen *et al.* [17]	2022	152	450	✗	S,T
Gharghasheh *et al.* [24]	2022	152	450	✗	S,T
Thaeler *et al.* [53]	2023	852	32,333	✗	S,T
This work	2025	29,716	11,413	✓	V

D Related Work

To further contextualize our work and complement the discussion in Section 5, we provide Table 14 that summarizes the main characteristics of the existing works on machine learning for macOS malware detection. In particular, it reports the number of malware and goodware samples in the dataset used by each work, whether the dataset is publicly available, and the type of features adopted for training the models.