Advanced Windows Methods on Malware Detection and Classification

Research Scientists: Dima Rabadi, Teo Sin Gee

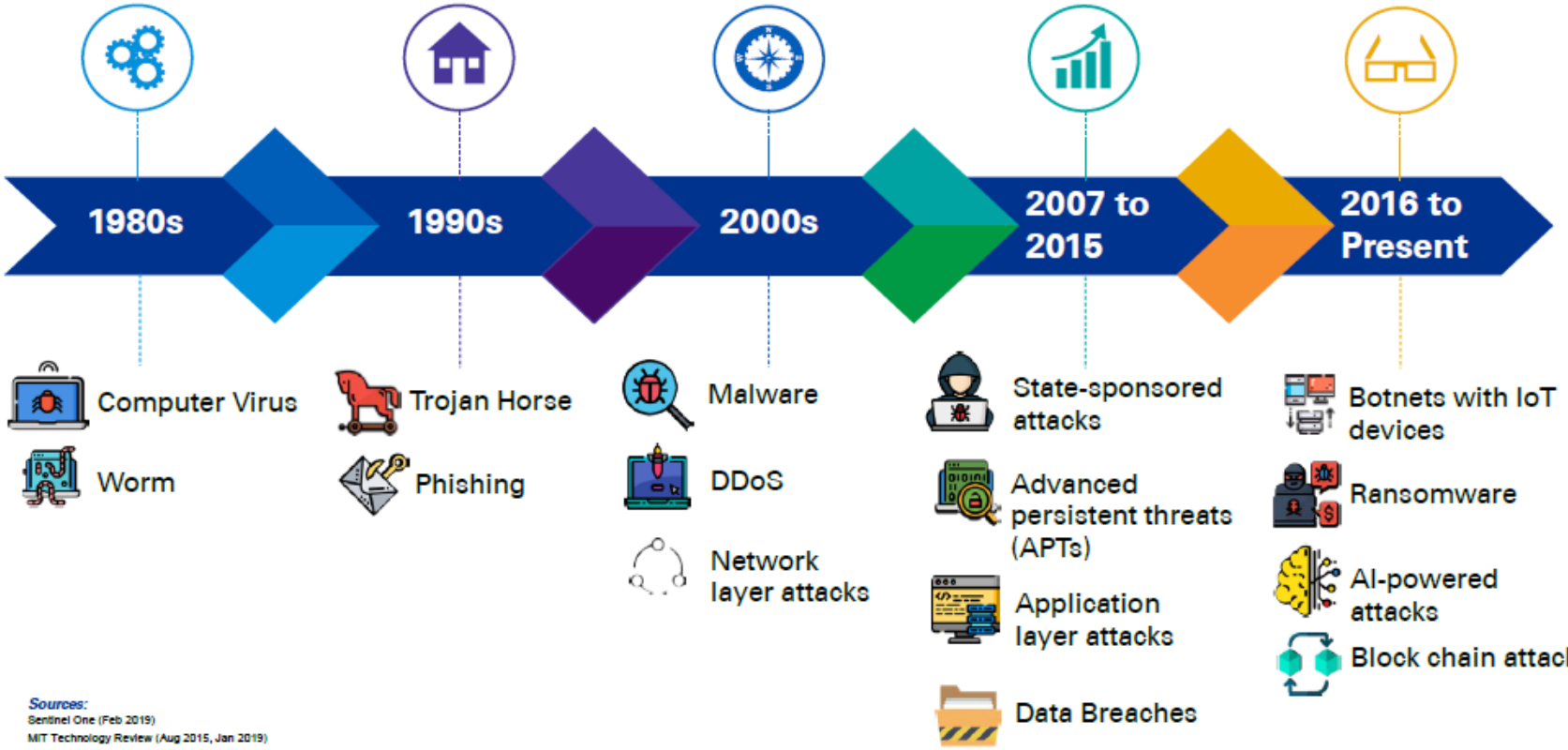
Cyber Security Department, Institute for Infocomm Research (I2R), A*STAR, Singapore

Date: 09-Dec-2020

Annual Computer Security Applications Conference (ACSAC-2020)

A*STAR I²R

The Evolution of Cyber Attacks



Sources:

Sentinel One (Feb 2019)
MIT Technology Review (Aug 2015, Jan 2019)
DarkTrace (2018)
Symantec (Feb, 2019)

CREATING GROWTH, ENHANCING LIVES

Problem we are trying to solve [1]

1. Despite the diversity of the available anti-virus vendors and the significant progress in malware detection research, it has been noticed that malware detection has increased year-over-year by only one percent [1].

2. Application Programming Interfaces (APIs) are still considered the standard accessible data source and core wok of the most widely adopted malware detection and classification techniques.

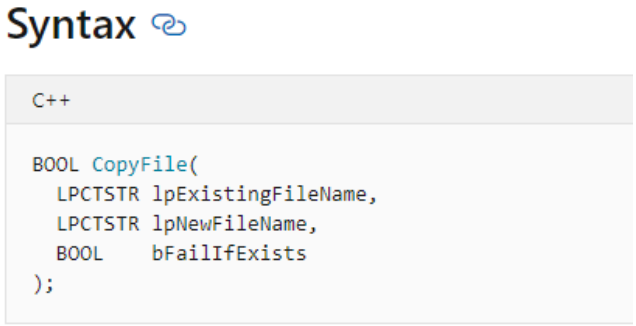
3. Current API-based malware detectors highly rely on:
   a) Measuring API's statistical features, such as calculating the frequency counter of calling specific API calls or
   b) Finding their malicious sequence pattern (i.e., signature-based detectors).

4. Such detector can be easily defeated using simple hooking tools to interrupt the sequence and shuffling the API calls or deleting/inserting the irrelevant calls (i.e., changing the frequency counter).

[1] Malwarebytes Labs. 2020 state of malware report. URL: https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf

Problem we are trying to solve [2]

- API includes two main parts, the **function name** and its **parameters** (arguments):
  - The function name is easily extracted as it is represented by a string that belongs to different categories (i.e., security, system services, and networking).
  - Function parameters are very complex and belong to various types (e.g., integers, strings, and address pointers). Consequently, most of the available AP-based malware detection **approaches do not consider API arguments** in their feature extraction techniques.



Syntax

C++

```
BOOL CopyFile(
  LPCTSTR lpExistingFileName,
  LPCTSTR lpNewFileName,
  BOOL    bFailIfExists
);
```

- Relying on API calls (e.g., function names) alone without taking into account their function parameters is **insufficient** to understand the purpose of the program. For example, the same API call (e.g., write on a file) would act in two different ways if we pass two different arguments (e.g., writing on a system versus user file).

Research Question [1]

- Due to the heterogeneous nature of API arguments, most of the available API-based malicious behavior detectors would consider only the API calls without taking into account their argument information (e.g., function parameters).

- However, as relying on API calls (e.g., function names) alone without taking into account their function parameters is insufficient to understand the purpose of the program. Alternatively**, other malware detector acquire a prohibitive cost**, such as demanding complex operations to deal with the arguments and having proficient knowledge about the API arguments, or **powerful processors to extract them**.

To overcome the above limitations, with the help of machine learning and without any expert knowledge of the arguments, we propose a light-weight API-based dynamic feature extraction technique, and we use it to implement a malware detection and type classification approach.

Research Question [2]

We explore a new low-cost malware detection and type classification approach by **studying the API calls together with their arguments**.

An **important research question** that we seek to answer is whether a low-cost API feature extraction without any domain knowledge of the arguments is sufficient to detect the malware and classify it into its type.

To answer this question, we propose **two methods** using the extracted API arguments:

- **Method 1**: all arguments of each API call are extracted as one feature
- **Method 2**: each argument of each API call is extracted as one feature

To the best of our knowledge, no research has studied the feature extraction technique presented in the second method.

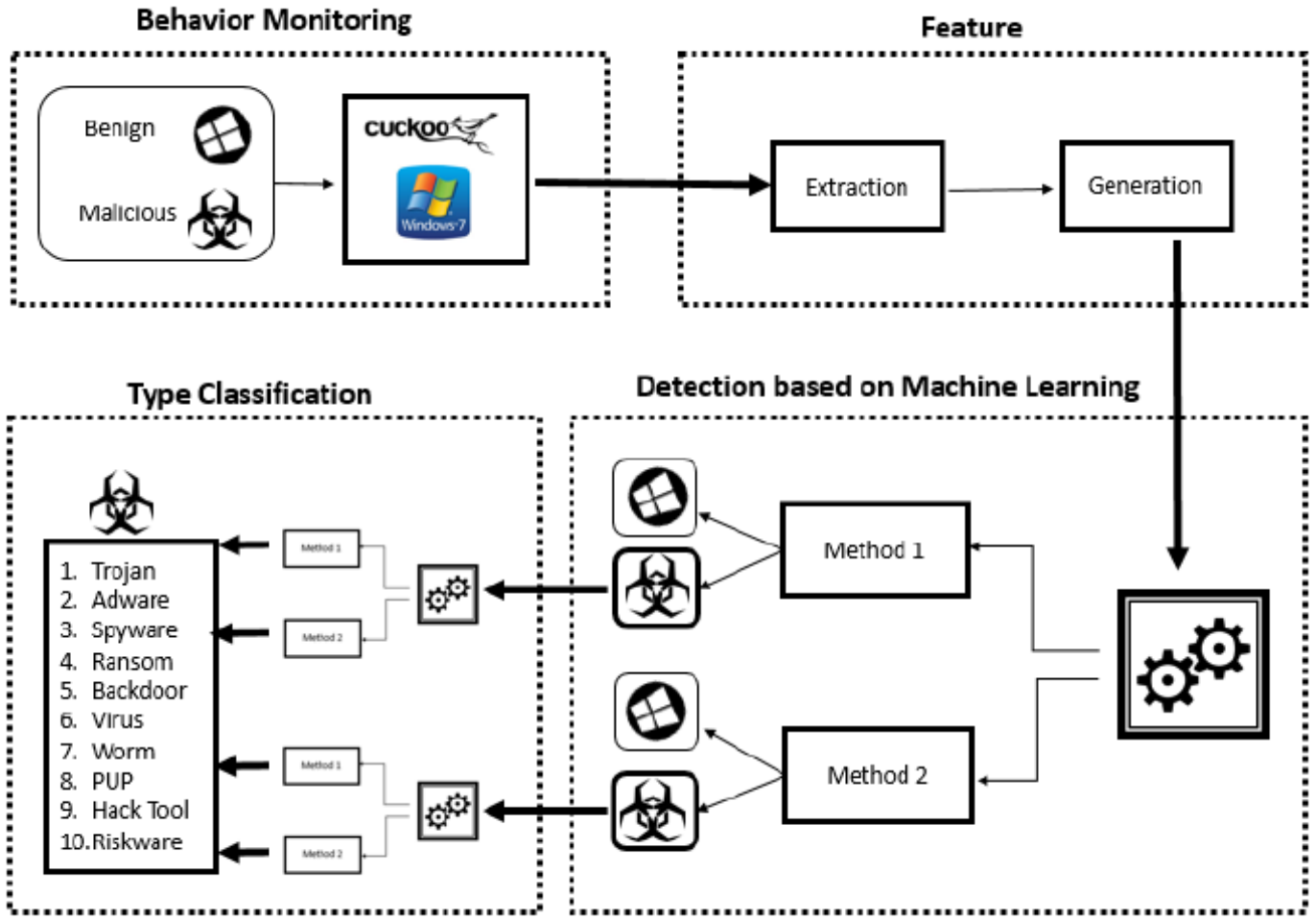
Our solution – Overview

The proposed two methods using the extracted API arguments are used to:

- **First**: Malware detection using machine learning algorithms. This approach distinguish benign from malicious samples
- **Second**: Malware type classification using machine learning algorithms. This approach classify the malware sample into its type.

We detect the malware then classify it into its type

Our solution – Block Diagram



Our solu

automatic processes. The Algorithms in Appendix A give the pseudocode of each of the proposed processes.

(a) API arguments with integer type are converted into their logarithmic bins. If the argument value represents an integer larger than 0, a prefix *numB* plus the logarithmic bin of the value is added to the feature. Furthermore, if the argument value is a negative number, a prefix *neg_* is added before the logarithmic bin in the feature. To distinguish 0 from the number in the range of $(0 - 10)$, 0 is tagged as *num*0 instead of *numB*0. This process is implemented in Algorithm 2. Moreover, arguments with concrete values will not be changed in the feature. For example, *allocation_type* argument in *NtAllocateVirtualMemory* API call has a concrete number equals 8192 that represents a *MEM_RESERVE*. Such constant value will not be changed in this process. This is implemented by Lines 20 to 23 in Algorithm 1.

(b) If the argument defines a directory path (URL), then it is checked whether it contains a System32 in its path to be categorized as *sys_dir*; otherwise, it will be added to the *other_dir* category. This process is implemented in Algorithm 5.

(c) The extension of the file is checked, whether it is one of the most popular Windows file types that are usually used by malware authors [49]. This is implemented in Algorithm 4. Similarly, if the argument is a registry key (e.g., *regkey* such as *BootExecute*, *Winlogon*, *Run*, and *RunOnce keys*, etc.), then its value is checked, whether it is one of the most common registry keys that is exploited by malware. This is implemented in Algorithm 3.

- Feature E
- Hashing V
- To do that

): the goal is to study and uments without the need

ode the generalize it-vector (i.e., Method 1 where the length of each bi

esented in the write-up,

Our Solution - Description (1)

We propose two methods using the extracted API arguments. In the first method, all arguments of each API call are extracted as one feature, whereas in the second method, each argument of each API call is extracted as one feature. To the best of our knowledge, no research has studied the feature extraction technique presented in the second method.

API-based features will be presented as a set of unique strings in the form of an API function name and its argument values. The two methods will prepare the features in two different formats. In the first method, each API call, including its arguments, present one feature. Thus, we have i features, where i is the number of API calls we have. Whereas in the second method, each API call and each element of its arguments are considered as one feature. Thus, we have $\sum_{z=1}^{i} j_z$ features, where j_z defines the number of arguments of API_z, where $z \in (0, i)$. Consequently, Method 1 and Method 2 features are constructed using the following formulas:

- **Method 1**: 'API_0: ARG_0; ARG_1...; ARG_{j_0}', 'API_1: ARG_0; ARG_1...; ARG_{j_1}', ..., 'API_i: ARG_0; ARG_1; ...; ARG_{j_i}'
- **Method 2**: 'API_0: ARG_0', 'API_0: ARG_1', ..., 'API_0: ARG_{j_0}', 'API_1: ARG_0', 'API_1: ARG_1', ..., 'API_1: ARG_{j_1}', ..., 'API_i: ARG_0', 'API_i: ARG_1', ..., 'API_i: ARG_{j_i}'

Methods

Method 1 uses each API call and the entire list of its arguments as one token (feature),
Method 2 uses each argument of the API call as one token separately.

Method1:

- "API(0): arg(0); arg(1)...; arg(j0)", "API(1): arg(0); arg(1)...; arg(j1)", ..., "API(i): arg(0); arg(1); ...; arg(ji)"

Method2:

- "API(0): arg(0)", "API(0): arg(1)", ..., "API(0): arg(j0)", "API(1): arg(0)", "API(1): arg(1)", ..., "API(1): arg(j1)", ..., "API(i): arg(0)", "API(i): arg(1)", ..., "API(i): arg(ji)"

Listing 1: Two API calls and their arguemnts snippet from Cuckoo analysis report.

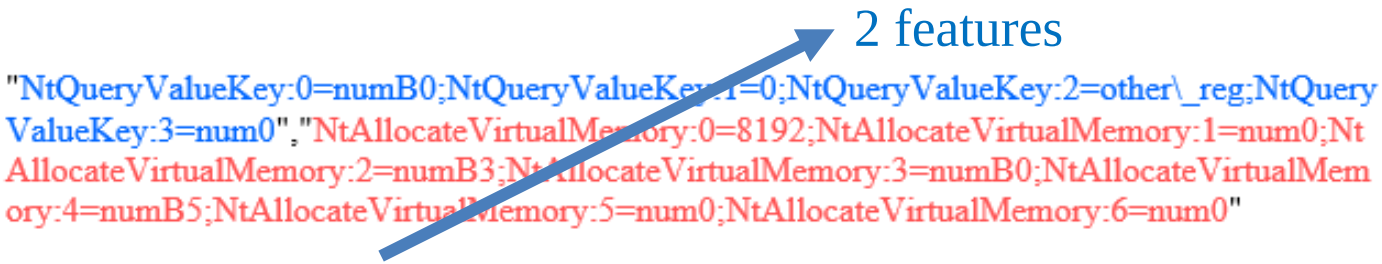
```
\\ report.json
"api": "NtQueryValueKey",
"return_value": 3221225524,
"arguments": {
"value": " ",
"reg_type": 0,
"information_class": 1,
"regkey": "HKEY_LOCAL_MACHINE/SYSTEM/
ControlSet001/Control/NlsExtendedLocale
/en-US"},

"api": "NtAllocateVirtualMemory",
"return_value": 0,
"arguments": {
"process_identifier": 3336,
"region_size": 524288,
"stack_dep_bypass": 0,
"stack_pivoted": 0,
"heap_dep_bypass": 0,
"protection": 4,
"allocation_type": 8192},
```

APIs- two tokenization methods

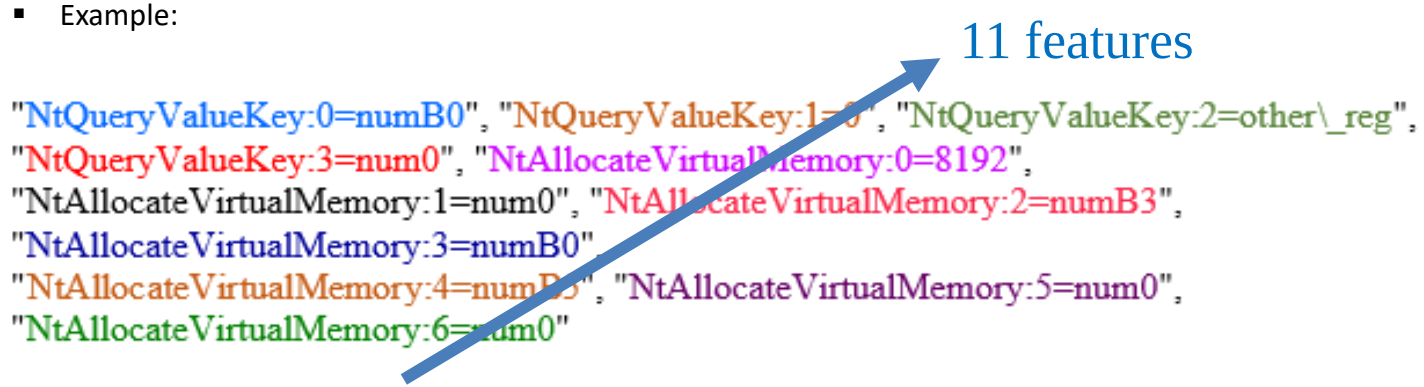
Method 1: each API call together with its arguments

- Example:



Method 2: each API and each of its arguments separately

- Example:



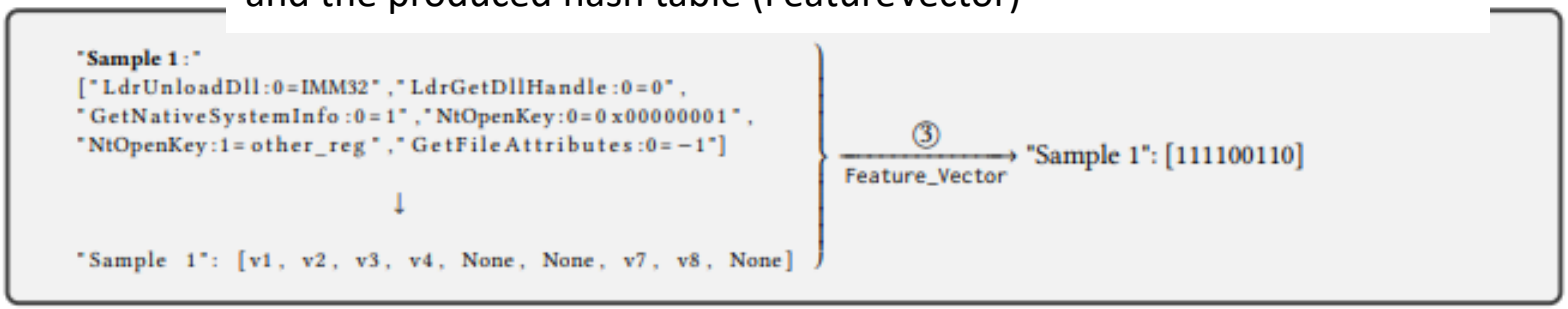
Hashing Vectorizer Function - Example

```
corpus = {
"Sample 1:"
["LdrUnload
"GetNativeS
"NtOpenKey:
"Sample 2:"
["LdrUnload
"NtFreeVirtu
"NtFreeVirtu
"Sample 3:"
["NtTermina
"LdrGetDllH
"NtOpenKey:
"GetFileAtt
}
```

The API calls sequence is not important in our features, as we do not consider the relationship between the API calls as a feature.

Each API alone (does not matter its sequence number in the list of API calls) is considered a feature. Thus, we only examine the occurrence of the API call and the argument.

For example, in Sample1, if the second API call (”LdrGetDllHandle:0=0”) comes before the fist API call (”LdrUnload-Dll:0=IMM32”), that **will not affect** the Hashing Vectorizer function and the produced hash table (FeatureVector)



Our solution – Datasets [1]

Our dataset includes 7105 and 7774, malicious and benign samples, respectively.

The malicious samples are obtained from the Malshare website using a daily downloading script.

The benign samples are obtained from eight different sources:

- After immediately installing a fresh version of Windows 7: we extracted the (1) Native Windows executable, and (2) the Dynamic Link Library (DLL) files which are located in C:\Windows\Systems32 directory.
- (3) We downloaded the APIMDS dataset.
- (4) We used websites for free-to-try legal downloads (e.g., download.cnet.com and softpedia.com).
- We downloaded (5) the top 300 portable applications for Windows7 and (6) the top 43 applications from the File Hippo Website.
- (7) We used the benign dataset from [1], which contains two folders, CYGWIN and (8) WINDOWS 7 benign samples. Both are Windows executable files that are copied from the authors directly

[1] Arzu Gorgulu Kakisim, Mert Nar, Necmettin Carkaci, and Ibrahim Sogukpinar. Analysis and evalu-ation of dynamic feature-based malware detection methods. InInternational Conference on Securityfor Information Technology and Communications, pages 247–258. Springer, 2018

Our solution – Datasets [2]

The number of samples of each benign source and the malware types and the number of samples in each type are presented in the following table:

Sample	Type	No. of samples	%
Malicious	Trojan	5485	36.864
	Adware	295	1.983
	Spyware	381	2.561
	Ransom	441	2.964
	Backdoor	191	1.284
	Virus	99	0.665
	Worm	152	1.022
	PUP	36	0.242
	Hack Tool	14	0.094
	Riskware	11	0.074
Benign	Windows executable	327	2.198
	DLL files	1069	7.185
	APIMDS	200	1.344
	CNET	173	1.163
	Portable applications	300	2.016
	File Hippo	43	0.289
	CYGWIN	4631	31.124
	WINDOWS 7	1031	6.929
Total		14879	-

Our solution - Performance evaluation [1]

– Our methods evaluation **in malicious behavior classification**:

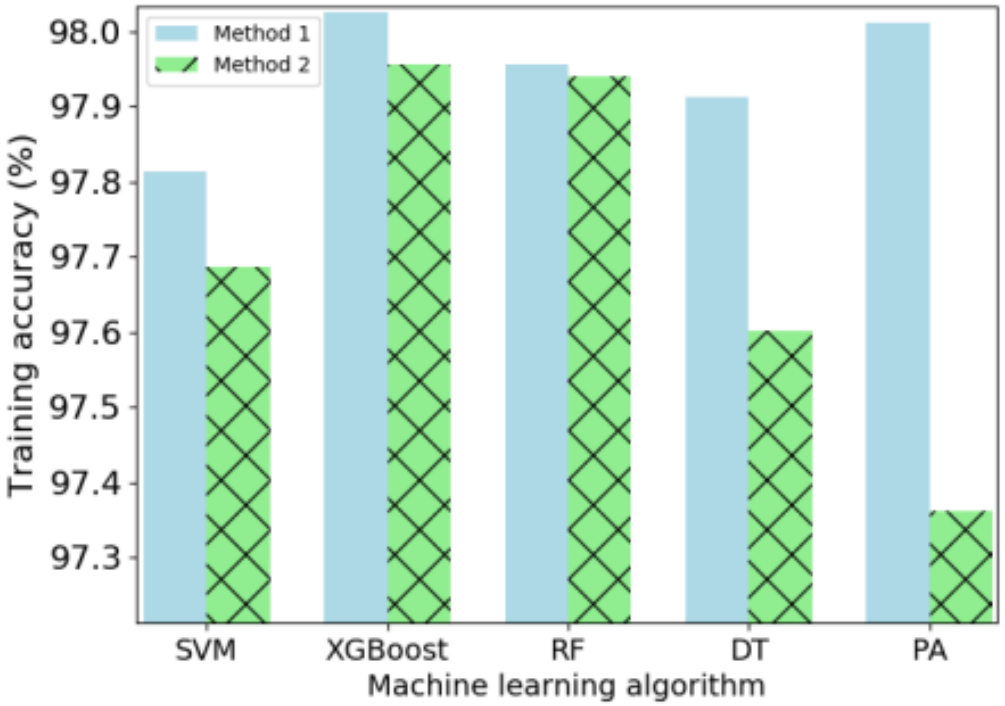
Machine Learning Algorithms	Method	FPR	FNR	Accuracy	5-fold cross-validation	
					Mean Accuracy	(±) STD
SVM	1	0.00192	0.00985	99.4287	96.2	0.079
	2	0.00192	0.00422	99.6976	96.0	0.106
Random Forest	1	0.00836	0.01900	98.6559	95.4	0.078
	2	0.00385	0.00633	99.4959	94.4	0.116
Passive Aggressive	1	0.00257	0.01337	99.2271	96.0	0.088
	2	0.00321	0.00422	99.6303	95.7	0.111
Decision Tree	1	0.00385	0.00492	99.5631	92.1	0.245
	2	0.01221	0.00422	99.1599	95.2	0.133
XGBoost	1	**0.00128**	**0.00140**	**99.8655**	**95.7**	**0.128**
	2	**0.00128**	**0.00070**	**99.8992**	**96.7**	**0.086**

– As XGBoost gives the best performance, we repeated the experiments using 10-fold cross validation

Method	Type	FPR	FNR	Precision	Recall	F1 Score	10 fold cross-validation	
							Mean Accuracy	(±)STD
Method 1	Benign	0.00128	-	99.87	99.87	99.87	99.2	0.024
	Malicious	-	0.00141	99.86	99.86	99.86		
	Total	0.00128	0.00141	99.87	99.87	99.87		
Method 2	Benign	0.00128	-	99.94	99.87	99.90	99.4	0.020
	Malicious	-	0.00070	99.86	99.93	99.89		
	Total	0.00128	0.00070	99.90	99.90	99.90		

Our solution - Performance evaluation [2]

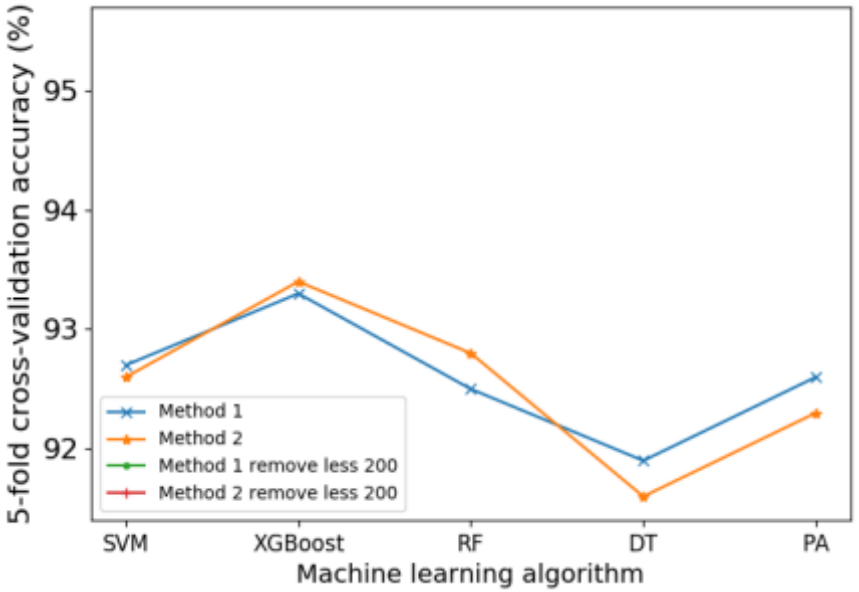
- Our methods evaluation **in** **malware type classification**:



- XGBoost performs as the best malware type classifier out of the five classifiers.
- It gives the best accuracy values:
  - (98.0253 %) for Method 1
  - (97.9548 %) for Method 2

Our solution - Performance evaluation [3]

- Our methods evaluation **in malware type classification**. We **apply 5-fold cross-validation** over the five machine learning algorithms. Results are shown in the following figure.



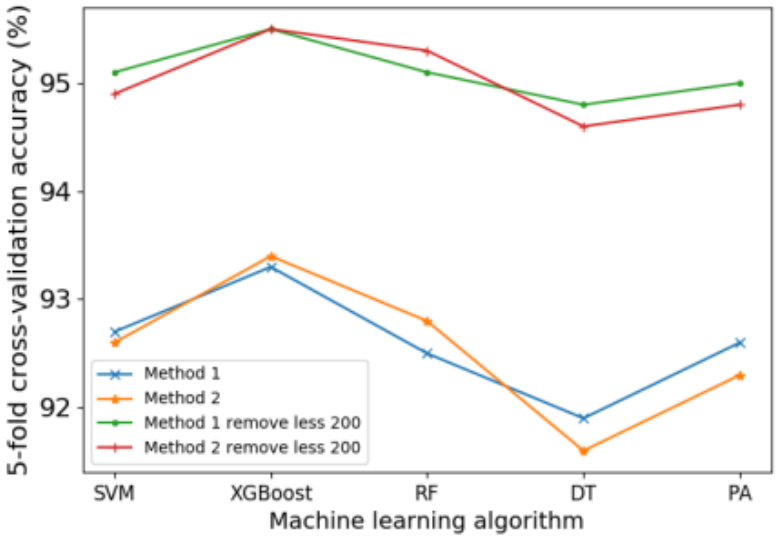
Type	No. of samples
Trojan	5485
Adware	295
Spyware	381
Ransom	441
Backdoor	191
Virus	99
Worm	152
PUP	36
Hack Tool	14
Riskware	11

- As shown in the figure, the accuracy is in the range between [91.9 % - 93.4 %].
- Such decreasing in the accuracy values is due to the small number of samples in some of the types (e.g., Backdoor, Virus, Worm, PUP, Hack Tool, and Riskware).

Our solution - Performance evaluation [4]

- **<u>To overcome this issue</u>**:
  - we remove individual types which have less than 200 samples in our model training.
- In other words, the experiments are done using the samples that belong to only the following four malware types: Trojan, Adware, Spyware, and Ransom.
- Again, we apply the 5-fold cross-validation over the five machine learning algorithms using the four malware types.

- The results are shown in the top half of the Figure.
- We clearly see that the accuracy values **have increased** to the range between **[94.6 % - 95.5 %].**



Comparison with state-of-the-art

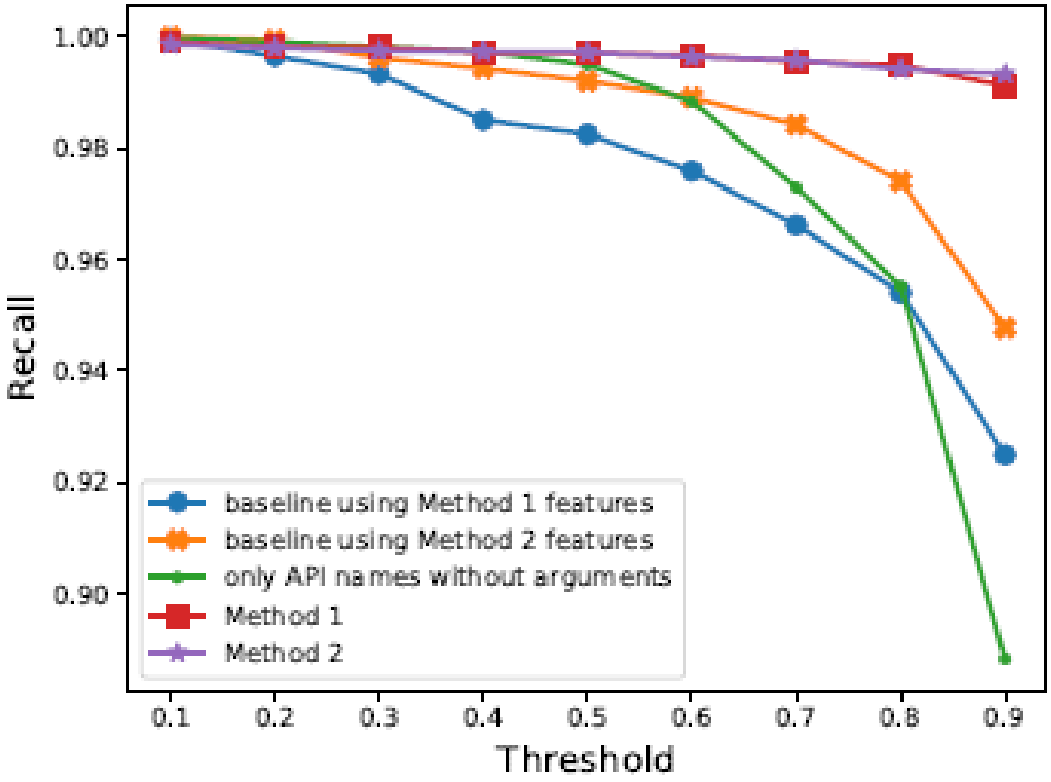
Approach	F1 (%)	Techniques	API Information	Limitations
[7]	97.7	Information retrieval theory, TF-IDF weighting	The sequence of Windows API calls and their arguments	Sequence-based[1]
[35]	97.9	Random forest, J48 decision tree, Sequential Minimal Optimization (SMO)	The sequence of API calls including their arguments and/or return value	A subset of 126 API calls, Sequence-based[1]
[33]	96.7	n-gram	System calls and arguments	System calls only
[41]	97.0	Weka library, SVM, decision table, random forest, Instance-based classifier (IB1)	Frequency counters of API calls and arguments	Frequency-based[2]
[31]	94.7	n-gram	Frequent item-sets of the sequence of API calls with and without their arguments	Frequency-based[2]
[1]	96.6	SMO, RIPPER, NB, J48, IB_k	Statistical features of API calls and their arguments (e.g., frequency, mean and size parameters)	Frequency-based[2]
Method 1 **Method** 2	99.87 99.90	SVM, XGBoost, RF, DT, PA	Method 1: API call and its entire list of arguments, Method 2: API call and each of its argument element separately	Section 6

[1] Sequence-based: malware authors can bypass sequence-based techniques by deleting and/or inserting API calls.
[2] Frequency-based: malware authors can bypass frequency-based techniques by removing and/or adding API calls hence altering the frequency counter value.

References can be found at the write-up

Our solution - Performance evaluation [6]

Comparison of our proposed methods with the baseline work [1], and [2] which uses only API names without arguments, in terms of **recall** scores over different threshold values.



[1] Salehi, Z., Sami, A., and Ghiasi, M.Maar: Robust features to detect malicious activity based on api calls, their arguments and return values. Engineering Applications of Artificial Intelligence 59(2017), 93–102

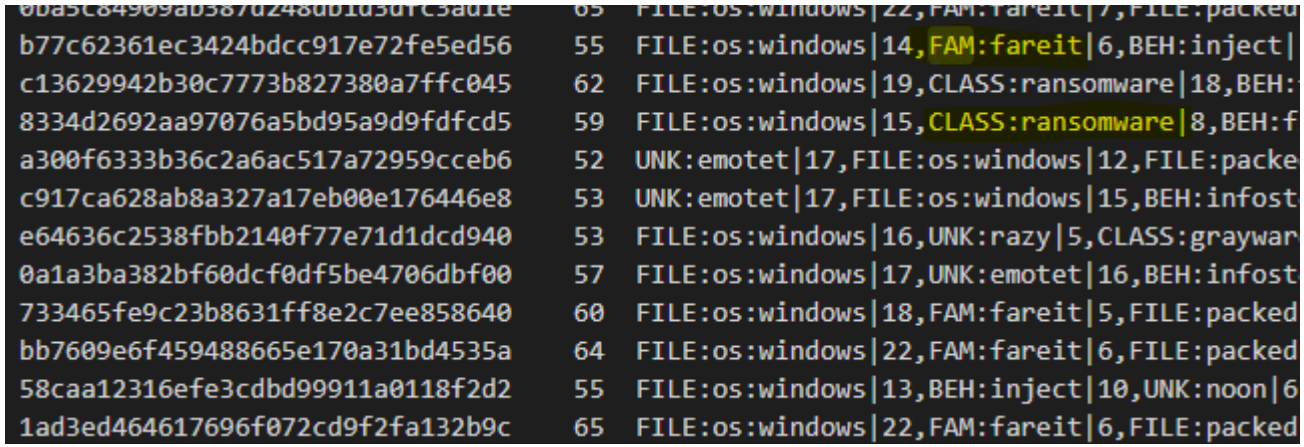
[2] Tian, R., Islam, R., Batten, L., and Versteeg, S. Differentiating malware from cleanware using behavioural analysis.In2010 5th international conference on malicious and un-wanted software(2010), IEEE, pp. 23–30

Limitation and Future Work

- We have only targeted Windows 7. We plan to explore the performance of our proposed methods on other newly released versions of Windows (i.e., Windows 10).
- Furthermore, Android is an API-based operating system; thus, we plan to test the performance of our methods on Android platforms.
- In addition, all experiments are conducted using the Cuckoo sandbox. Therefore, this work is limited to the list of Cuckoo's hooked API calls.
- In the future, we plan to design a run-time malware analysis and detection tool that can extract the API calls simultaneously while the program is running.
- Lastly, we have tested our methods on a dataset of 14,879 samples. To check the performance of our methods on more massive datasets, we are collecting daily and up-to-date malicious samples and generating daily benign samples.

More and More on the Future Work

- Bigger Dataset!
- More variant dataset using more than 10 types
- Deep Learning
- More feature extraction
- We have used only dynamic analysis, we can check our feature performance using Static vs Dynamic analysis
- Family not only type, using AVCLASS



CREATING GROWTH, ENHANCING LIVES

Institute for Infocomm Research

I²R

THANK YOU

www.a-star.edu.sg