On the Effectiveness of Binary Emulation in Malware Classification

Vasilis Vouvoutsis[a], Fran Casino[b,c], Constantinos Patsakis[a,c]

[a] *Department of Informatics, University of Piraeus, 80 Karaoli & Dimitriou str., 18534 Piraeus, Greece*
[b] *Department of Computer Engineering and Mathematics, Universitat Rovira i Virgili*
[c] *Information Management Systems Institute, Athena Research Centre, Artemidos 6, Marousi 15125, Greece*

Abstract

Malware authors continuously evolve their code base to include counter-analysis methods that can significantly hinder their detection and blocking. While malware execution in a sandboxed environment may provide insightful feedback about what the malware does in a machine, anti-virtualisation and hooking evasion methods may allow malware to bypass such detection methods. The main objective of this work is to complement sandbox execution with the use of binary emulation frameworks. The core idea is to exploit the fact that binary emulation frameworks may test samples quicker than a sandbox environment as they do not need to open a whole new virtual machine to execute the binary. While with this approach we lose the granularity of the data collected through a sandbox, one may only need to efficiently determine whether a file is malicious or to which malware family it belongs. To this end, we record the performed API calls and use them to explore the efficacy of using them as features for binary and multiclass classification. Our extensive experiments with real-world malware illustrate that this approach is very accurate, achieving state-of-the art outcomes with a statistically robust set of classification experiments while simultaneously having a relatively low computational overhead compared to traditional sandbox approaches. In fact, we compare the binary analysis results with a commercial sandbox, and our classification outperforms it at the expense of the fine-grained results that a sandbox provides.

Keywords: Malware, Binary Emulation, Classification, Machine Learning

1. Introduction

The continuous increase in the volume and complexity of malware, coupled with the introduction of new methods, introduces many challenges in detecting and blocking malware. The most utilised malware detection strategy is signature checking, which depends on pattern matching of known indicators of compromise (IOC) [1]. Whereas signature filtering is compelling for numerous sorts of malware, it is ineffectual for recognising modern malware, as, e.g. packers can easily break many such patterns. Therefore, malware authors create various malware instances from the same malware family [2]. As a result, members of the same malware family are functionally identical, even though their binaries may greatly differ. Given such a threat landscape, many researchers and practitioners try to leverage machine learning to address these challenges, which seems to live up to the expectations [3].

One of the most widely used methods to determine the behaviour of a specific piece of software and whether it is malicious or not is to execute it inside a sandbox. Practically, we create a highly monitored and controlled environment that replicates a real user environment and execute the binary inside it to track its behaviour. Note that deviations in the execution environment may lead to radically different malware behaviour [4]. Clearly, malware analysis is something that malware authors do not want to happen; therefore, they embed several evasion methods, e.g. anti-virtualisation methods or detection of monitoring processes in the arsenal of their malware [5, 6, 7]. In fact, as shown in [8], many publicly and widely used malware sandboxes have issues with these measures, which means that they can be easily bypassed. Moreover, malware sandboxes are "expensive" in various ways. They operate inside a virtual machine that needs many computational and storage resources to imitate a real one. Failure to comply with these expectations

will result in the detection of the analysis environment. Therefore, we need more cost-efficient ways to analyse malware. Additionally, as shown in [9], traditional anti-analysis methods could be used to unhook binaries and create many issues for the analysts as, in most cases, they operate with the same permissions.

Beyond sandboxes, there are several binary emulation frameworks, such as Qiling [10], Speakeasy [11], angr [12], Zelos [13], BitBlaze [14], and Binee [15], which enable analysts to emulate the execution of a binary in a host by replicating functions, system calls, and operating subsystems. The key difference here compared to sandboxing is that one can scale this better than virtual machines since binary emulation can be performed in containers that are more resource friendly. Exploiting this technology, one can understand that while in its infancy, for malware analysis, is very promising, so its potential must be further explored. The frameworks above allow for more detailed analysis by hooking and debugging binaries. However, this part is primarily manual and cannot scale. In this regard, the goal of this work is to investigate how binary analysis frameworks can facilitate binary (benign vs malicious scenario) as well as multiclass classification in a large and real-world dataset in a cost-effective manner. To this end, we perform binary emulation of 71,536 binaries and record their API calls to use them as features to feed machine learning models. While this does not leverage the full potential of binary emulation, it allows us to classify malware and even classify them into families efficiently. Evidently, the results of binary emulation cannot compete in terms of granularity with the ones of traditional sandboxes. Nevertheless, we illustrate that they are enough for the classification task, using significantly fewer computational and storage resources.

It has to be highlighted that the construction of a malware analysis system necessitates a number of architectural considerations with far-reaching implications. Whenever a program runs in a monitoring environment, an analysis component must keep track of every element of the program's execution. The time required to reset the analysis environment to a clean state is another factor that may impact the design of an analysis technique. This is required since findings can only be compared if each sample is executed in the same environment. The greater the number of samples to be examined, the greater the influence of this reset time. Clearly, containers are far more efficient than virtual machines as they are more lightweight and cloud-friendly.

Beyond scalability issues, binary emulation can also bypass some sandbox analysis limitations. For instance, when a binary is emulated, all calls are initiated by the sample under investigation. However, in a sandbox environment, since the whole operating system is being used, many calls which are recorded are initiated by processes of the operating system and have nothing to do with the analysis of the binary, introducing a lot of noise. The same applies to API hooking, whose origin might create additional noise in the analysis.

The positioning of the proposed approach is illustrated in Figure 1. In essence, static analysis is very efficient in terms of the time needed to analyse a sample and the resources required for analysis. However, it is very sensitive to changes, so an adversary can easily bypass it. Moreover, only a small amount of information about what the binary can actually do can be extracted. On the contrary, dynamic analysis with a sandbox introduces high costs in resources that have to be committed and the time needed to analyse a sample is dramatically increased. However, the analysis is more robust and fine-grained, giving us good insight into what the binary actually does. In our proposal, we leverage binary emulation with static features, which significantly reduces the cost of computational resources that have to be committed while simultaneously increasing the speed that it can be performed. Moreover, since the binary is executed, we gain an insight into what the binary actually does. Due to the amount of maturity of sandboxes and the monitoring mechanisms they have, the amount of collected information by such an approach is far greater; nevertheless, it is enough to perform precise malware classification.

In this context, the main contribution of this work is to showcase in a real-world experiment the efficacy of binary emulation for malware analysis in scale. In the bulk of literature, when binary emulation is used on a large scale, the focus is on finding bugs and vulnerabilities [16, 17]. Despite the continuous discussions about the use of binary emulation to analyse malware, all related work, belonging only to grey literature, refers to isolated and small-scale experiments which require manual interaction. Moreover, although some of these frameworks belong to companies, none of them reports using them in production, let alone scale, but rather as a tool for their analysts to examine and interact with one sample at a time. Moreover, there is no prior work on feature extraction and machine learning methods on artefacts collected from binary

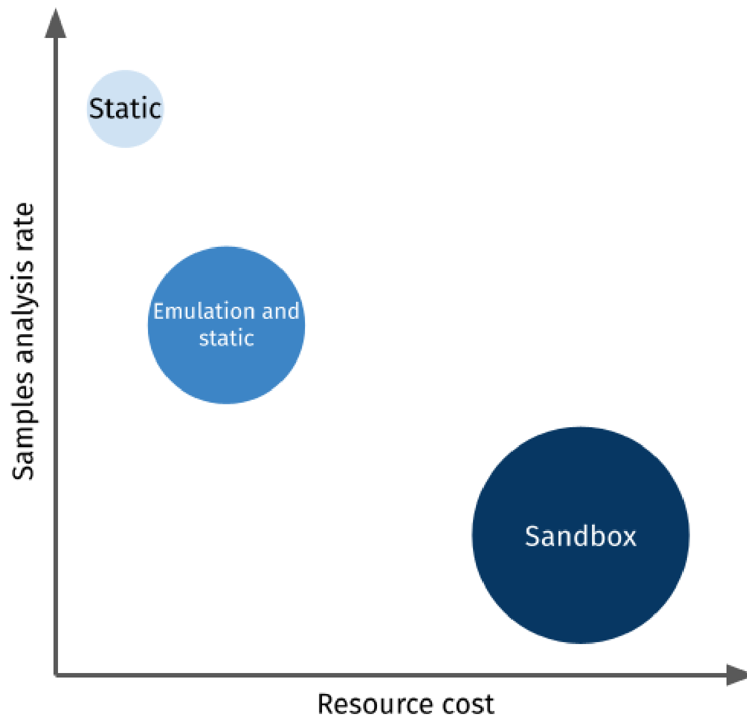

Figure 1: Positioning of the proposed approach. Sensitivity is depicted by colour, the darker the less sensitive. Insight is illustrated by size, the bigger the circle, the greater the insight.

emulation of binaries. Thus, to the best of our knowledge, this is the first work to test binary emulation in a large real-world dataset and exploit it to perform classification. As a result, the proposed approach requires a fraction of the actual resources and time that would be needed for sandbox analysis. Moreover, our analysis is more robust than static and with deeper insight. In practice, we expect to use our approach to quickly filter what has to be analysed by a sandbox after an initial static analysis, significantly reducing the resources needed to analyse malware samples.

In Figure 2 we try to illustrate the decrease of the number of samples that have to be analysed via a sandbox. In the first step, we apply static analysis on the samples applying YARA rules, extracting strings, computing fuzzy hashes (e.g. ssdeep, TLSH) and other hashes (e.g. imphash) etc. We may prune most samples from the extracted information, and we perform binary emulation on them to extract some dynamic features. These features allow us to prune the samples that have to be sent for further analysis as we may tell which is the family and whether it differentiates from the other samples of the family. As a result, what has to be executed in the sandbox is only a fraction of the samples of the previous step.

Finally, to prove the efficacy of our proposed pipeline, we go a step beyond merely comparing it with the current state of the art by comparing our approach to a commercial sandbox. This allows us to establish a baseline comparison in a real-world setting where both detection efficiency and resource allocation can be easily compared. While binary emulation cannot provide the fine-grained results of a sandbox analysis at this stage, the extracted features and our thorough and targeted feature engineering process allow our method to outperform a commercial sandbox in binary classification.

The rest of this work is structured as follows. In the next section, we provide an overview of the related work. In Section 3, we detail our automated pipeline for binary analysis and our data collection methodology. Then, we present our extracted dataset, discussing its composition and fitness for training machine learning models. In Section 4, we demonstrate how one should train the machine learning model properly and the efficacy in the binary and multiclass setting. Next, we discuss our results and compare our work with the current state of the art. To this end, we provide a comparison with a commercial sandbox and then go

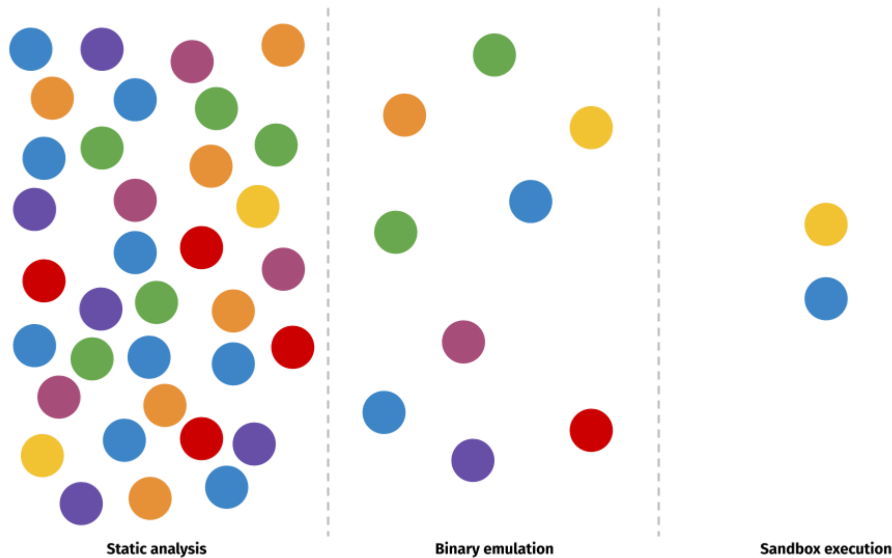

Figure 2: Filtering process of malware samples.

through the limitations of our approach. Finally, we conclude the article summarising our contributions and providing ideas for future work.

2. Related work

2.1. Malware analysis

There exist two main methods to detect and analyse malware, namely static analysis and dynamic analysis [18]. In static analysis, there is no execution of the file under inspection. Thus, the analysis is based solely on the contents and metadata of the file. As a result, only static features of the file under investigation are used to detect its behaviour. Typical examples can be file fragments (not necessarily continuous), imported libraries, strings, entropy, opcodes, etc., as they can be extracted from a file without executing it.

Dynamic analysis processes the actions taken by a program while it is running [19]. In this regard, a file under investigation is executed in a sandbox, a controlled and monitored environment, to determine what it does in terms of file system changes, network connections, opening processes, or performing specific system calls [20]. Since the file is executed, the analysis does not care about the obfuscation and packing as the file will be unpacked, memory dumps can be extracted, and the proper execution path can be revealed. Beyond simply executing it, one may also try to debug the binary that has to be investigated to find out the capabilities of the file, how it performs specific tasks, and even alter its execution flow, e.g. to bypass counter-analysis measures.

These analysis methods are well-known and, on most occasions, available to malware authors; thus, they try to bypass them or at least hinder them using several anti-analysis methods [21]. Traditional measures include packers and encryption to obfuscate their contents and create files with different signatures that allow them to bypass static analysis checks [22]. To bypass dynamic analysis, malware authors embed checks in their binary which assess the environment in which their file is executed and how it differs from a typical user environment. These checks try to determine whether the host the file is executed is monitored by, e.g. detecting user interaction (mouse movement, key pressing), checking the names of running processes for well-known analysis software, searching for environmental variables that may disclose an analysis environment, the existence of specific hooks, or even deviations from normal execution flow in terms of timing.

Of specific interest to our work is the use of API calls, as they are a very persistent characteristic that cannot be removed. The method is by no means new. One of the first works on the topic was from Forrest

et al. [23] who utilised n-grams for irregularity detection. Their tests indicate that brief sequences of system calls in operating processes provide a consistent signal of normal operation. Reddy and Pujari [24] also proposed that byte sequence and byte n-gram might be considered for feature extraction. As the number of resulting features would be exceptionally large, they utilised different strategies of feature determination and reported the use of information gain based selection strategies as the leading approach within the malware classification. Anderson et al. [25], introduced a unique malware detection technique based on graph analysis of dynamically gathered instruction traces of the target executable. Qiao et al. [26] collected API calls through dynamic analysis using Cuckoo and enriched the features with Maleur [27]. Then, they transformed the features into byte sequences that were used for binary classification. Ki et al. [28] utilised DNA sequence alignment algorithms on API call sequences and found that malware follows specific patterns. Tang and Qian [29] also used API calls to classify malware of 9 families; however, the extracted API call sequences were converted to feature images to train a convolutional neural network to perform multiclass classification. In [30], Amer and Zelinka use word embeddings to create contextual relationships between the API calls that are performed by malware and benign software to train classifiers, but also a prediction model that tries to determine whether an API call sequence is malicious or not with high accuracy.

A key point to the discussion here is how the API calls were collected. In several studies, the API calls are statically extracted [31, 30] which implies that several calls are missed due to obfuscation of the binaries. To prevent such issues, researchers usually use a sandbox-like Cuckoo[1] to collect all the calls, network connections, and filesystem changes a binary makes. While most interactions tend to happen within the first two minutes [32], this is not a panacea. Moreover, previous malware characteristics are not necessarily good indicators as malware evolves [33]. Beyond the errors that the testing environment may have and which may impede the malware analysis due to the embedded evasive methods, malware is known to be sensitive to environmental factors and exhibit different behaviour depending on where, when, and how it is executed [34, 35, 8, 4].

Christodorescu and Jha [] illustrated that strategies such as polymorphism and metamorphism are effective in evading commercial infection scanners. The reason is that syntactic signatures are insensible of the semantics of instructions.

2.2. Tools

In the following paragraphs, we mention the core tools used that allowed the current work to be implemented. Even though those tools can be swapped effortlessly for different ones, we based our decision on the fact that they have active development, extensive communities behind them and are greatly versatile to support future work.

2.2.1. Unicorn

Unicorn [36] is a CPU emulator framework based on Qemu [37]. It focuses on emulating CPU instructions that can understand emulator memory. Beyond that, Unicorn is not aware of higher-level concepts, such as dynamic libraries, system calls, I/O handling or executable formats like PE, MachO or ELF. As a result, Unicorn can only emulate raw machine instructions without Operating System (OS) context. Unicorn adds an easy-to-use API to QEMU, exposing capabilities like reading and writing memory and hooking specific locations, and memory accesses with custom callbacks [38]. Given a binary sample for a CPU platform, Unicorn, or QEMU for that matter, operates during run-time by executing the following steps [39]:

1. Convert a basic code block from the instruction set of the target platform to the instruction set of the host platform.
2. Save the translated blocks in a cache.
3. Keep a mapping from the source program counter to the destination program counter in an address lookup database.
4. Put the translated block into action.

[1] `https://cuckoosandbox.org/`

5. Continue with the next found block.

Unicorn Emulator can implement hooks for writing or reading a specific memory block [40]. It can also execute the firmware instructions, and when access to the registered address occurs, the simulation will be terminated. The simulator user will then have a callback to input/output data, control interruptions, or change the simulation state. After that, the simulation will resume.

2.2.2. Qiling.io

Qiling [10] is a sophisticated binary emulation framework powered by the Unicorn engine [36]. Qiling is intended to be a higher-level framework that uses Unicorn to simulate CPU instructions while also understanding OS. It contains executable format loaders (currently for PE, MachO, and ELF), dynamic linkers (to load and move shared libraries), syscall and IO handlers. As a result, Qiling can run executable binaries without requiring its native operating system.

Unicorn is a CPU emulator that can be scripted. Once a program makes a system call Qiling attempts to fully simulate what the host (Windows, Linux, etc.) would do. It is not easy to emulate the systems that an operating system provides. An operating system offers networking, a file system, the ability to load a binary (ELF, PE, MachO) into memory, and so on. While QEMU with complete system emulation may accomplish part of this, it lacks Unicorn's script-able control and deep analysis capabilities.

angr [12], for example, can be useful for focusing on certain portions of code. This is handy if you have reversed a binary sufficiently to know where to target. For example, parsers are generally complicated and susceptible or locating a specific input to reach a target place (i.e. CTF challenges). The difficulty that Qiling solves is that programs do not function in a vacuum; they are extremely dependent on the operating system on which they run. Emulating each operating system enables dynamic analysis that is not feasible with other frameworks.

3. Experimental Setup

We use two relatively mature frameworks to conduct our experiments using an automated pipeline, namely Qiling and Unicorn Engine. Beyond their maturity and features described in the previous section, we opted for their use due to the frequent update circles and the easy to use output they produce.

A generic overview of the workflow we developed is illustrated in Figure 3. In essence, the samples are sent for execution to Qiling, which uses the Unicorn Engine to emulate their execution. The generated code runs native on the host processor, and the processor directly executes the code generated by the compiler. In our case, the Unicorn Engine processed the instructions, and the binary was orchestrated by the Qiling binary emulation framework. Therefore, we passed each binary in Qiling, which started the binary emulation. With the help of the Unicorn engine, instructions starting from the executable's entry point are interpreted. Finally, Qiling returns a dictionary of all API calls initiated by the analysed sample during its execution. In order for Qiling to function properly, it requires several dependencies of the emulated sample to be collected and made available to it. Hence, we performed a preliminary analysis of the samples to collect from the Windows file system all required file dependencies, e.g. DLLs. The whole execution is performed in a docker container to allow for better resource allocation and scalability. The recorded API calls are then analysed, grouped, and are used to train a machine learning model that is used for classification, both binary and multiclass. The execution of most of the samples was rather fast; however, we noticed that some of them hung for an indefinite time. Since we noticed that this would happen for samples that executed more than 30sec, we introduced a timeout of one minute. The API calls were collected regardless of whether there was a graceful exit or a timeout. On average, the execution lasted 10.5 sec.

3.1. Reference dataset

To have a broad and realistic dataset to assess our methodology accurately, we opted to collect live and recent malware. Therefore, we downloaded from Malware Bazaar [41]; a well-known publicly available malware repository, all available executable malware samples; denoted with mime type `x-dosexec` and file type guess `exe` in its database. From these files, we kept only the Windows PE files, non-dynamic libraries,

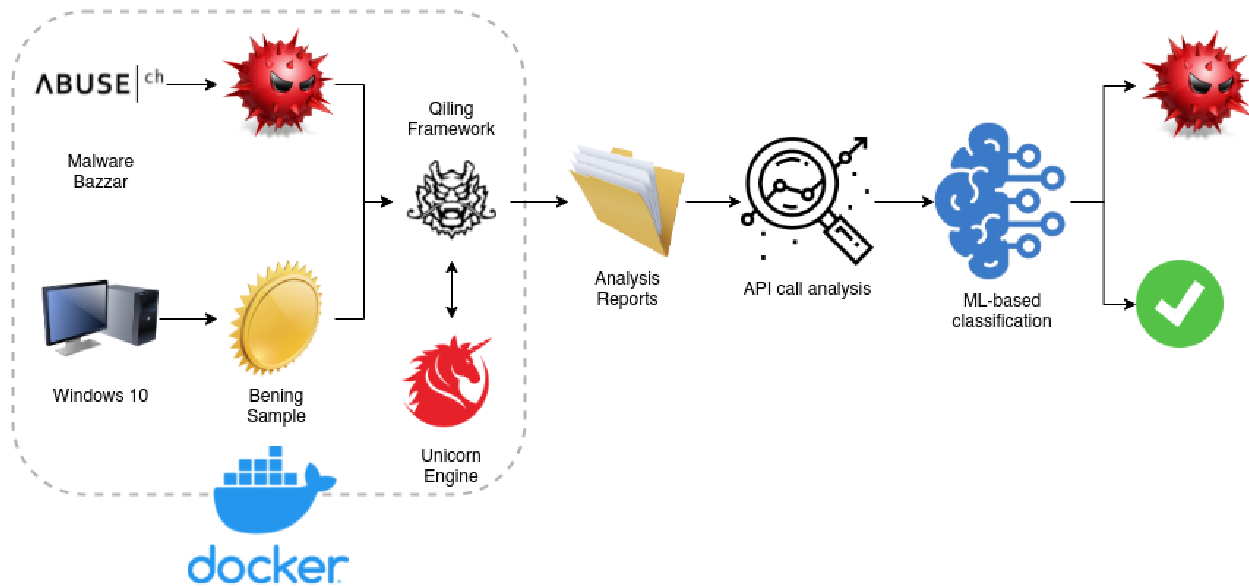

Figure 3: Overview of the developed workflow.

unmanaged code executables. That is because unmanaged code executables compile straight to machine code and are directly executed by the operating system. Thus, the total number of malicious files is 70,477. Beyond these malware samples, we added 1,059 benign files. These files were collected from a Windows 10 installation by doing a full drive file listing and then filtering out non-PE, managed executable, that is `.dll`, `.sys`, and `.mui` files.

The resulting dataset consists of 71.536 binaries belonging to different families as shown in Table 1.

3.2. Feature extraction

The resulting reports for each sample from Qiling were collected, and the API calls were extracted. Further to the API call extraction, we implemented a unification step to reduce the sparseness of the dataset and group similar artefacts together. This part is essential as there are too many different API calls which, if used as is, would end up in a significantly sparser dataset than the one obtained (which is already sparse). Note that sparse datasets entail further challenges when trying to derive statistical properties and correlations of a dataset, affecting the prediction accuracy of machine learning methods in multiple contexts [42, 43, 44, 45].

To this end, the unification involved the following steps:

- **API counter:** For each generated report, we counted every reported API call.
- **ASCII/Unicode Win32 API merge:** If any of the API's arguments takes a string, Win32 provides two versions of the API. These are the API's ASCII and Unicode versions, which produce the letters A and W, respectively. The ASCII version of the API accepts ASCII strings, whereas the Unicode version allows Unicode wide character strings [46, 47].
- **Extended Win32 API merge:** Some Win32 APIs have a more extensive version. An API's expanded version is denoted by the suffix 'Ex'. The distinction between a non-extended and extended API is that the extended version may take more parameters/arguments and may also provide extra functionality [46]. Many compromises had to be introduced to facilitate both the old 16-bit Windows API and the new 32-bit Windows API. Numerous functions are available to both APIs; hence, one of them had to change, and for backward compatibility, it is, in almost all cases, the 32-bit version.

- **C runtime library:** The Microsoft runtime library provides routines for programming the Microsoft Windows operating system. These routines automate many common programming tasks that are not provided by the C and C++ languages [48]. Many of these routines have an identical implementation with differences in the routine's name.
- **Interesting APIs**. We chose to extract the arguments of the called Win32 API and include them in our analysis. The APIs used are:
  - `LoadLibrary*` function (`libloaderapi.h`). Loads the specified module into the address space of the calling process. Other modules may be loaded as a result of the given module. Malware developers frequently opt not to employ load-time linking when calling Windows APIs. Instead, they choose to disguise such calls by dynamically loading and resolving API references.
  - `GetModuleHandle*` function (`libloaderapi.h`). Retrieves a module handle for the specified module. The calling process must have loaded the module. Some modules are loaded by default into all non-native-system processes. Such modules are `ntdll.dll` and `kernel32.dll`, therefore you do not need to call `LoadLibrary/FreeLibrary` on these and can instead just call `GetModuleHandle`.
  - `GetProcAddress` function (`libloaderapi.h`). Retrieves the address of an exported function or variable from the specified dynamic-link library (DLL). Most malware researchers are aware of how APIs can be resolved dynamically via a call to `GetProcAddress`, so they can trace down all calls to that function and check the second parameter, or `lpProcName`, for the presence of high-risk APIs (or those that can download and run the malware's actual payload.) This usually helps them narrow down and eliminate API calls that do not introduce a considerable security risk.
- **Name mangling:** Name mangling is the process of encoding function and variable names into unique names so that linkers can differentiate common names in the language. Some names use a mangling format similar to Visual C/C++ [49]. In that scenario, our strategy is to use the name's strings to produce a signature generated of the name based on Bonfante et al., [50] but more extended to not lose most of the included information.

4. Classification Experiments

In what follows, we describe our experimental setup and the tests performed to showcase the efficacy of our method. The first set of experiments focused on the enhancement of the dataset via a feature selection optimisation method, namely Boruta [51], and the binary classification. Next, we leverage the Random Forest model to perform a multiclass classification of malware samples. Note that, in addition to the classification models used in our experiments, we studied models such as Support Vector Classifier (SVC) and Gaussian Process Classifier (GPC), yet further experiments were discarded due to the poor scalability of such models when applied to high-dimensional datasets, requiring considerable memory resources and/or time to obtain similar or even slightly worse outcomes than the rest of the models.

4.1. Feature selection and binary classification

We assess the power of our proposed features to differentiate between malicious and benign samples (i.e., binary classification). We selected a set of machine learning methods to leverage a binary classification task. More concretely, we used Random Forest, a non-parametric ensemble classifier, XGBoost, which implements gradient boosted decision trees, and a k-nearest neighbours classifier (KNN).

Before tuning the hyperparameters of each model, and after observing the sparseness of the dataset (the 98.78% of the dataset's possible values are empty), we applied a feature selection process to it intending to reduce the dimensionality of the database. The latter allows us to observe which features are the most relevant and increases the efficiency of the classification task. We selected Boruta [51] as the methodology to perform feature selection optimisation. Boruta is an iterative method that performs a top-down search for relevant features by comparing their importance with importance achievable at random (i.e. using

permutations of different feature values, namely shadow copies as declared by the authors), and discarding the ones that are irrelevant for the classification task. Since Boruta is applied to ensemble classifiers such as Random Forest, we performed our optimisations using such a classification model. We applied Boruta with the parameters shown in Table 2. The outcomes of the procedure allowed us to discard 2278 features. Therefore, the binary classification experiments will be performed over a new dataset with 239 features instead of 2536 from the original dataset.

We tuned the hyperparameters of each classification model with a grid search to maximise classification performance in the task of distinguishing between benign and malicious samples. Table 2 summarises the configuration parameters that achieved the highest performance. It is important to note that hyperparameter optimisation is a crucial task closely tied with the robustness of the training procedure. In this regard, we could observe that the max_depth parameter was very relevant when maximising the efficacy of the tree-based models, increasing the possible combinations of features while exploring data interrelationships. In the case of the KNN model, using five neighbours and a weighted distance metric improved the outcomes, probably leveraging the inter-family similarities. Thus, correctly understanding the models' parameters is crucial when combined with datasets exhibiting specific features and characteristics such as our sparse, high-dimensional dataset.

The training procedure and the granularity of the metrics used are also critical. Due to the reliability of the models, the outcomes reported during all training procedures obtained F1-scores above 98%. However, due to the unbalanced nature of the dataset, reporting outstanding outcomes for the malicious class could cover a very low performance in the case of benign class. The latter was happening in all cases when the default parameters were used during the hyperparameter optimisation phase. In such cases, the classification outcomes of the benign class were very low or close to 0. Therefore, to produce robust methods that can classify all classes with high reliability, the classification reports should provide us with granular information during hyperparameter optimisation and during training. Finally, we employed standard 10-fold cross-validation and repeated such an experiment three times to get a roughly unbiased estimate of the performance of our trained predictive models.

To ensure the replicability of our experiments, we used the popular platform Google Colab [2] in its free version (2x 2.2GHz CPU and 12GB of RAM), while we utilised the implementations of the `scikit-learn`[3] library. Note that such specifications establish a bottom-line that low-mid performance desktop computers can easily outperform. We evaluate the efficacy of the trained classifiers using the standard classification metrics of precision, recall, and F_1 score.

As already discussed, since our dataset has 1059 benign and 70,477 malicious samples, we report the classification outcomes per class in Table 3. As can be observed, such outcomes highlight the robustness of our methodology in identifying both benign and malicious samples. Note that in unbalanced datasets, aggregated outcomes may hide critical errors in underrepresented classes. As observed in Table 3, our method identified the malicious samples with outstanding reliability on average regardless of the classification model used (i.e. the outcomes of RF and KNN were almost identical). Yet, several benign samples were considered malicious due to the use of features that in some cases overlapped with those used by malicious ones. However, the latter guarantees the security level that we aimed at in our system (i.e., misclassifying a malicious sample would incur further security issues than misclassifying a benign sample).

In addition to the previous experiment, we performed another binary classification experiment to showcase the robustness of our methodology further. While using all the benign samples, we randomly sampled malicious ones to obtain a dataset with a 1:1 ratio between benign and malicious. We repeated this procedure to create 100 different datasets (i.e. due to the high number of malicious samples) and performed a binary classification applying 10-fold cross-validation in each of such datasets. The average outcomes of such an experiment are reported in Table 4. The outcomes are smoother, especially in terms of recall in the benign samples. Note that the higher the number of malicious samples, the more complex feature overlapping the classifier needs to distinguish. Thus, reducing such unbalance benefited the outcomes of the benign class while slightly decreasing the performance of the malicious ones due to the lower amount

[2] https://research.google.com/colaboratory/

[3] `https://scikit-learn.org`

of training data available. A slight increase in the average σ values is observed in the case of the malicious samples, which is understandable due to the use of disparate sets in each sampled dataset. Finally, note that this experiment aims to reflect that the features and methods used are statistically sound regardless of the unbalanced nature of our dataset.

4.2. Multiclass classification

In this experiment, we aim to classify the families in a multiclass classification setting. Due to the unbalanced nature of the dataset and due to the inability of classifiers to be trained in cases where there are not enough samples [52], we selected two subsets of well-represented families in our dataset to perform a multiclass classification, as described in the following sections. Further to the dynamic features, in our features, we added one static feature, the imphash of each sample. Note that since each sample is a PE executable, one can extract the imported libraries and compute the imphash.

Practically, imphash is the MD5 hash of the ordered list of lower-case function names and the DLL names of a PE file. It has been used for malware clustering in many use cases [53, 54]. Since the hash is computed over the imported libraries and functions, it is partially correlated with the extracted features as some functions might not have been triggered in the dynamic analysis.

4.2.1. Dataset D100

For this experiment, we set up an eligibility threshold of 100 samples to create a new dataset, namely D100. Thus, the 42 families that fulfilled such conditions were selected, as highlighted in Table 1. Next, to avoid biased outcomes derived from over-represented classes, we created a balanced dataset by randomly sampling 100 elements of each family. We repeated this procedure (i.e. creating a new dataset each time) 100 times, and for each dataset, we applied 10-fold cross-validation using Random Forest classifier to leverage the multiclass classification (i.e. KNN and XGBoost performed slightly worse; thus, we used only RF for the sake of clarity). The obtained outcomes are depicted in Table 5. Most of the families that obtained very low F1-scores on average obtained a substantially higher precision than recall, a common problem of multiclass classification, especially when the classes are not well-separated [55]. Note that we used the original dataset with all the features (i.e. 2536 features) for this experiment to ensure that the classifier was able to observe all the possible features overlapping between malware families. As it can be observed, from the 42 families, 10 were identified with F1-score close to or above 0.8 (i.e. Orbus, GandCrab, Plugx, TrickBot, Heodo, BazaLoader, Sodinokibi, IcedID, QuakBot, Gozi), with Orbus, GrandCrab and Plugx reporting F1-score values above 0.95.

To provide further insight into the metrics reported by the multiclass classification experiment, we created a confusion matrix with the average classification data and depicted it in Figure 4. For each row representing a family, the columns show the predicted families by the RF classifier. In a perfect classification, all values in the main diagonal should be equal to the sample size (i.e. 100). Several families were misclassified due to the use of similar features, in some cases overlapping to the extent that made them indistinguishable for the classification method. AgentTesla, AsyncRAT, Formbook, Loki, ModiLoader, NanoCore, RedLineStealer, and RecomcosRat were often misclassified among the families with the most overlapping. We can also observe that in several cases, some malware of the same type (e.g. RATs) are misclassified between them, denoting the possible use of a similar modus operandi. If we observe the table column-wise, many families were misclassified into Adware.Generic, DiamondFox, MassLogger, OskiStealer, QuasarRAT, SmokeLoader, and Stop. Such overlapping showcases the use of the same methods and actions from the malware authors.

To provide a more comprehensive insight into the similarity between families, we used kmeans++ [56] on the classification data outcomes. We tuned the number of clusters according to the Silhouette metric [57]. By setting the initial number of clusters $k = 25$, the malware families were clustered in 15 groups as depicted in Table 6. As it can be noted, several families fall into the same groups denoting strong similarities according to the families they are confused with. Isolated families correspond in most cases to the ones that obtained high F1-score in the multiclass classification experiment. Note that different clustering strategies varying the value of k may create slightly different groups. Thus, this experiment aims only to complement the outcomes of the multiclass classification.

4.2.2. Dataset D300

In this case, we set up an eligibility threshold of 300 samples to create a new dataset, namely D300. Thus, we selected the 23 families highlighted in purple (see Table 1). Next, as in the previous experiment, we created a dataset by randomly sampling 300 elements of each family and repeated the experiment 100 times, using 10-fold cross-validation in each experiment. The average outcomes of such an experiment are depicted in Table 7.

As can be observed, the average F1-score is higher than in the case of D100. The latter can be explained due to the use of more samples to leverage the training procedure and the selection of a smaller subset of families, which may reduce the overlapping. As seen in Table 7 and in Figure 5, most of the families are captured with more reliability than in the case of D100 (see Table 5). However, since families such as GrandCrab, Orbus, and PlugX, which obtained the best classification outcomes in D100 are not present in D300, the impact on the average values were diminished.

Finally, like in the case of D100, we used kmeans++ to cluster the families of D300 and depicted the outcome in Table 8. Notably, the "Stealer" and "Loader" families were clustered together, and we can observe similarities as the ones highlighted in Table 6. We used $k = 12$ in this experiment since smaller values tend to create groups with high cardinality. As can be observed, some elements of D100 and D300 are classified in different groups. This is expected behaviour since clustering algorithms will perform differently according to each dataset. Note that the groups created may obey the heuristics of the algorithm, different data distribution, and some elements that would be in a cluster may appear in another according to an algorithm's optimisation metric, e.g. the overall information loss or the inter-cluster similarity [58, 44]. However, the distance between the elements will persist despite such group variations since the data are the same. One can tune the parameters of clustering algorithms to create different grouping strategies according to specific needs.

5. Discussion

As discussed in Section 4, the tuning and training stages are crucial to developing robust models. Note that in the cybersecurity field, obtaining samples and benchmarks can be challenging in several contexts [59], and authors cannot always have rich and easily obtainable data. In this context, the use of explainable methodologies and a detailed study of the problem that has to be solved are critical to avoid error propagation, which could deliver solutions hindering AI safety [60].

While overfitting is a critical issue when tuning the model's hyperparameters, one can alleviate it by using several strategies. First, the fact that the models can obtain accurate classification outcomes even when modifying such parameters denotes that the classes can be differentiated with a certain dynamism. In fact, this was the case since slight variations in the depth's values of RF and XGBoost, and variations in the number of neighbours in the KNN classifier reflected minor changes in the outcomes. However, the outcomes' accuracy decreased dramatically when such parameters were relaxed (for instance, a very low depth). Second, bias of the models can be detected by using cross-validation and repeating the experiments. As seen in Tables 3, 4, 5, and 7 the average standard deviation of the outcomes was very low in most of the cases, denoting stable outcomes. In cases where the dataset has enough samples to provide robust training and validation, sampling is an effective technique to guarantee unbiased estimates [52]. We used sampling in both the binary and the multiclass classification several times using random subsets of the original dataset. Thus, we can claim that the reported outcomes represent the dataset distribution in a statistically sound manner.

After analysing the outcomes obtained in Section 4, we can claim that our method can efficiently identify malware samples in a binary classification setup, and identify their family with high reliability in several cases, depending on the number of samples used and the overlapping between similar families. The latter was captured by our multiclass classification and the clustering experiments, showcasing the possible use of similar strategies and modus operandi leveraged by malicious actors across different families.

In Table 9 we compare our approach with the current state of the art in terms of reported metrics, as not all researchers use the same metrics. Note that the proper selection of metrics and the granularity of the

reported outcomes is essential to guarantee fair comparison. For instance, when datasets are imbalanced, the accuracy metric may reflect outstanding outcomes if the overrepresented class is classified with high accuracy, even if the other class or classes are not [61, 62]. Clearly, compared to the state of the art, we use the biggest dataset with the most malware families. What is even more interesting in this comparison is that the all these approaches, we had full execution of the binaries, which required execution of at least two minutes and the computational resources of a virtual machine that would resemble a typical host. On the contrary, in our approach, we emulated the execution in dockers managing to scale the experiment smoothly. Moreover, the execution lasted on average almost 10.5sec. Therefore, we managed to have comparable results with state of the art in a heterogeneous and less controlled dataset than our peers in a fraction of the time (on the scale of 1/12) and with better resource allocation.

We opted to provide a comparison based on the methods, not on the dataset; therefore, in one of the state of the art methods the researchers used Android packages and not Windows binaries as the rest. Notably, as we stress in our work, due to the constraints of dynamic analysis in terms of resources, researchers use datasets that are at least one scale smaller than ours. We argue that small datasets with a limited number of families impede methods in real-world cases as the scale factor is crucial. On the contrary, our approach outperforms the current state of the art in all aspects.

5.1. Comparison with a commercial sandbox

Intrigued by the exceptional results of the binary analysis in a real-world dataset, we opted to compare our approach to a commercial sandbox. This way, beyond showing the efficacy of our approach, we also overcome several limitations and biases of performing such a task on our own. For instance, such a task requires a lot of resources for a significant amount of time. Moreover, configuring the sandbox properly to detect and bypass modern malware evasion and anti-analysis methods is not trivial and subject to a lot of hardware, software, and configuration constraints.

Given that MalwareBazaar[4] has good interaction with Hatching's Triage[5]; the collected samples are submitted for analysis in their platform, we opted to collect the sandbox analysis results from it and compare its verdict with ours. Each sample for Windows in Triage is executed in two VMs, one with Windows 7 and one with Windows 10. Moreover, the platform performs some static tests and returns a score on the scale of one to ten of how malicious it was detected. As a rule of thumb, scores below five are not considered malicious. Therefore, we opted to collect the scores for each sample in our dataset from the API that Triage provides and use it to assess our efficacy. At this point, we have to note that since Triage provides at least two scores for each sample, one may request later to analyse the sample again; we decided to record the highest and minimum score for each sample. Nonetheless, even if it is not in our favour, we keep the highest score for the comparison.

Given that the analysis of samples from MalwareBazaar was not integrated from the very beginning of the platform, 8740 samples from our dataset were not analysed from Triage. Moreover, since the benign files that we use in our dataset are already shipped with Windows, their hashes are known and included in databases such as the National Software Reference Library (NSRL)[6] and xCyclopedia[7]. Practically, they would immediately be reported as benign as Microsoft also signs them; therefore, we believe that they should not be scanned from a sandbox. The scores of the remaining 61701 malicious samples are reported in Table 10. Practically, 5306 (8.6% of the files analysed in Triage) managed to bypass the sandbox analysis and were classified as benign. On the contrary, approximately 0.1% of the malicious samples were misclassified in our analysis, as seen in the granular information depicted in Table 3 (i.e. we report the outcomes per class; thus, malicious classification values reported a recall value of 0.999. Moreover, since Triage devotes 2.5 minutes per VM by default, the user may prolong the execution time per VM; at least 5 minutes are used per sample. Without comparing the resources, which for a sandbox are far greater as they imply the dedicated use of a VM, the sandbox approach of Triage is 28.57 times slower than ours.

[4] `https://bazaar.abuse.ch/about/`
[5] `https://tria.ge/`
[6] `https://www.nist.gov/itl/ssd/software-quality-group/national-software-reference-library-nsrl`
[7] `https://strontic.github.io/xcyclopedia/`

5.2. Limitations

While we argue that binary emulation is an excellent complimentary solution for traditional sandbox analysis, it has several limitations and constraints. Firstly, due to its maturity, one cannot analyse arbitrarily any file nor for every operating system. Indeed, in our experiments, we use only a part of the EXE files. More precisely, we kept only the Windows PE files, non-dynamic libraries, unmanaged code executables. This choice is because unmanaged code executables compile straight to machine code and are directly executed by the operating system, so binary emulation is more straightforward. Using other formats, e.g. Windows PE files based on .NET framework, would create issues for the binary emulation at the current state. Thus, this analysis approach cannot cater to the needs of arbitrary binary emulation at this stage. Nonetheless, as the frameworks mature, we expect these gaps to be filled. We also had to skip Dynamic-link libraries because they cannot be executed directly. For a PE file to be executed, an entry point (main function) is required for the execution. In addition to the above, as many community-powered frameworks, Qiling has not implemented all Windows APIs, Linux, and Darwin syscalls yet. As a result, API-complex programs are failing to complete an end-to-end binary emulation. The Qiling community continuously implements and submits new API that we expect will result in finer-grained results, more detailed results, and even new capabilities.

6. Conclusions

With the continuous advances in malware design and deployment, early detection and automated malware analysis have become an imminent need. The latter part is rather a time and resource-consuming task as, for instance, executing a binary in a sandbox environment requires the replication of an actual user host in a virtual machine, the hooking of all calls and the monitoring of all its interactions in the network, file system level, API and system calls etc. As a result, a significant amount of resources have to be dedicated to the execution of a single binary for a considerable amount of time. When the amount of daily malware samples is on the scale of 450,000, such approaches cannot meet this rate. The latter has to be considered in parallel to the fact that static analysis can be easily bypassed. As a result, we need to find ways to significantly boost the speed of dynamic analysis, or at least make it scale more efficiently.

One of the approaches that have been fostered in the past few years is binary emulation. Practically, the execution of a binary is emulated, so there is no need to utilise a whole virtual machine. Additionally, the analyst may interrupt the execution of the binary and manipulate it to explore additional execution paths. This work performs the first evaluation of binary emulation in malware classification to the best of our knowledge.0- Notably, this is verified with a real-world experiment against a commercial sandbox. We argue that the above proves the prevalence of our method compared to the state of the art and practice.

Due to the current maturity of existing binary emulation frameworks, not all syscalls have been implemented, so the collected information is a fragment of their actual potential. We argue that we managed to have such outstanding results with binary emulation in such early stages in a real-world dataset with diverse malware families against a mature and commercial sandbox clearly indicates that the results are robust and subject to further greater improvements. The study of different feature selection methods and their performance in the context of binary emulation in malware classification is an interesting research line that we will explore in future work. Since this efficacy is coupled with a significant reduction in resource allocation at a fraction of the time of sandbox approaches, it implies that this line of research can efficiently complement current approaches to timely counter malware spread. Following this line of research, in the future, we plan to explore the exploitation of further static features along with the dynamic ones in the context of binary emulation. Moreover, we plan to explore the effectiveness of symbolic and concolic execution in scale to determine whether they can meet the requirements of real-world scenarios.

Acknowledgements

This work was supported by the European Commission under the Horizon 2020 Programme (H2020), as part of the projects *LOCARD* (Grant Agreement no. 832735), HEROES (Grant Agreement no. 101021801),

and CyberSec4Europe (Grant Agreement no. 830929). Fran Casino was supported by the Beatriu de Pinós programme of the Government of Catalonia (Grant No. 2020 BP 00035).

The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

References

[1] M. Yong Wong, M. Landen, M. Antonakakis, D. M. Blough, E. M. Redmiles, M. Ahamad, An inside look into the practice of malware analysis, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 3053–3069. doi:10.1145/3460120.3484759. URL https://doi.org/10.1145/3460120.3484759

[2] A. Dinh, D. Brill, Y. Li, W. He, Malware sequence alignment, in: 2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), IEEE, 2016, pp. 613–617.

[3] D. Ucci, L. Aniello, R. Baldoni, Survey of machine learning techniques for malware analysis, Computers & Security 81 (2019) 123–147.

[4] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, T. Dumitras, When malware changed its mind: An empirical study of variable program behaviors in the real world, in: 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, 2021, pp. 3487–3504.

[5] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes, et al., Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, 2016, pp. 165–187.

[6] E. Rudd, A. Rozsa, M. Gunther, T. Boult, A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions, IEEE Communications Surveys & Tutorials 19 (2) (2017) 1145–1172.

[7] A. Bulazel, B. Yener, A survey on automated dynamic malware analysis evasion and counter-evasion: PC, mobile, and web, in: Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium, ACM, ACM, New York, NY, USA, 2017, p. 2.

[8] V. Koutsokostas, C. Patsakis, Python and malware: Developing stealth and evasive malware without obfuscation, in: S. D. C. di Vimercati, P. Samarati (Eds.), Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021, July 6-8, 2021, SCITEPRESS, 2021, pp. 125–136.

[9] T. Apostolopoulos, V. Katos, K.-K. R. Choo, C. Patsakis, Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks, Future Generation Computer Systems 116 (2021) 393–405.

[10] qiling.io, Qiling framework, https://github.com/qilingframework/qiling, accessed: 2021-03-05 (2021).

[11] Mandiant, speakeasy, https://github.com/mandiant/speakeasy, accessed: 2021-12-29 (2021).

[12] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, G. Vigna, SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis, in: IEEE Symposium on Security and Privacy, 2016, pp. 138–157.

[13] Zeropoint Dynamics, LLC, Zelos, https://github.com/zeropointdynamics/zelos, accessed: 2021-12-29 (2021).

[14] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena, Bitblaze: A new approach to computer security via binary analysis, in: International Conference on Information Systems Security, Springer, 2008, pp. 1–25.

[15] Carbon Black, Binee, https://github.com/carbonblack/binee, accessed: 2021-1-4 (2021).

[16] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, K. R. Butler, Firmusb: Vetting usb device firmware using domain informed symbolic execution, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 2245–2262.

[17] H. Peng, Y. Shoshitaishvili, M. Payer, T-fuzz: fuzzing by program transformation, in: 2018 IEEE Symposium on Security and Privacy (SP), IEEE, 2018, pp. 697–710.

[18] E. Gandotra, D. Bansal, S. Sofat, Malware analysis and classification: A survey, Journal of Information Security 5 (02) (2014) 56.

[19] M. Egele, T. Scholte, E. Kirda, C. Kruegel, A survey on automated dynamic malware-analysis techniques and tools, ACM computing surveys (CSUR) 44 (2) (2008) 1–42.

[20] O. Or-Meir, N. Nissim, Y. Elovici, L. Rokach, Dynamic malware analysis in the modern era—a state of the art survey, ACM Computing Surveys (CSUR) 52 (5) (2019) 1–48.

[21] R. R. Branco, G. N. Barbosa, P. D. Neto, Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies, in: Blackhat USA, 2012.

[22] M. Christodorescu, S. Jha, Static analysis of executables to detect malicious patterns, Tech. rep., Wisconsin Univ-Madison Dept of Computer Sciences (2006).

[23] S. Forrest, S. A. Hofmeyr, A. Somayaji, T. A. Longstaff, A sense of self for unix processes, in: Proceedings 1996 IEEE Symposium on Security and Privacy, IEEE, 1996, pp. 120–128.

[24] D. K. S. Reddy, A. K. Pujari, N-gram analysis for computer virus detection, Journal in Computer Virology 2 (3) (2006) 231–239.

[25] B. Anderson, D. Quist, J. Neil, C. Storlie, T. Lane, Graph-based malware detection using dynamic analysis, Journal in computer Virology 7 (4) (2011) 247–258.

[26] Y. Qiao, Y. Yang, J. He, C. Tang, Z. Liu, Cbm: Free, automatic malware analysis framework using api call sequences, in: F. Sun, T. Li, H. Li (Eds.), Knowledge Engineering and Management, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 225–236.

[27] K. Rieck, P. Trinius, C. Willems, T. Holz, Automatic analysis of malware behavior using machine learning, Journal of Computer Security 19 (4) (2011) 639–668.

[28] Y. Ki, E. Kim, H. K. Kim, A novel approach to detect malware based on api call sequence analysis, International Journal of Distributed Sensor Networks 11 (6) (2015) 659101.

[29] M. Tang, Q. Qian, Dynamic api call sequence visualisation for malware classification, IET Information Security 13 (4) (2019) 367–377.

[30] E. Amer, I. Zelinka, A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence, Computers & Security 92 (2020) 101760.

[31] Y. Aafer, W. Du, H. Yin, Droidapiminer: Mining api-level features for robust malware detection in android, in: International conference on security and privacy in communication systems, Springer, 2013, pp. 86–103.

[32] A. Küchler, A. Mantovani, Y. Han, L. Bilge, D. Balzarotti, Does every second count? time-based evolution of malware behavior in sandboxes, in: Proceedings of the Network and Distributed System Security Symposium, NDSS, The Internet Society, 2021.

[33] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, L. Cavallaro, Transcend: Detecting concept drift in malware classification models, in: 26th USENIX Security Symposium (USENIX Security 17), USENIX Association, Vancouver, BC, 2017, pp. 625–642.
URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/jordaney

[34] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, G. Vigna, Efficient detection of split personalities in malware., in: NDSS, Citeseer, 2010.

[35] M. Lindorfer, C. Kolbitsch, P. M. Comparetti, Detecting environment-sensitive malware, in: International Workshop on Recent Advances in Intrusion Detection, Springer, 2011, pp. 338–357.

[36] Unicorn Engine, Unicorn engine, https://github.com/unicorn-engine/unicorn, accessed: 2021-03-05 (2021).

[37] N. A. Quynh, D. H. Vu, Unicorn: Next generation CPU emulator framework, BlackHat USA 476 (2015).

[38] D. Maier, L. Seidel, S. Park, Basesafe: Baseband sanitized fuzzing through emulation, in: Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, 2020, pp. 122–132.

[39] D. Maier, B. Radtke, B. Harren, Unicorefuzz: On the viability of emulation for kernelspace fuzzing, in: 13th USENIX Workshop on Offensive Technologies (WOOT 19), USENIX Association, Santa Clara, CA, 2019.
URL https://www.usenix.org/conference/woot19/presentation/maier

[40] T. Jakubík, Cortex-m simulator, in: 2020 International Conference on Applied Electronics (AE), IEEE, 2020, pp. 1–4.

[41] abuse.ch, MalwareBazaar, https://bazaar.abuse.ch/, accessed: 2021-07-10 (2021).

[42] O. Kuss, Global goodness-of-fit tests in logistic regression with sparse data, Statistics in medicine 21 (24) (2002) 3789–3801.

[43] A. Zigomitros, F. Casino, A. Solanas, C. Patsakis, A survey on privacy properties for data publishing of relational data, IEEE Access 8 (2020) 51071–51099. doi:10.1109/ACCESS.2020.2980235.

[44] F. Casino, C. Patsakis, A. Solanas, Privacy-preserving collaborative filtering: A new approach based on variable-group-size microaggregation, Electronic Commerce Research and Applications 38 (2019) 100895.

[45] J. Li, T. Zhang, W. Luo, J. Yang, X.-T. Yuan, J. Zhang, Sparseness analysis in the pretraining of deep neural networks, IEEE transactions on neural networks and learning systems 28 (6) (2016) 1425–1438.

[46] A. Mohanta, A. Saldanha, Windows internals, in: Malware Analysis and Detection Engineering, Springer, 2020, pp. 123–162.

[47] Microsoft, MSDN: Working with Strings, https://docs.microsoft.com/en-us/windows/win32/learnwin32/working-with-strings, accessed: 2021-08-15.

[48] Microsoft, MSDN: Microsoft C runtime library (CRT) reference, https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/crt-alphabetical-function-reference?view=msvc-160, accessed: 2021-08-19.

[49] F. Agner, Calling conventions for different c++ compilers and operating systems, http://www.agner.org/optimize/calling_conventions.pdf (2014).

[50] G. Bonfante, J. O. Nogues, Function classification for the retro-engineering of malwares, in: International Symposium on Foundations and Practice of Security, Springer, 2016, pp. 241–255.

[51] M. B. Kursa, W. R. Rudnicki, et al., Feature selection with the boruta package, J Stat Softw 36 (11) (2010) 1–13.

[52] F. Casino, N. Lykousas, I. Homoliak, C. Patsakis, J. Hernandez-Castro, Intercepting hail hydra: Real-time detection of algorithmically generated domains, Journal of Network and Computer Applications 190 (2021) 103135.

[53] N. Moran, J. T. Bennett, Supply chain analysis: From quartermaster to sunshop, Vol. 11, FireEye, 2013.

[54] Mandiant, Tracking malware with import hashing, https://www.mandiant.com/resources/tracking-malware-import-hashing (2014).

[55] D. Silva-Palacios, C. Ferri, M. J. Ramírez-Quintana, Improving performance of multiclass classification by inducing class hierarchies, Procedia Computer Science 108 (2017) 1692–1701.

[56] D. Arthur, S. Vassilvitskii, K-means++: The advantages of careful seeding, in: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '07, Society for Industrial and Applied Mathematics, USA, 2007, p. 1027–1035.

[57] P. J. Rousseeuw, Silhouettes: a graphical aid to the interpretation and validation of cluster analysis, Journal of computational and applied mathematics 20 (1987) 53–65.

[58] A. Maya-López, F. Casino, A. Solanas, Improving multivariate microaggregation through hamiltonian paths and optimal univariate microaggregation, Symmetry 13 (6) (2021) 916.

[59] F. Casino, T. K. Dasaklis, G. P. Spathoulas, M. Anagnostopoulos, A. Ghosal, I. Böröcz, A. Solanas, M. Conti, C. Patsakis, Research trends, challenges, and emerging topics in digital forensics: A review of reviews, IEEE Access 10 (2022) 25464–25493. doi:10.1109/ACCESS.2022.3154059.

[60] R. V. Yampolskiy, M. Spellchecker, Artificial intelligence safety and cybersecurity: A timeline of ai failures, arXiv preprint arXiv:1610.07997 (2016).

[61] E. Burnaev, P. Erofeev, A. Papanov, Influence of resampling on accuracy of imbalanced classification, in: Eighth international conference on machine vision (ICMV 2015), Vol. 9875, SPIE, 2015, pp. 423–427.

[62] M. Bekkar, H. K. Djemaa, T. A. Alitouche, Evaluation measures for models assessment over imbalanced data sets, J Inf Eng Appl 3 (10) (2013).

[63] D. Uppal, R. Sinha, V. Mehra, V. Jain, Malware detection and classification based on extraction of api sequences, in: 2014 International conference on advances in computing, communications and informatics (ICACCI), IEEE, 2014, pp. 2337–2342.

[64] S. Naval, V. Laxmi, M. Rajarajan, M. S. Gaur, M. Conti, Employing program semantics for malware detection, IEEE Transactions on Information Forensics and Security 10 (12) (2015) 2591–2604.

[65] F. O. Catak, A. F. Yazı, O. Elezaj, J. Ahmed, Deep learning based sequential model for malware analysis using windows exe api calls, PeerJ Computer Science 6 (2020) e285.

[66] M. Ficco, Malware analysis by combining multiple detectors and observation windows, IEEE Transactions on Computers (2021).

[67] D. Brubacher, Detours: Binary interception of win32 functions, in: Windows NT 3rd Symposium (Windows NT 3rd Symposium), USENIX Association, Seattle, WA, 1999.
URL `https://www.usenix.org/conference/windows-nt-3rd-symposium/detours-binary-interception-win32-functions`

[68] A. Dinaburg, P. Royal, M. Sharif, W. Lee, Ether: malware analysis via hardware virtualization extensions, in: Proceedings of the 15th ACM conference on Computer and communications security, 2008, pp. 51–62.

Family	#	Family	#	Family	#	Family	#	Family	#
Heodo	29998	NetSupport	37	Lazarus	7	SakulaRAT	3	VegaLocker	1
Unknown	9520	Riskware.Generic	37	Lu0Bot	7	MyloBot	3	JigsawLocker	1
AgentTesla	3105	Bunitu	37	SendSafe	7	Vidar	3	SocksBot	1
TrickBot	2739	Chthonic	36	FIN7	7	Adware.InstalleRex	3	GoCryptoLocker	1
QuakBot	2462	Meterpreter	36	Lucifer	7	BlackKingdom	2	GoldMax	1
RaccoonStealer	1930	VTFlooder	36	Adware.Eorezo	7	Suncrypt	2	Snojan	1
Loki	1907	GCleaner	35	StrongPity	7	DarktrackRAT	2	Gootkit	1
FormBook	1854	DarkVNC	35	Citadel	6	Djvu	2	Rasftuby	1
RedLineStealer	1650	Socelars	33	Loda	6	AvosLocker	2	Siplog	1
RemcosRAT	1353	PandaZeuS	33	RevCodeRAT	6	RTM	2	H1N1	1
AveMariaRAT	1232	RemoteManipulator	31	Ranzy	6	Cuba	2	Hamweq	1
ArkeiStealer	850	TaurusStealer	30	Matrix	6	FuxSocy	2	Shiotob	1
NanoCore	819	Adware.ExtenBro	29	TVRat	6	Spambot.Kelihos	2	ShikataGaNai	1
Gozi	742	DarkSide	27	Bancteian	6	Adwind	2	Reconyc	1
CobaltStrike	668	KPOTStealer	27	Turla	5	RanzyLocker	2	Jackpot	1
Smoke Loader	436	Babuk	27	KINS	5	Arechclient2	2	Renamer	1
NetWire	387	YoungLotus	27	Blackmoon	5	ZeusSphinx	2	OzoneRAT	1
ModiLoader	383	Cerber	26	Worm.Dorkbot	5	nccTrojan	2	Retefe	1
BazaLoader	364	TriumphLoader	24	Adware.InstallCore	5	Adware.Adload	2	Milum	1
njrat	355	Andromeda	22	Zegost	5	M00nD3v	2	Micropsia	1
MassLogger	341	RevengeRAT	20	SchoolBoy	5	Ryuk	2	Rustyloader	1
AZORult	328	BazarCall	20	LockFile	5	Adware.DownloadAdmin	2	Mercurial	1
CryptBot	326	Hancitor	20	DoejoCrypt	5	Xtrat	2	LodaRAT	1
DanaBot	322	Troldesh	20	Locky	5	Mimikatz	2	Scarab	1
IcedID	287	Zeppelin	19	StormKitty	5	Qulab	2	LolKek	1
Stop	278	CyberGate	18	FinderBot	5	Jigsaw	2	MSILStealer	1
SnakeKeylogger	230	LockBit	18	FlawedAmmyy	5	Adware.LoadMoney	2	Sazoora	1
HawkEye	230	Osiris	18	RagnarLocker	5	Adware.PushWare	2	Maener	1
Amadey	201	Mespinoza	17	Maze	5	IRCbot	2	MaktubLocker	1
Pony	194	MedusaLocker	17	PhoenixKeylogger	5	Hive	2	Maoloa	1
PlugX	167	Cutwail	16	Worm.Ramnit	4	Urelas	2	GlobeImposter	1
Adware.Generic	161	Tofsee	16	Vjw0rm	4	ShipUp	2	Gelsemium	1
Orbus	154	GoldenSpy	16	Nemty	4	Grandoreiro	2	Solaso	1
GandCrab	154	CoinMiner.XMRig	15	IAmTheKing	4	Ardamax	2	Foudre	1
QuasarRAT	152	Sage	15	WannaCry	4	DemonWare	2	CobianRAT	1
ZeuS	143	Nitol	15	StealthWorker	4	BlueBot	1	CoderWare	1
BitRAT	140	ISRStealer	15	MyDoom	4	BlackNET	1	VHDLocker	1
DiamondFox	134	Avaddon	15	HelloKitty	4	BlackRAT	1	VHD	1
FickerStealer	133	WastedLocker	15	LegionLoader	4	Blackbone	1	CrimsonRAT	1
Dridex	129	ObliqueRAT	14	TinyNuke	4	BlackRose	1	Crypt888	1
OskiStealer	121	Neshta	14	Zatoxp	4	SaintBot	1	Upatre	1
AsyncRAT	120	a310Logger	14	PandaStealer	4	BillGates	1	UniWinniCrypt	1
Sodinokibi	113	PredatorStealer	14	Ostap	4	Adware.Techsnab	1	RansomEXX	1
Glupteba	97	Macoute	13	Dorv	4	modi	1	TorrentLocker	1
DarkComet	93	Downloader.Upatre	13	LuminosityLink	4	Zeoticus	1	TeslaCrypt	1
DCRat	92	Makop	13	Ramnit	4	Adware.CloudScout	1	Ransomware	1
ParallaxRAT	88	LimeRAT	12	Xorist	4	Adware.Duote	1	TA505	1
ServHelper	87	Neutrino	12	SysVenFak	4	Pykspa	1	Ransomware.Nemty	1
Adware.Breitschopp	87	OrcusRAT	11	Anyplace	4	Pojie	1	OOO CM	1
CoinMiner	85	WSHRAT	11	BlackShades	4	Adware.FlyStudio	1	NukeSped	1
Matiex	83	Kovter	10	Bandook	4	Adware.InstallMonster	1	EduRansom	1
SystemBC	77	BlackMatter	10	Deathransom	3	Adware.Qjwmonkey	1	Encrpt3d	1
Drolnux	67	Shifu	10	ShurkStealer	3	XpertRAT	1	Erica	1
ZLoader	66	CryLock	10	Floxif	3	BigLock	1	Exorcist	1
TeamBot	64	Snatch	9	Worm.Virut	3	AnyDesk	1	Ransomware.Petya	1
Adware.FileTour	58	Expiro	8	FONIX	3	AresRAT	1	NexusStealer	1
GuLoader	58	Gh0stRAT	8	Bancos	3	Wintenzz	1	Netsky	1
BuerLoader	57	MountLocker	8	Pitou	3	Winlock	1	PyXie	1
Tinba	53	Balaclava	8	Adware.Koutodoor	3	WellMess	1	FlawedAmmyyRAT	1
NetWalker	52	Phobos	8	BozokRAT	3	BabylonRAT	1	Fonix	1
Conti	49	VMZeuS	8	VirLock	3	RDAT	1	SpyEye	1
Dharma	46	Redosdru	7	ClipBanker	3	Beastdoor	1	PredatorTheThief	1
Neurevt	45	1xxbot	7	Warezov	3	Bentley	1		
404Keylogger	43	ImminentRAT	7	BreakWin	3	Carbanak	1	Benign	1059
Phorpiex	39	Nefilim	7	Clop	3	TinyLoader	1		

Table 1: Composition of our Dataset. The families highlighted in green and purple were used in the multiclass classification experiments in Section 4.2.1. Only the families highlighted in purple were used in the experiments of Section 4.2.2.

Table 2: Hyperparameters of the feature selection model and the classification methods.

Model	Best configuration
Boruta	learning_rate= 0.05, max_iter= 50, max_depth= 5, perc= 90
Random Forest	n_estimators= 100, max_depth= 20
XGBoost	learning_rate= 0.02, max_depth= 10, subsample= 0.8
KNN	n_neighbors= 5, weights= distance

Table 3: Average outcomes and their corresponding standard deviation σ. The best F1-score outcomes are highlighted in green.

	Benign						**Malicious**					
Method	**precision**		**recall**		**f1**		**precision**		**recall**		**f1**	
	Avg.	σ	**Avg.**	σ	**Avg.**	σ	**Avg.**	σ	**Avg.**	σ	**Avg.**	σ
RF	0.962	0.020	0.875	0.001	0.916	0.046	0.998	<0.001	0.999	0.025	0.999	<0.001
XGBoost	0.949	0.021	0.862	0.037	0.903	0.024	0.998	0.001	0.999	<0.001	0.999	<0.001
KNN	0.944	0.023	0.891	0.031	0.916	0.020	0.998	<0.001	0.999	<0.001	0.999	<0.001

	Micro average						**Macro average**					
Method	**precision**		**recall**		**f1**		**precision**		**recall**		**f1**	
	Avg.	σ	**Avg.**	σ	**Avg.**	σ	**Avg.**	σ	**Avg.**	σ	**Avg.**	σ
RF	0.998	0.001	0.998	0.001	0.998	0.001	0.980	0.010	0.937	0.023	0.957	0.013
XGBoost	0.997	0.001	0.997	0.001	0.997	0.001	0.974	0.011	0.931	0.018	0.951	0.012
KNN	0.998	<0.001	0.998	<0.001	0.998	0.001	0.971	0.012	0.945	0.016	0.958	0.010

Table 4: Average outcomes and their corresponding standard deviation σ in the binary classification with a 1:1 ratio between benign and malicious samples. The best F1-score outcomes were highlighted in green.

	Benign						**Malicious**					
Method	**precision**		**recall**		**f1**		**precision**		**recall**		**f1**	
	Avg.	σ	**Avg.**	σ	**Avg.**	σ	**Avg.**	σ	**Avg.**	σ	**Avg.**	σ
RF	0.963	0.020	0.977	0.015	0.970	0.012	0.977	0.014	0.962	0.021	0.969	0.013
XGBoost	0.967	0.017	0.957	0.020	0.962	0.013	0.958	0.019	0.967	0.018	0.962	0.013
KNN	0.970	0.017	0.968	0.017	0.969	0.012	0.968	0.016	0.970	0.018	0.969	0.012

	Total					
Method	**precision**		**recall**		**f1**	
	Avg.	σ	**Avg.**	σ	**Avg.**	σ
RF	0.970	0.017	0.969	0.018	0.969	0.013
XGBoost	0.962	0.018	0.962	0.019	0.962	0.013
KNN	0.969	0.016	0.969	0.017	0.969	0.012

Family	Precision		Recall		F1	
	Average	σ	Average	σ	Average	σ
Orbus	0.982	0.013	1.000	0.002	0.991	0.007
GandCrab	0.964	0.017	0.985	0.007	0.974	0.010
PlugX	0.986	0.012	0.952	0.013	0.969	0.010
TrickBot	0.889	0.036	0.883	0.032	0.886	0.028
Heodo	0.834	0.032	0.890	0.031	0.861	0.025
BazaLoader	0.808	0.032	0.911	0.031	0.856	0.024
Sodinokibi	0.882	0.026	0.831	0.015	0.855	0.015
IcedID	0.868	0.029	0.816	0.033	0.841	0.026
QuakBot	0.891	0.040	0.793	0.036	0.838	0.029
Gozi	0.828	0.054	0.771	0.035	0.797	0.031
AveMariaRAT	0.771	0.065	0.809	0.043	0.788	0.038
Pony	0.931	0.028	0.576	0.038	0.711	0.032
CobaltStrike	0.788	0.048	0.618	0.049	0.691	0.037
Dridex	0.526	0.032	0.938	0.015	0.673	0.026
DiamondFox	0.597	0.049	0.737	0.037	0.658	0.033
Adware.Generic	0.588	0.085	0.663	0.052	0.618	0.048
Amadey	0.710	0.055	0.470	0.038	0.564	0.035
OskiStealer	0.420	0.051	0.693	0.077	0.519	0.037
ModiLoader	0.647	0.054	0.436	0.061	0.518	0.051
njrat	0.454	0.060	0.524	0.067	0.482	0.041
ZeuS	0.478	0.049	0.472	0.047	0.472	0.035
NetWire	0.744	0.104	0.347	0.060	0.467	0.053
BitRAT	0.705	0.059	0.301	0.030	0.420	0.032
QuasarRAT	0.315	0.019	0.607	0.031	0.415	0.021
Stop	0.306	0.025	0.641	0.046	0.413	0.028
AZORult	0.789	0.093	0.280	0.039	0.412	0.048
DanaBot	0.508	0.067	0.348	0.052	0.410	0.049
SnakeKeylogger	0.330	0.037	0.518	0.088	0.400	0.040
FickerStealer	0.425	0.060	0.339	0.060	0.372	0.040
AsyncRAT	0.442	0.069	0.315	0.044	0.366	0.044
HawkEye	0.447	0.076	0.289	0.072	0.341	0.054
MassLogger	0.192	0.010	0.826	0.044	0.311	0.015
Smoke Loader	0.268	0.047	0.359	0.079	0.300	0.041
ArkeiStealer	0.292	0.046	0.276	0.081	0.277	0.051
RaccoonStealer	0.271	0.053	0.251	0.068	0.256	0.052
AgentTesla	0.311	0.083	0.179	0.065	0.224	0.071
RedLineStealer	0.252	0.086	0.181	0.072	0.199	0.057
CryptBot	0.263	0.084	0.150	0.062	0.182	0.055
FormBook	0.318	0.136	0.096	0.055	0.144	0.072
RemcosRAT	0.487	0.166	0.085	0.043	0.142	0.061
Loki	0.331	0.174	0.054	0.034	0.090	0.054
NanoCore	0.153	0.122	0.049	0.046	0.071	0.061
Total	0.571	0.059	0.530	0.046	0.518	0.038

Table 5: D100 - Average outcomes of the multiclass classification and their corresponding standard deviation σ, displayed according to their F1-score in descending order.

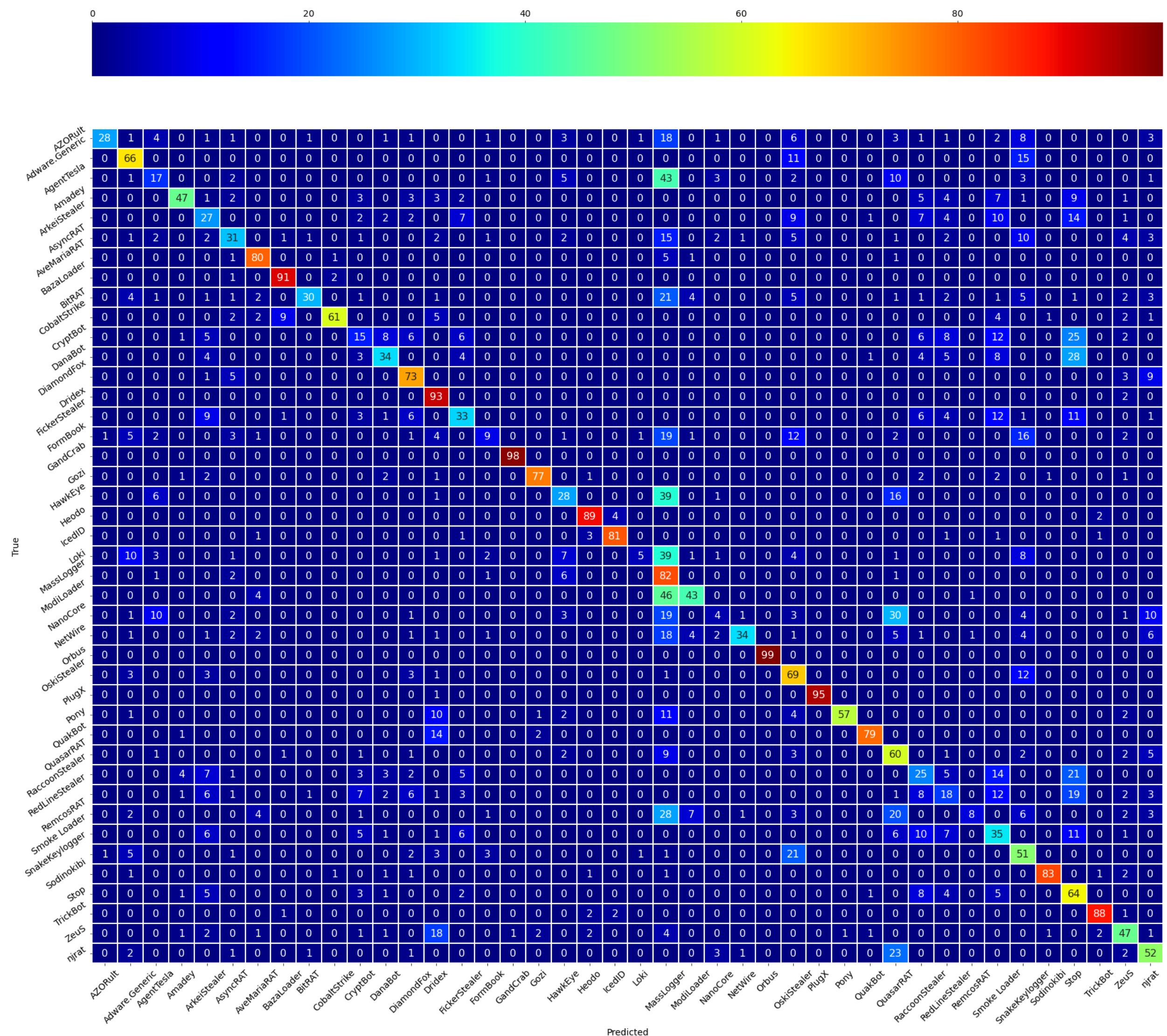

Figure 4: D100 - Confusion matrix of the multiclass classification task.

Group	Families
1	CryptBot, DanaBot, RaccoonStealer, RedLineStealer, Smoke Loader, Sodinokibi, Stop
2	Adware.Generic, AgentTesla, CobaltStrike, HawkEye, Loki, MassLogger, ModiLoader, NanoCore, Pony, QuasarRAT, RemcosRAT, SnakeKeylogger
3	AZORult, AsyncRAT, BitRAT, FormBook, Heodo, NetWire, OskiStealer
4	Amadey
5	BazaLoader
6	AveMariaRAT, QuakBot, TrickBot, njrat
7	IcedID
8	ZeuS
9	Orbus
10	Gozi
11	PlugX
12	DiamondFox
13	GandCrab
14	Dridex
15	ArkeiStealer, FickerStealer

Table 6: Kmeans++ clustering applied to D100 with $k = 25$. A total of 15 groups were created.

Family	Precision		Recall		F1	
	Average	σ	Average	σ	Average	σ
Heodo	0.928	0.012	0.966	0.010	0.947	0.008
TrickBot	0.955	0.013	0.913	0.016	0.934	0.011
QuakBot	0.907	0.020	0.943	0.014	0.925	0.011
BazaLoader	0.811	0.025	0.962	0.009	0.880	0.014
AveMariaRAT	0.808	0.035	0.852	0.017	0.829	0.019
CobaltStrike	0.867	0.030	0.718	0.021	0.785	0.018
Gozi	0.687	0.028	0.814	0.017	0.745	0.017
ModiLoader	0.771	0.031	0.537	0.026	0.633	0.023
njrat	0.552	0.051	0.661	0.047	0.598	0.018
NetWire	0.740	0.078	0.381	0.026	0.501	0.020
DanaBot	0.418	0.015	0.577	0.022	0.484	0.013
ArkeiStealer	0.452	0.028	0.498	0.047	0.472	0.023
AZORult	0.823	0.072	0.294	0.017	0.432	0.016
MassLogger	0.262	0.004	0.916	0.007	0.407	0.006
Smoke Loader	0.396	0.035	0.408	0.041	0.400	0.021
RaccoonStealer	0.392	0.033	0.375	0.038	0.382	0.027
FormBook	0.367	0.028	0.380	0.045	0.372	0.030
RedLineStealer	0.365	0.065	0.262	0.054	0.298	0.033
CryptBot	0.307	0.034	0.278	0.049	0.287	0.023
AgentTesla	0.406	0.043	0.219	0.035	0.284	0.040
NanoCore	0.268	0.024	0.281	0.057	0.271	0.031
Loki	0.432	0.062	0.161	0.029	0.232	0.032
RemcosRAT	0.540	0.118	0.119	0.027	0.192	0.035
Total	0.585	0.038	0.544	0.029	0.534	0.021

Table 7: D300 - Average outcomes of the multiclass classification and their corresponding standard deviation σ, displayed according to their F1-score in descending order.

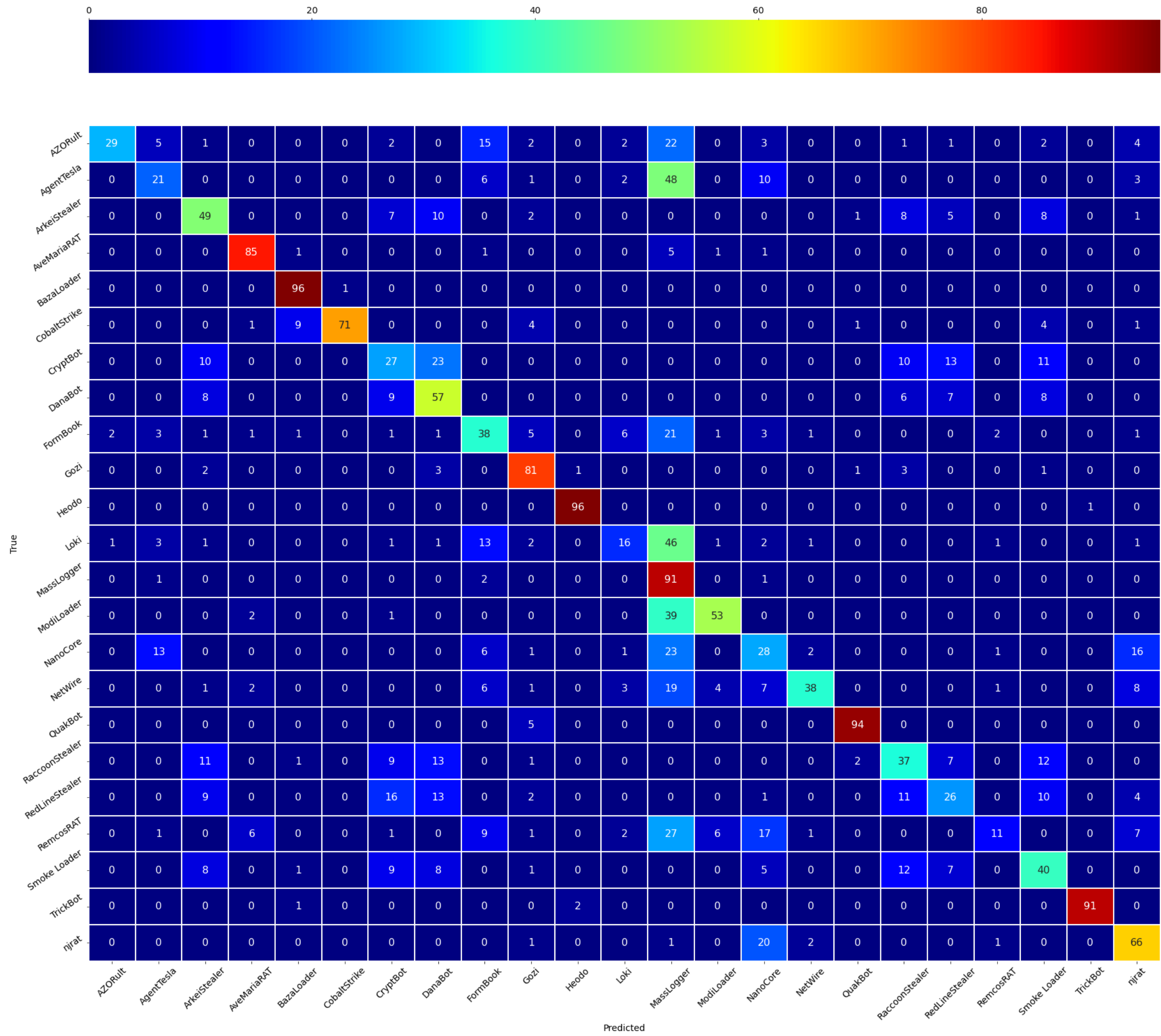

Figure 5: D300 - Confusion matrix of the multiclass classification task.

Group	Families
1	AZORult, AgentTesla, FormBook, Gozi, Loki, NanoCore, NetWire, QuakBot, RemcosRAT
2	ArkeiStealer, BazaLoader, CobaltStrike, CryptBot, DanaBot, ModiLoader, RaccoonStealer, RedLineStealer, Smoke Loader
3	AveMariaRAT
4	njrat
5	MassLogger
6	Heodo
7	TrickBot

Table 8: Kmeans++ clustering applied to D300 with $k = 12$. A total of 7 groups were created.

Ref.	Scope	Method	Dataset	Classification	Results
[26]	W	C&M	3,131M	M(24)	F1 between 0.909 and 0.95
[63]	W	A	120M/150B	B	Accuracy of 0.985
[64]	W	E	1,209M/1,316B	B	Accuracy of 0.954
[28]	W	D	23,080M/114B	B	Accuracy 0.998
[29]	W	C	9,000M	M(9)	TPR, precision, recall and F1 are all >99%, while the FPR is <0.1%
[65]	W	C	7107M	B & M	F1 score of 0.47 in the multiclass setup, and F1 scores between 0.27 and 0.83 in the binary classification according to different malware types.
[66]	W	C	4,960M/1,200B	B	Best Accuracies reported between 0.954 and 0.989.
Our work	W	BE	70,477M/1,059B	B & M(42)	F1 scores between 0.958 and 0.969 in the binary classification, and between 0.518 and 0.534 in the multiclass setup, in all cases outcomes vary according to the sampling strategy.

Table 9: Comparison with state of the art.
Notation: Scope: (A)ndroid, (W)indows. Method: A:https://www.apimonitor.com/ C: Cuckoo, D: Detours [67] M: Maleur, E: Ether [68] BE: Binary Emunlation. Dataset: We report the number of (M)alicious and (B)enign samples that were used. Classification: (B)inary, (M)ulticlass and (#families). The latter classifies samples into worms, trojans, droppers, etc.

Score	Samples$_M$	Samples$_m$
1	4531	7030
2	0	0
3	549	810
4	226	156
5	245	323
6	1640	1773
7	966	1291
8	3126	3560
9	310	202
10	50108	46556
Total		**61701**

Table 10: Distribution of scores of the samples from Triage. Samples$_M$ denotes the number of samples with the highest score specified in the first column. Similarly, Samples$_m$ denotes the number of samples with lowest score specified in the first column.