Classification of Malware Families Based on Runtime Behaviour

Munir Geden(✉) and Jassim Happa

Department of Computer Science, University of Oxford, Oxford, UK
{munir.geden,jassim.happa}@cs.ox.ac.uk

Abstract. This paper distinguishes malware families from a specific category (i.e., ransomware) via dynamic analysis. We collect samples from four ransomware families and use Cuckoo sandbox environment, to observe their runtime behaviour. This study aims to provide new insight into malware family classification by comparing possible runtime features, and application of different extraction and selection techniques on them. As we try many extraction models on *call traces* such as bag-of-words, ngram sequences and wildcard patterns, we also look for other behavioural features such as *files*, *registry* and *mutex* artefacts. While wildcard patterns on call traces are designed to overcome advanced evasion strategies such as the insertion of junk API calls (causing ngram searches to fail), for the models generating too many features, we adapt new feature selection techniques with a classwise fashion to avoid unfair representation of families in the feature set which leads to poor detection performance. To our knowledge, no research paper has applied a classwise approach to the multi-class malware family identification. With a 96.05% correct classification ratio for four families, this study outperforms most studies applying similar techniques.

Keywords: Malware · Dynamic analysis · Feature selection · Security

1 Introduction

According to McAfee's recent report [13], the number of total malware samples has already exceeded 680M, out of which 63M new instances were released in the last quarter of 2017. Despite the widespread use of Anti-Virus (AV) systems for a long time, new malware families and its variants continue to infect computers, smartphones and even small IoT devices. The recent Mirai and WannaCry cases have once again shown how the malware problem can harm our economy at large scales and can stop our critical infrastructures within hours.

There have been many studies trying to solve the malware problem by using different static [5,19,26] and dynamic analyses techniques [6,8,24]. Despite the advantage of full code coverage, static techniques mainly suffer from evasion techniques such as repackaging, obfuscation and polymorphism. To overcome these challenges, dynamic techniques can be used as they focus on actual runtime

A. Castiglione et al. (Eds.): CSS 2018, LNCS 11161, pp. 33–48, 2018.
https://doi.org/10.1007/978-3-030-01689-0_3

behaviour of malware. In this study, we aim to contribute to the state-of-the-art by proposing how improvements of dynamic techniques can in-turn mean protection of millions of more devices. Although the priority of end-users would be the protection of their devices via malware detection, we focus on family classification—as a more challenging task—which can provide more insight for security companies and researchers to understand the recent trends and how families evolve.

Thus, this study distinguishes malware families from the same category by analysing the runtime behaviour with different feature extraction models and selection techniques. Our study makes the following contributions:

1. **Design and implementation of a scalable dynamic analysis framework** that can identify different families from the same category (i.e., ransomware) by using *call traces* and other behavioural artefacts such as *files*, *registry edits*, *mutex names*.
2. **Comparing different feature extraction models** from API and system calls such as *bags-of-words*, *ngrams* or *wildcard* patterns.
3. **Applying new feature selections** and adapting them to a class-wise manner to avoid the domination of a selected feature set by specific families.

2 Related Work

A dynamic analysis technique by Salaehi et al. [17] detects malware by using API calls with their arguments. After the generation of traces via *WINAPIOverride32* tool for 826 malicious and 385 benign samples, the features are selected via document frequencies to create binary vectors for each sample. With 10-fold cross-validation, the authors observed 98.4% accuracy by using Adaboost meta-classifier of Weka. Another similar study by Uppal et al. [23] also uses API calls with ngram models without arguments. The authors use *odds ratio* to select the features. Similar to the previous study, the authors train the classifiers with a 10-fold cross validation resulting in 98.5% accuracy for SVM and 4-grams. Based on their methodology sections, both studies seem to be subject to overfitting bias since the feature selection is applied prior to the cross-validation phase.

Instead of malware detection, MEDUSA [14] classifies metamorphic engines by using the frequencies of API calls. The authors use statistical measures to distinguish metamorphic families by creating signature vectors for each. These vectors are created based on the average frequencies of selected critical API calls for the given family. Furthermore, there are studies [11,15] achieving category and family identification by using API calls in different ways and adding other feature sources such as DNS requests and accessed files to their classifications. For instance, Pirscoveanu et al. [15] distinguish malware types (i.e., trojan, worm, adware, and rootkit) via API sequences, API frequencies and their counter behaviour such as level of DNS requests. Their experiments yield 0.896 True Positive Rate (TPR) for the identification of four categories. Hansen et al. [11] detect malware and distinguishes their families by using similar feature models by including API arguments. The study achieves 0.864 TPR for five families that have different functionalities and components which issues the category bias for the results. Both studies generate the runtime features by *Cuckoo Sandbox* [2] and achieve the best results with Random Forest classifier. Another study by Tsyganok et al. [22] propose a dynamic analysis framework supposed to be resilient against evasion mechanisms such as polymorphic and metamorphic malware. The study uses WinAPI calls and files as features. To decide for the similarity of two samples, they extract *Longest Common Subsequences (LCS)* from the call traces.

Lastly, Canali et al. [7] offer a systematic approach to demonstrate how different feature extraction models—applied on API calls—can influence the accuracy of the detection. They provide a benchmark and elaborate the computational limitations of different models such as bags-of-words, ngram sequences and tuple models which care only about the order of the calls regardless of the distance in between. However, since the authors do not define a threshold distance for their tuple model, its maximum cardinality is very limited compared to other models and performs poorly in some configurations.

Although many studies have tried various feature models to detect malware, there is a lack of a comprehensive approach that assesses the value of different behavioural artefacts and feature extraction/selection models for the classification of malware families from the same category.

3 Problem Definition

Our study aims to demonstrate new and diverse experiment settings in order to answer these four key research questions.

- **RQ1: Can API and system calls be used to differentiate malware families from the same category? (i.e., ransomware)**. We investigate the usability of API and system calls as features to distinguish families from the same category. All samples are picked as ransomware to minimise the behavioural bias of different categories since we expect them to show similar characteristics such as encryption of the files in the system and displaying payment instructions for the ransom.
- **RQ2: Which feature models extracted from API and system calls perform better for family classification and are more resilient against evasion mechanisms such as junk API calls?** By different extraction models on call traces such as bag-of-words, ngrams, wildcard patterns with or without arguments, we explore the drawbacks and advantages of them regarding the accuracy, scalability and resiliency to evasion techniques.
- **RQ3: To what extent other behavioural artefacts can be used to classify malware families?** In addition to API and system calls, we do classifications by using more coarse-grained artefacts such as files accessed, registry keys obtained, mutexes created or Dynamic-Link Libraries (DLL) loaded. Although the call traces can provide the same information through the function arguments, we try to understand the usability of these artefacts without the noise of call traces.

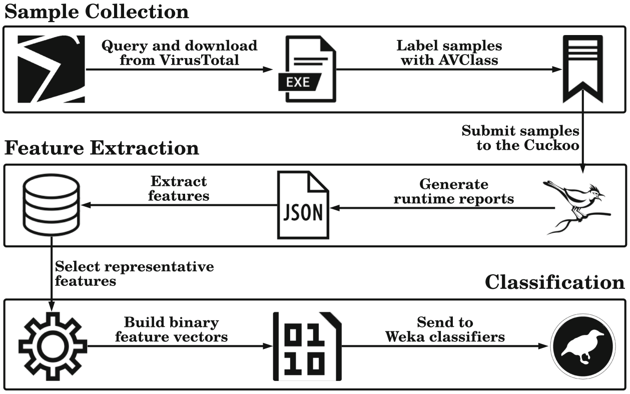


Fig. 1. Design of the experiment framework

– **RQ4: Which configuration settings regarding feature selections and classifier algorithms yield better results?** Since some models generate too many features beyond the limits of classifiers that can handle, we have used different feature selection techniques which is an important factor affecting the accuracy. Moreover, each classifier can perform differently. Thus, we aim to understand optimum settings that provide better results.

4 Methodology

Our experiment framework consists of three stages: *1-Sample Collection*, *2-Feature Extraction* and *3-Classification*, as seen in Fig. 1. We first collect and label samples from different families, followed by the generation of their runtime behaviour in Cuckoo sandbox to extract the features. After applying feature selections on different models, we represent the samples as (binary) feature vectors to be sent to Weka [10] classifiers.

4.1 Dataset Collection

To collect samples, we firstly queried for recent 10–11 well-known ransomware families [1] via VirusTotal API. The queries were designed to return the samples with similar characteristics concerning file type (portable exe), first submission date/year, and file size to avoid any bias issues. However, these queries with family names do not provide reliable evidence about the family belongings since VirusTotal returns any files if at least one of the 82 AV engines labels it with the searched keyword. To address this problem, we used AVClass [20] which extracts the most common family name from AV labels and picked four ransomware families that gave the highest number of samples which were *cerber, crysis, hydracrypt* and *wannacry*.

Table 1. Number of samples for each family in training and test portions

	Cerber	Crysis	HydraCrypt	WannaCry	Total
Training	48	47	24	39	159
Test	24	23	11	19	77

Although the initial queries returned more than a thousand samples, the number of usable samples dropped to hundreds (see Table 1) after filtering with AVClass. In other words: to eliminate any concerns about the soundness of the study, we favoured more meaningful samples and preferred to experiment with the samples that AV engines have a consensus on the family.

Next, we split the samples into training and test portions randomly with a 2:1 ratio as a commonly used ratio in malware domain. We used separate portions instead of cross-validation since its proper application with a feature selection phase requires an extra effort to avoid overfitting.

4.2 Behaviour Generation

After dataset collection, we observed and collected the runtime behaviour of the samples. Due to advantages of VM-based sandbox approach such as ease of collecting many features at the same time or handling process trees, we used *Cuckoo* [2] analysis environment. It enabled us to monitor API calls, files, registry keys and mutexes accessed by the submitted samples. Setting up *Windows XP-SP3* as the experiment environment, we ran samples within for 120 s which generated JSON reports as behavioural logs.

4.3 Feature Models

We used call traces, files accessed, registry keys and mutexes as features while putting special effort on call traces with different extraction models since these calls represent the most valuable runtime features that even most of the evasion techniques including metamorphism cannot avoid.

Call Traces. Windows API and its subset system calls are used in different ways for malware detection (i.e., frequency-based approaches, bag-of-words and ngram models [14,17,23]). This study compares the accuracy and scalability aspect of both API calls and system calls.

The simplest feature model extracted from both API calls and system calls was the bag-of-words model without function arguments whereas bag-of-words with function arguments generated the highest number of features (see Table 2). Furthermore, we extracted ngrams as call sequences for a fine-grained representation of malicious behaviour. Although ngram extraction with function arguments was computationally infeasible with available resources, we successfully extracted 2-grams and 3-grams of both API calls and system calls.

Table 2. Number of unique features in different models

Feature model	# of features
API calls	210
API calls with args	1907098
API calls 2grams	2690
API calls w.card 2calls	6635/45769
API calls 3grams	10823
Sys calls	35
Sys calls with args	958474
Sys calls 2grams	321
Sys calls w.card 2calls	651/1369
Sys calls 3grams	1396
Sys calls w.card 3calls	6748/50653
Files accessed	55581
Dll loaded	166
Registry keys	4785
Mutexes	101

Wildcards models (A/B) represent the features found (A) and permutations generated (B).

To address samples bypassing ngrams by junk calls, our study proposes wildcard-based search models on call traces to catch the malicious activity hidden behind the broken call sequences. To search wildcard strings efficiently, we firstly assigned base-36 ID numbers to each API function (instead of using longer function names). Then, by using these IDs, we generated regular expressions as features that represent possible permutations of functions for the required size (2-calls, 3-calls) with an adjustable distance buffer between two functions. We set the distance buffer as 4 junk calls because larger distances could result in too many false positives while increasing the search cost unnecessarily.

We generated ngrams and wildcard patterns for 2-calls, 3-calls without arguments whereas function arguments are only used by the bag-of-words model. To illustrate all call trace-based models, a short fabricated trace and extractable features from this trace are given in Figs. 2 and 3.

Files. Malware can create new files or read/write the existing ones. We also used file names accessed by the samples during the execution as features to distinguish different families.

Registry Keys. Registry database of Windows systems provides additional artefacts for the executed programs which are frequently used for forensic

```
GetFileType(0)  //Assigned ID:A1
NtClose(0xb0) //Assigned ID:B2
RegCloseKey(0xa4) //Assigned ID:C3
NtTerminateProcess(0,0,1) //Assigned ID:D4
```

Fig. 2. A short example of call trace

```
A1:{GetFileType,NtClose..} //API calls
A2:{GetFileType(0),NtClose(0xb0)..} //API calls with args
A3:{GetFileTypeNtClose,NtCloseRegCloseKey..} //API calls ngram (2)
A4:{A1[0-9A-Z-]{0,12}B2, B2[0-9A-Z-]{0,12}C3..} //API calls w.card (2)
S1:{NtClose, NtTerminateProcess}  //Sys calls
S2:{NtClose(0xb0), NtTerminateProcess(0,0,1)}  //Sys calls with args
S3:{NtCloseNtTerminateProcess}  //Sys calls ngram (2)
S4:{B2[0-9A-Z-]{0,12}D4}  //Sys calls w.card (2)
```

Fig. 3. Feature examples extracted from the short trace (2)

analysis. While a few studies focus on the registry-based malware detection, there are malware [4] in the wild hiding themselves in the registry without causing any file artefacts. Moreover, specific registry operations can indicate hiding attempts from the analysis environments (e.g., checking the existence of VM environment) which we explore by extracting registry keys accessed as features.

Mutexes. Mutexes of operating systems represent the program objects managing the shared resources by different threads. Since these resources can be required by malware as well, we explore the use of created mutex names by the samples to identify malware families [3].

Dynamic-Link Libraries (DLL). Another feature used is the DLL files loaded during the execution of samples. Even though this feature type provides an overview of API calls without any details, it can be useful to understand the value of a more coarse-grained approach without any noise.

4.4 Feature Selection

For the models generating too many features beyond the limits that classifiers can handle such as calls with function arguments and wildcard/ngram models (see Table 2), we eliminated noisy and non-informative features via selection techniques. Despite the use of different techniques in malware domain such as *Document Frequency Threshold* [16], *Fisher Score* and *Chi-Square* scores [18,21], *Information Gain (IG)* [12,17,27] (adapted from text-categorisation) represents the most dominantly used selection technique. Thus, we performed our experiments by applying *Information Gain (IG)* [25], and our novel feature selection technique *Normalised Angular Distance (NAD)*—which is explained below—with their classwise adaptations.

Normalised Angular Distance (NAD). As a new feature selection technique based on our work [9], NAD uses the representation of features in a vector space where each dimension corresponds to the class likelihoods of the features which can be expressed as $P(f|C_i)$ and defined as the proportion of the samples containing the feature f for the given family class C_i.

This method relies on the assumption that feature vectors have equal class likelihoods for each class are not distinguishing and have no value to be selected. As the distinguishing power of a feature increases, the ratio of the difference between class likelihoods should increase as well, which our approach aims to measure via angular distance between the feature vector and the reference vector that has equal class-likelihoods.

After the representation of the features in vector space, by using the Eq. 1, the method firstly calculates α the angular distance between the feature vector $\boldsymbol{f}$ and any reference vector that has equal likelihoods for all classes such as $\boldsymbol{r} = (1, 1, 1, 1)$.

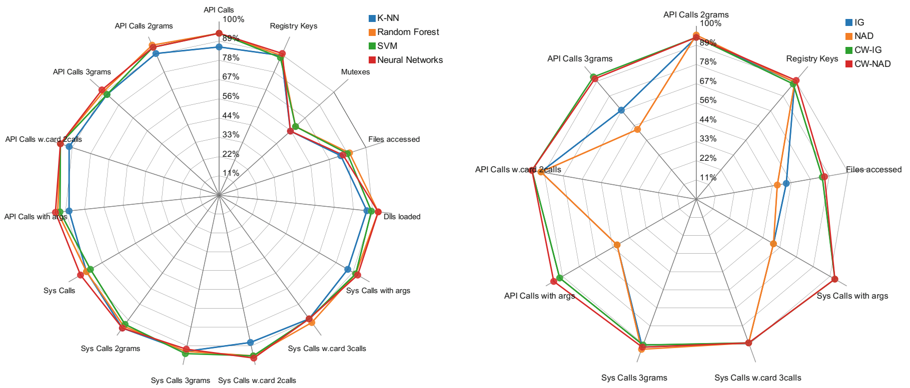
$$\alpha = \cos^{-1} \frac{\boldsymbol{f} \cdot \boldsymbol{r}}{||\boldsymbol{f}|| \cdot ||\boldsymbol{r}||} \tag{1}$$

However, regardless of the vector magnitudes, the angle will be the same for the features that have the same likelihood ratios such as $\boldsymbol{f_1} = (0.1, 0.2, 0.3, 0.4)$ and $\boldsymbol{f_2} = (0.01, 0.02, 0.03, 0.04)$ which can result in the selection of noisy and sparse features. In order to manage this trade-off between being more common and more distinguishing, the method takes into account the magnitude of the feature vector as a normalisation factor with a degree parameter k to adjust the weight of the magnitude for the final score. During our experiments, we set $k = 2$ which could be experimented with a range of $[1.5, 4]$.

$$\mathrm{NAD}(f) = \alpha \times ||\boldsymbol{f}||^{1/k} \tag{2}$$

Classwise Selections. To prevent the domination of features from specific family classes and deliver a fair representation of each class in the selected feature set, we modified *Information Gain* and *Normalised Angular Distance* scores with a class-wise fashion. Although there are studies [16,28] proposing class-wise selection techniques to solve the issue, we offer a more practical solution which is adaptable to any naive solution. Firstly, we create separately ranked lists for each family class as seen in Eq. 3 for NAD which scores only the features with the highest class likelihood for the given class in the list. Then, we build our final feature set with the features ranked in each class list by ensuring that for every n number of features there will be $n/|C|$ features from each family class to create an equally distributed feature set for the classifiers.

$$\mathrm{CWNAD}(f, C) = \begin{cases} \mathrm{NAD}(f), & \text{if } C = argmax_{C_i} P(f|C_i) \\ 0, & \text{otherwise} \end{cases} \tag{3}$$



(a) Classifiers (Classwise NAD used) (b) Feature selections (N. Networks used)

Fig. 4. Weighted TPRs of feature models for different classifiers and selections

4.5 Classifications

After extracting features from Cuckoo reports and selecting features for necessary models, we created binary-feature vectors with a size of 1000 for each sample where the ones represent the existence of a feature for the given sample and zeros represent the absence of that feature. Created feature matrices for each training and test portions are later sent to Weka [10] classifiers. As classifier algorithms, we experimented with *k-Nearest Neighbour* (k=3), *Support Vector Machines* (with SMO functions and poly-kernel), *Random Forests* (no. of trees=100) and *Neural Networks* (default settings of Multilayer Perceptron).

5 Results and Discussion

Although we have defined the correct classification ratio as the key performance metric, we assess the results from the scalability aspect as well. At some points of the discussion, we will be using accuracy and weighted TPR interchangeably. We experimented with 16 different feature models, 4 feature selection techniques and 4 classifier algorithms on a dataset consisting of 236 samples. Due to the excessive number of combinations caused by different settings, we discuss the results with the setting yielding better results on average which is Classwise NAD (selection technique) and Neural Networks (classifier) (see Fig. 4(b) and 4(a)).

5.1 Call Traces

All feature models relying on call traces are promising with the accuracy results varying from 88.16% to 96.05%. Despite the small variations of different classifiers, any model using API and system call traces perform well enough to

Table 3. Accuracy results of different classifiers with Classwise NAD

Feature Model	K-NN	RF	SVM	NN	Average
API calls	85.53%	93.42%	93.42%	93.42%	91.45%
API calls 2grams	89.47%	94.74%	93.42%	93.42%	92.76%
API calls w.card 2calls	90.79%	96.05%	96.05%	96.05%	94.74%
API calls 3grams	86.84%	89.47%	86.84%	90.79%	88.49%
API calls with args	86.84%	93.42%	92.11%	94.74%	91.78%
Sys. calls	88.16%	88.16%	85.53%	92.11%	88.49%
Sys calls 2grams	94.74%	93.42%	92.11%	94.74%	93.75%
Sys calls w.card 2calls	86.84%	94.74%	94.74%	96.05%	93.09%
Sys calls 3grams	92.11%	92.11%	93.42%	90.79%	92.11%
Sys calls w.card 3calls	88.16%	90.79%	88.16%	88.16%	88.82%
Sys calls with args	85.53%	90.79%	90.79%	92.11%	89.81%
DLLs loaded	85.53%	92.11%	88.16%	92.11%	89.48%
Files accessed	73.68%	78.95%	77.63%	75.00%	76.32%
Mutexes	55.26%	59.21%	59.21%	55.26%	57.24%
Registry keys	88.16%	88.16%	86.84%	89.47%	88.16%
Average	87.89%	93.42%	92.37%	93.68%	

distinguish the malware families (RQ1). Regarding the comparison of API and system calls, API calls perform better on average although the system calls have scalability advantages with a less number of features.

Bag-of-Words Model. With 210 API and 35 system functions extracted, the simplest model of call traces is the bag-of-words model not using function arguments. Despite the lack of information such as function arguments and order between the calls, bag-of-words model on API calls has yielded 93.42% accuracy and followed by system calls with a 92.11%.

With Arguments. Feature models using function arguments generate the highest number of features (i.e., 1.9M unique features from API and 958 K from system call traces) which makes them infeasible for ngrams/wildcards (n-calls) due to the number of possible features ($\sim 10^{6n}$). Regarding the accuracy, API calls (94.74%) performs slightly better than system calls (92.11%) with Neural Networks and Classwise NAD settings. One benefit of function arguments is having more insight about the calls whereas standalone function names do not reveal much information about the intention of the called functions.

Another point is that due to the large feature space, these models demonstrate how the feature selection techniques can suffer from unfair representation without classwise adaptations. Since the features extracted have unbalanced distributions (e.g., feature distributions of API calls with arguments can be seen

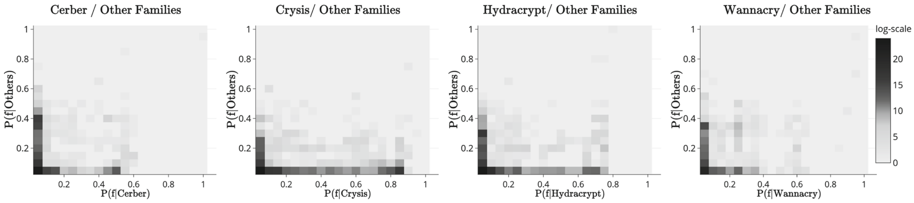


Fig. 5. Extraction distributions of features for *API Calls with args* (1.9M extracted)

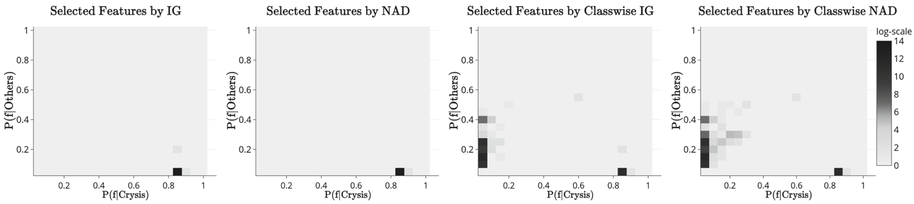


Fig. 6. *Crysis*-based selection distributions for *API Calls with args* (10K selected)

in Fig. 5), naive versions of *Information Gain* and *Normalised Angular Distance* inevitably favour one class for the selected feature sets. Figure 6 illustrates how the selection can differ for naive and classwise techniques on the basis of *Crysis* family. Figure 4(b) shows how the selection bias of naive techniques causes classifiers to perform poorly compared to the classwise selections.

Ngrams. Call sequences are expected to represent the malicious intentions better than the bag-of-words model while this is the case for most models (except API 3-grams). Based on the average results (see Table 3), both 2-grams (93.75%) and 3-grams (92.11%) of system calls have performed better than API 2-grams (92.76%) and 3-grams (88.49%) which conclude that the malicious characteristics are identified better via system calls due to possible existence of noisy and junk calls on API traces. Concerning scalability, the limited number of unique sequences found on traces makes ngrams more scalable than the feature models with arguments (see Table 2).

Wildcard Searches. These models are designed to have resiliency against the evasion mechanisms such as the insertion of junk API calls. The wildcard model running on API calls for 2-calls and with the distance buffer of 4-calls has yielded the best results of all models with an accuracy of 96.05% for three classifiers. Compared to the 2-grams of API calls, it is a legitimate assumption that there are samples in our dataset inserting junk API calls caught by the wildcard model. For system calls of which wildcard models seem to have slightly worse performance than the ngrams correspondents, the results are reasonable because of following issues. Firstly, as malware developers may not prefer to inject specifically junk system call functions, the distance buffer that we set (4-calls) corresponds to much wider distances for system calls due to the elimination of non-system calls in between. Secondly, the application of wildcard models on such a small feature space (35 unique system calls in total) can cause false positives.

In term of scalability, we use hash tables for the storage of features in any model to count and analyse them for the given trace (N) with a $\mathcal{O}(N)$ time complexity. However, for the wildcard models: to check the match of wildcard features for the given call trace, we run regular expressions for each feature which means an additional $\mathcal{O}(N)$ complexity layer that needs be to multiplied by the number of wildcard permutations for one sample trace. Moreover, since we generate all the possible permutations as wildcard features at the beginning, $P(n, r)$ becomes infeasible with $r > 3$ and n for the cardinality of hundreds.

For RQ2, we can conclude that the wildcard model of APIs for 2-calls yields the best accuracy of the experiments, while 2-grams and 3-grams of system calls perform slightly better than the others. Although the wildcard models show resilience against possible junk API calls, the results imply that either the insertion of junk system calls is not practised by malware developers or larger buffer distances (due to the elimination of non-system calls at the beginning) for a small feature space such as system calls cause false positives.

5.2 Other Artefacts

Other feature models relying on *registry keys* and *DLLs loaded* have also produced promising results whereas the *files accessed* and *mutexes* have performed poorly. DLLs and registry keys used during the analyses have yielded 89.45% and 88.16% accuracy on average respectively. The files accessed has produced 76.32% correct classification ratio, while the mutex names represent the worst performing model with a 57.24% (RQ3).

5.3 Optimum Settings and Comparison

As a response to our RQ4, regarding classifier algorithms, Neural Networks and Random Forest have performed quite well with classwise selection techniques. Experiment results manifest that unfair representation of classes by the selected feature sets is an important issue that needs to be addressed by selection techniques. While both classwise adaptations produce significantly more accurate results than the naive ones (see Fig. 4(b)), experiment results demonstrate that our selection technique NAD is able to compete with IG without any significant difference (t-tests are applied). With Neural Networks and Classwise NAD settings, the wildcard model of API calls for 2-calls is the best performing feature model of which detailed class-based performance metrics can be seen in Table 4. Despite the selection of all samples from the same malware category, this study outperforms the majority of similar studies identifying malware categories [15] or families [11] from different categories.

Table 4. Class-based performance metrics and comparison with related work

	TPR	FPR	Prec.	Recall	F-M	ROC
Cerber	0.957	0.019	0.957	0.957	0.957	0.993
Crysis	1	0.038	0.92	1	0.958	0.995
Hydracrypt	0.909	0	1	0.909	0.952	0.996
Wannacry	0.947	0	1	0.947	0.973	0.999
Our study (weighted)	0.961	0.017	0.963	0.961	0.961	0.995
Hansen et al. [11]	0.864	0.035	0.872	N/A	0.864	0.978
Pirscoveanu et al. [15]	0.896	0.049	0.907	N/A	0.898	0.980

5.4 Discussion and Future Work

As this study provides insights into which type of behavioural information can be used for family classification, and to what extent, it also reveals the benefits and drawbacks of different feature extraction models from the same information (e.g., call traces) such as bag-of-words, ngrams, and adjustable and computationally optimised wildcard search models. In addition to the introduction of a new feature selection technique, the study shows how naive selection techniques for high-dimensional models without classwise adaptation causes poor performance. Additionally, this study minimises the potential bias of different categories by selecting all samples from the same category which is an issue that most studies do not address.

While this research focuses on the performance comparison of different feature models and selection techniques, there are other factors defining the success on which we do not have much control. The first issue is various VirusTotal family labels of AV engines. Even though the queries made by family names return too many file hashes, filtering them with a labelling tool [20] reduces the number of usable samples significantly. As we apply supervised-learning techniques and our classifiers use the most common VirusTotal family names as training labels, the performance of the classifiers are still dependent on the way how these AV engines analyse and label these samples.

Our research relies on analysis reports generated by Cuckoo sandbox. Although Cuckoo integrates some mechanisms to avoid the detection of analysis environment by samples and we have eliminated the samples that do not execute, there may be samples hiding the malicious behaviour and acting like legitimate software. Since the study relies on the sandbox approach as a dynamic analysis technique, we are not fully aware of these samples possibly modifying behaviour within the analysis environment. As a future work, we plan to integrate code coverage mechanisms that can measure the confidence of observed behaviour or other heuristics to detect such behaviour modification attempts.

6 Conclusion

In this study, we have implemented a malware family classifier that uses the runtime behaviours collected by Cuckoo Sandbox as features. Although the focus is the exploration of feature models that can be extracted from call traces such as the application of bag-of-words, ngrams or wildcard search models to the API and system call functions, we have also investigated the value of other artefacts such as registry keys, files and mutexes as features. Our experiments have shown that any feature model relying on call traces and DLL libraries and registry keys used during the execution yield promising results which are mostly above 88%. Two other feature models, files accessed (76.32%) and mutexes (57.24%) have produced less accurate results.

In addition to the application of *Information Gain* during our experiments, we have also adapted a new feature selection technique *Normalised Angular Distance*, to family classification as an example of the multi-class classification problem. We have also demonstrated that the adaptation of these feature selection techniques with classwise fashion yields better results since they do not suffer from the unfair representation of specific families.

Acknowledgements. We want to thank VirusTotal community for providing a private API to our research that enabled us to search for and download the ransomware samples.
Cuckoo reports (1.4GB) of the samples and framework's source code: Reports: https://goo.gl/e8jbXq
Source code: https://bitbucket.org/msgeden/familyclassifier

References

1. 11 of the worst ransomware - we name the internet nastiest extortion malware - Gallery - Computerworld UK. https://goo.gl/wNDoL4
2. Cuckoo Sandbox: Automated Malware Analysis. https://cuckoosandbox.org/
3. Hunting the Mutex - Palo Alto Networks Blog. https://researchcenter.paloaltonetworks.com/2014/08/hunting-mutex/
4. TrendLabs Security Intelligence BlogPOWELIKS: Malware Hides In Windows Registry - TrendLabs Security Intelligence Blog. https://goo.gl/3nrgo7
5. Abou-Assaleh, T., Cercone, N., Keselj, V., Sweidan, R.: N-gram-based detection of new malicious code. In: Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004, COMPSAC 2004. vol. 2, pp. 41–42. IEEE (2004). https://doi.org/10.1109/CMPSAC.2004.1342667
6. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A tool for analyzing malware. In: 15th Annual Conference on European Institute for Computer Antivirus Research, pp. 180–192 (2006)
7. Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: A quantitative study of accuracy in system call-based malware detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012, p. 122 (2012). https://doi.org/10.1145/2338965.2336768

8. Fukushima, Y., Sakai, A., Hori, Y., Sakurai, K.: A behavior based malware detection scheme for avoiding false positive. 2010 6th IEEE Workshop on Secure Network Protocols (NPSec), pp. 79–84 (2010)
9. Geden, M.: Ngram and signature based malware detection in android platform. Msc dissertation, University College London (2015). https://goo.gl/uKJsHv
10. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software. ACM SIGKDD Explor. **11**(1), 10–18 (2009). https://doi.org/10.1145/1656274.1656278
11. Hansen, S.S., Larsen, T.M.T., Stevanovic, M., Pedersen, J.M.: An approach for detection and family classification of malware based on behavioral analysis. In: 2016 International Conference on Computing, Networking and Communications, ICNC 2016, pp. 1–5. IEEE (2016). https://doi.org/10.1109/ICCNC.2016.7440587
12. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. J. Mach. Learn. Res. **7**, 2721–2744 (2006). https://doi.org/10.1002/asi.20427
13. McAfee: McAfee Labs Threats Report March (2018). https://goo.gl/ZeugSV
14. Nair, V.P., Jain, H., Golecha, Y.K., Gaur, M.S., Laxmi, V.: MEDUSA: MEtamorphic malware dynamic analysis using signature from API. In: Proceedings of the 3rd International Conference on Security of Information and Networks - SIN 2010 (January), p. 263 (2010). https://doi.org/10.1145/1854099.1854152
15. Pirscoveanu, R., Hansen, S.S., Larsen, T., Stevanovic, M. Pedersen, J., Czech, A.: Analysis of malware behavior: type classification using machine learning. In: International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA), pp. 1–7 (2015). https://doi.org/10.1109/CyberSA.2015.7166128
16. Reddy, D.K.S., Pujari, A.K.: N-gram analysis for computer virus detection. J. Comput. Virol. **2**(3), 231–239 (2006)
17. Salehi, Z., Ghiasi, M., Sami, A.: A miner for malware detection based on API function calls and their arguments. In: The 16th CSI International Symposium on Artificial Intelligence and Signal Processing (AISP 2012), pp. 563–568. IEEE, May 2012. https://doi.org/10.1109/AISP.2012.6313810
18. Sami, A., Yadegari, B., Peiravian, N., Hashemi, S., Hamze, A.: Malware detection based on mining API calls. In: Proceedings of the 2010 ACM Symposium on Applied Computing - SAC 2010, p. 1020 (2010). https://doi.org/10.1145/1774088.1774303
19. Schultz, M., Eskin, E., Zadok, F., Stolfo, S.: Data mining methods for detection of new malicious executables. In: Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001, pp. 38–49. IEEE Computer Society (2001). https://doi.org/10.1109/SECPRI.2001.924286
20. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: AVCLASS: A tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 230–253. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45719-2_11
21. Shabtai, A., Fledel, Y., Elovici, Y.: Automated static code analysis for classifying android applications using machine learning. In: Proceedings - 2010 International Conference on Computational Intelligence and Security, CIS 2010, pp. 329–333 (2010). https://doi.org/10.1109/CIS.2010.77
22. Tsyganok, K., Tumoyan, E., Babenko, L., Anikeev, M.: Classification of polymorphic and metamorphic malware samples based on their behavior. In: Proceedings of the Fifth International Conference on Security of Information and Networks - SIN 2012, pp. 111–116 (2012). https://doi.org/10.1145/2388576.2388591

23. Uppal, D., Sinha, R., Mehra, V., Jain, V.: Malware detection and classification based on extraction of API sequences. In: 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), pp. 2337–2342. IEEE, September 2014. https://doi.org/10.1109/ICACCI.2014.6968547
24. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Secur. Priv. Mag. **5**(2), 32–39 (2007). https://doi.org/10.1109/MSP.2007.45
25. Yang, Y., Pedersen, J.O.: A comparative study on feature selection in text categorization. In: Machine Learning-International Workshop Then Conference, pp. 412–420 (1997). https://doi.org/10.1093/bioinformatics/bth267
26. Ye, Y., Wang, D., Li, T., Ye, D., Jiang, Q.: An intelligent PE-malware detection system based on association mining. J. Comput. Virol. **4**(4), 323–334 (2008). https://doi.org/10.1007/s11416-008-0082-4
27. Yerima, S.Y., Sezer, S., McWilliams, G.: Analysis of Bayesian classification-based approaches for android malware detection. IET Inf. Secur. **8**(1), 25–36 (2014). https://doi.org/10.1049/iet-ifs.2013.0095
28. Zhang, P., Tan, Y.: Class-wise information gain. In: 2013 IEEE Third International Conference on Information Science and Technology (ICIST), pp. 972–978. IEEE, March 2013. https://doi.org/10.1109/ICIST.2013.6747700