Enhancing Malware Classification with Symbolic Features

Chinmay Siwach[1], Gabriele Costa[1] and Rocco De Nicola[1]

[1]IMT School for Advanced Studies, Lucca, Italy

Abstract

Malware is a constant threat for the security of devices and users. Successful and automatic malware detection is a critical necessity [1]. Existing malware detection solutions cannot accurately characterize the behavior of a malware and, thereby, they rely on other indicators, e.g., digital signatures. Nevertheless, behavior-based detection is an active field of research since it can deal with zero-day malware. Although many proposals leveraging machine learning (ML) classifiers have been put forward, finding proper behavioral features is still an open problem. Existing solutions typically consider either static or dynamic software features. Static refers to the program syntax while dynamic refers to features observed at runtime. However, both of them suffer from limitations which impact on the effectiveness of the ML classification.

Here we follow a different approach. We used symbolic execution to model features that denote the malware behavior in a more precise way. To this aim, we introduce a novel feature specification language called Symbolic Feature Specification Language (SFSL). Each rule precisely models a specific malicious behavior that has been documented in past malware samples. Then, we apply local, bounded symbolic exploration to establish whether a binary under analysis matches the defined rules. Eventually, the result of the rule matching process is used to generate vectors of features for a ML classifier.

Our current experiments with different ML classifiers show that this technique can lead to actual improvements of the classification accuracy. Moreover, since behavioral features do not depend on the program syntax, our methodology can even detect threats in new malware samples.

Keywords

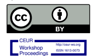
ML based malware detection, symbolic program features, static program analysis

CEUR-WS.org/Vol-2940/paper43.pdf

1. Introduction

Malware is one of the main threats that come along with the diffusion and growth of digital technologies. In the past few years, we have observed a dramatic rise of malware attacks also in the context of cyberespionage and sabotage [1]. Nowadays, signature-based malware detection tools are the main protection mechanism. Nevertheless, since they solely rely on databases of known malware signatures, they can be easily circumvented by slightly changing the malware syntax. Although databases can be kept updated over time, zero-day menaces cannot be faced in this way.

In principle, to effectively detect a malware, we should consider its *behavior* to understand whether it carries our malicious operations. Nevertheless, finding a suitable definition of

ITASEC'21: Italian Conference on CyberSecurity, April 07–09, 2021, all-digital conference
chinmay.siwach@imtlucca.it (C. Siwach); gabriele.costa@imtlucca.it (G. Costa); rocco.denicola@imtlucca.it (R. D. Nicola)


© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

malware behavior is non trivial. An approach followed by most authors is to consider API calls. Under reasonable assumptions, APIs are the only way for a software to interact with the underlying operating systems and its resources, e.g., to read files and open connections. Thus, most of the malicious operations carried out by a piece of malware involve one or more APIs.

There are two ways to inspect how APIs are used by a software. *Static* approaches are based on code analysis and do not require program execution. On the other hand, *dynamic* approaches observes the API invocations performed during one or more executions of the target binary inside a sandbox environment. Nevertheless, these approaches have some serious limitations. As a matter of fact, in most cases static analysis cannot deal with runtime properties related to data and control flows inside the program, e.g., execution branches and invocation arguments. Symmetrically, dynamic analysis can only observe a limited amount of the possible execution traces of a program. The result is that modern malware have plenty of countermeasures to fool automatic classifiers using either static or dynamic features.

Ideally, there is one static technique that can accurately model the actual behavior of software, that is *symbolic execution* [2]. Symbolic execution engines replace runtime values with abstract, symbolic expressions that denote all the possible executions of a certain piece of software. When symbolic execution reaches a certain program state, the generated expressions can be evaluated by a SMT solver to verify the satisfiability of the collected constraints. If the result is negative, the symbolic execution does not correspond to any real execution of the program. Instead, when the the constraints are satisfiable, SMT solvers return a valid assignment to program variables that witness the existence of a concrete execution. The main limitation for the adoption of symbolic execution is its extremely high computational complexity. Although many significant improvements in symbolic execution engines and SMT solvers have been proposed in the past, nowadays the complete symbolic exploration of even small malware samples if out of reach.

In this paper we propose a novel technique that combines symbolic execution and ML-based malware detection. The idea is to take advantage of symbolic execution to extract accurate, meaningful behavioral features from malware samples and to apply state-of-the-art ML classifiers to identify suspicious features profiles. To avoid the complexity blowup of symbolic execution, we recur to local, bounded exploration. In particular, we locally explore limited sequences of program instructions instead of the entire code. In this way, symbolic expressions are kept under control and SMT solvers can be efficiently applied to reasonably small problems. Clearly, this comes at the price of an over-approximation of the actual behavior of the target program. Nevertheless, our features are both (*i*) more accurate than standard, static features, and (*ii*) more general than dynamically generated execution traces.

To obtain this result, we introduce a novel specification for behavioral features called *Symbolic Feature Specification Language* (SFSL). SFSL feature allows to model sort sequences of APIs and relationships between their arguments and return values. Whenever a match is found between a SFSL rule and the target code, a corresponding element of the feature vector is activated. Eventually, the feature vector of a program is passed to a trained ML classifier that decides whether the program should be considered suspicious.

Contributions. The major technical contributions of this paper are the following.

- We describe a novel feature specification language known as SFSL to encode the rule used for malware classification.

- We introduce a novel symbolic feature extraction technique based on matching the rule with the target code in a binary sample.
- We show the effectiveness of our technique by presenting analysis of real-world malware sample in Section 2.
- We implemented our approach by combining symbolic feature extraction technique with machine learning classifiers for malware classification.

Paper organization. In Section 2 we introduce a motivating example that we will develop along the paper. In Section 3 we present our methodology based on symbolic feature extraction. Experimental results are given in Section 4. In Section 5 we present the related work. Section 6 summarizes the final thoughts and ideas for further investigation.

2. Motivating example

In this section we present our motivating example based on malware *Derusbi* [3]. This sample appeared in late 2011. It was involved in several attacks against Japan and USA carried out by Deep Panda (APT-19). This group has targeted security, telecommunications, high tech, and other sectors. The primary goal of Derusbi was to be a long term intelligence gathering tool. To obtain this result, Derusbi is equipped with several functionalities that we list below.

- Backdoor. This sample is a DLL which is capable of registering itself as a service. It also stops security services and opens an interactive command line shell to the Command and Control.
- Spyware. It targets user credentials from client storage in Internet Explorer, Mozilla Firefox and other major browsers. It also does intelligence gathering on the infected system by identifying security tools by their process name, proxy accounts, and version numbers from the Operating System (OS) and Internet Explorer.
- Dropper. The sample is also capable of dropping an encrypted kernel driver. Encrypted kernel driver is responsible for relying information to C2.
- Anti-analysis. If ZhuDongFangYu.exe (AntiVirus360 program) is running, sample don't write encrypted kernel driver to its disk. It also loads a dll known as `pstorec.dll` to detect SunBelt Sandbox.
- Persistence. It can bypass User Account Control (UAC) to achieve persistence by using 'sysprep.exe' (Microsoft executable by Windows) to elevate its privileges.
- Disabling security tools. It can start stop delete system services, managing running processes and enumerating or altering registry keys.

Figure 1 depicts the callgraph of malware Derusbi. Function marked as red contains malicious code listed in Listing 1.

Summarizing, the code below behaves as follows. The first instruction invokes API `OpenSCManagerA` to gain access to the service database. Since the first two arguments are `NULL`, the invocation requests access to the system services. The last argument (`GENERIC_EXECUTE` = `0x20000000`) defines the access privileges. The return value of the function is a pointer to the service

```
1   // Begin at 1000d461
2   service_db = OpenSCManagerA(NULL,NULL,GENERIC_EXECUTE);
3   if (service_db == NULL) break;
4   service = OpenServiceW(service_db,(pointer + 20),0x20);
5   //...
6   if (service != NULL) {
7     err = ControlService(service,SERVICE_CONTROL_STOP,&service_state);
8     break;
9   }
10  // End at 1000d48f
```

Listing 1: Derusbi routine gaining control on Windows security service.

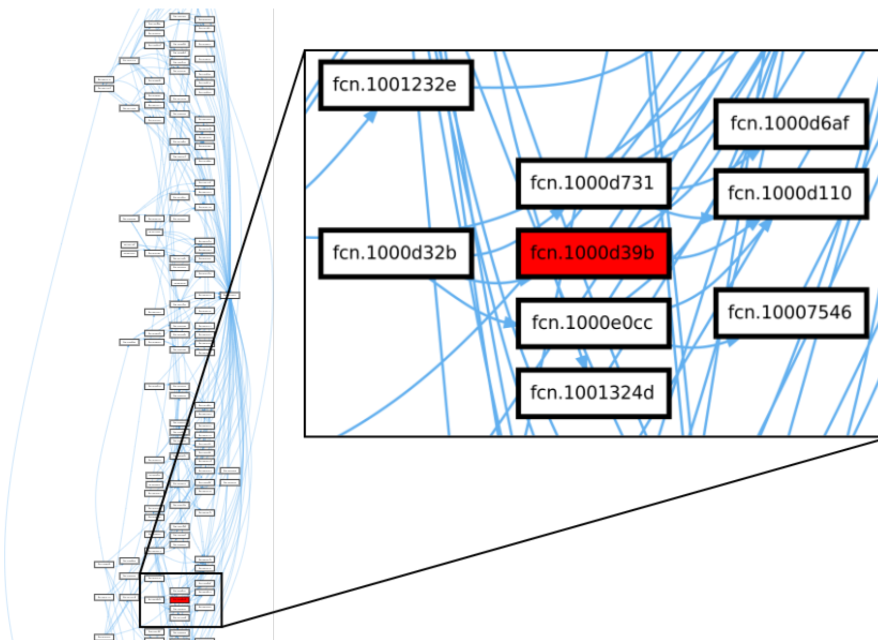


Figure 1: Excerpt of the call graph of Derusbi malware.

database and it is stored in variable `service_db`. If `service_db` is equal to `NULL`, the service database cannot be accessed and execution jumps to another location.[1] Otherwise, instruction at line 4 is invoked `OpenServiceW`. The first argument `service_db` is handle (pointer to service database) from previous call to `OpenSCManagerA` which provides control access to the service control manager (SCM) and service objects [4]. The second argument `pointer + 20`, is obtained through computation that corresponds to service name. The third argument is `0x20` specifying `SC_MANAGER_MODIFY_BOOT_CONFIG` as desired access provides `GENERIC_WRITE` access [4]. The return value is the pointer to the service and it is stored in variable `service`. If `service` is equal to `NULL`, the service cannot be accessed and execution jumps to another location. Otherwise, instruction

[1] For the sake of presentation we replaced jumps to external locations with `break`.

invokes API `ControlService` which sends a control code to a service. The first argument `service` is handle (pointer to a service) from previous call to `OpenServiceW` which provides access to the service. The second argument is `SERVICE_CONTROL_STOP` which corresponds to control code specifying service stop [5]. The third argument `&service_state` is a pointer to a `SERVICE_STATUS` structure that receives the latest service status information.

Although the above fragment only involves a few instructions, static analyzers based on API name inspection cannot effectively understand the malicious nature of the performed operation. The reason is twofold. On the one hand, the chain of involved APIs is standard for most applications, including benign one. As a matter of fact, according to official documentation [6], `OpenSCManagerA`, `OpenServiceW` and `ControlService` are meant to be used in sequence. Furthermore, since there are conditional jumps between the API calls, static approaches may need to approximate the actual instruction flow, e.g., by considering all the possible execution branches instead of a specific sequence. On the other hand, the malicious nature of the code might be spotted out by checking some of the APIs arguments. For instance, one might want to evaluate which service is actually accessed through `OpenServiceW`. This can be achieved by checking whether the second argument belongs to a predefined list of service identifiers. Nevertheless, since the second argument of the call is obtained through a computation, i.e., `pointer + 20`, pure static analysis cannot be applied in general.

3. Methodology

Loosely speaking, our approach amounts to ML-based malware classification, that is, we train a machine learning classifiers to distinguish between malicious and benign software. The main difference with previous proposals is the feature extraction methodology.

3.1. Overview

Our methodology involves three phases that we list below.

- *Phase 1: Features definition.* We implement it by defining a feature specification language called SFSL. A SFSL specification consists of a list of rules, each defining a sequence of security relavent APIs together with a boolean condition. Roughly speaking, this corresponds to asking whether the program under analysis can execute the API sequence so that the rule condition is satisfied.
- *Phase 2: Feature exploration.* We implement a feature extraction engine that performs symbolic exploration of a program to check whether program instructions matches the SFSL rule. The result of this phase is a vector of program features where every location contains 1 if the corresponding SFSL rule is satisfied (0 otherwise).
- *Phase 3: Training machine learning classifier.* The features generated by the extraction algorithm are used for training several machine learning classifiers. Trained classifiers are then used in our experiments for assessing the methodology.

```
SPEC ::= FEAT; ...; FEAT;
FEAT ::= DECL; PATH; BPRE
DECL ::= TYPE VAR, ..., TYPE VAR
TYPE ::= int | char | string
PATH ::= CALL, ..., CALL
CALL ::= VAR := API(VAR, ..., VAR)
BPRE ::= ¬ BPRE | BPRE ∧ BPRE | BPRE ∨ BPRE | true | false | SPRE | APRE
SPRE ::= VAR in SEXP
SEXP ::= [WCHR ... WCHR]
APRE ::= AEXP = AEXP | AEXP > AEXP
AEXP ::= NUM | VAR | AEXP + AEXP | AEXP - AEXP | AEXP × AEXP | AEXP / AEXP
```

Table 1
Feature specification grammar.

3.2. Feature specification language

The abstract syntax of our feature specification language is given in Table 1. A specification (SPEC) amounts to a finite sequence of features (FEAT). Each feature is a triple consisting of some declarations (DECL), a path (PATH) and a Boolean predicate (BPRE). Each declaration is a pair of a type (TYPE) and a variable name (VAR). Supported types include the C base types. For the sake of presentation, here we restrict ourselves to integers, characters and strings. A variable name is any valid identifier. A path is a sequence of API calls (CALL). Each call consists of an assignment of a variable (for the return value) to an API invocation with a list of variables (for the API parameters). Boolean predicates include standard propositional logic connectives as well as string and arithmetic predicates (SPRE and APRE, respectively). A string predicate checks whether a variable matches a certain sequence of characters (WCHR) between [...]. For string matching, we use . as a wildcard to denote that the corresponding position in the string can be any character. For instance x in [a.a] is valid for both x=aaa and x=aba. Instead, arithmetic predicates are comparisons between expressions (AEXP).
For brevity, we feel free to use parentheses for grouping and to introduce some abbreviations. In particular, we use **?** in API calls in place of a variable name when the variable is immaterial, i.e., it does not appear in the Boolean predicate of the feature specification (and we skip the declaration of such a variable). Also, in case the return variable is equal to **?** we simply omit it.

Finally, we use constants in API calls as a shortcut for equality checks as in the following.

$$\text{int x; f(x, 0); x > 2;} \quad \triangleq \quad \text{int x, int y; f(x, y); x > 2} \wedge \text{y = 0;}$$

Example 1. We introduce the following SFSL rule to model the behavior discussed in Section 2.

```
int r, int s, int d, string e, int h, int g;
r := OpenSCManagerA(?,?,?),s := OpenServiceW(d,e,?), ControlService(h,g,?);
r = d ∧ s = h ∧ g = 1 ∧ (e in [WinDefend] ∨ e in [wuauserv] ∨ e in [wscsvs]);
```

The rule above declares 6-variables, i.e., `r`, `s`, `d`, `e`, `h` and `g`. Then, it contains a path consisting of the three API invocations appearing in Listing 1. Finally, the rule concludes with a boolean predicate. The predicate defines a relationship between the invocations by stating

that (*i*) `OpenServiceW` is invoked on the same service database returned by `OpenSCManagerA`, i.e., `r = d`, and (*ii*) `ControlService` is invoked on the same service returned by `OpenServiceW`, i.e., `s = h`. Also, the predicate says that `g = 1`, i.e., the command to be issued to the target service is `SERVICE_CONTROL_STOP` (1). Finally, the rule is matched if the second `OpenServiceW` argument belongs to a list of relevant Windows defense services, i.e., `WinDefend`, `wuauserv` or `wscsvs`.

3.3. Feature extraction algorithm

The feature extraction algorithm aims to compile a vector of features for ML classification. We implemented a bounded symbolic exploration strategy. The algorithm used for symbolic feature extraction is given below 1. Application of algorithm on example mentioned in Section 2 is illustrated Example 2.

Algorithm 1 Symbolic Feature Extraction algorithm

1: **Input** Rule R = (DECL; PATH; PRED) Sample S
2:     **for** `each function f in sample S` **do**
3:         C := set of calls to PATH[0]
4:         **for** `each call c in C` **do**
5:             init := empty_symbolic_state(f)
6:             σ := explore(init,c)
7:             add_bindings (σ,DECL)
8:             target := next(PATH)
9:             **repeat**
10:                 σ := *explore*(σ,target)
11:                 *add_bindings* (σ,DECL)
12:                 target := next(PATH)
13:             **until** end(PATH)
14:             *add_predicate*(σ,PRED)
15:             **if** satisfiable(σ) **then**
16:                 vector(R) :=1
17:                 exit;
18:             **end if**
19:         **end for**
20:     **end for**
21:     vector(R):=0
22: **end Input**

Symbolic feature extraction methodology is based on below mentioned Algorithm 1. Algorithm takes RULE(R) as an input. RULE(R) is a triple of DECL (declaration), PATH (path) and PRED (predicate) mentioned in Subsection 3.2. Symbolic state σ is a set of boolean predicates. Exploration is bounded in nature. Exploration begins by checking the presence of API calls in set C inside a function f of a sample, S and by initializing empty symbolic state. Function *explore* is called with parameters namely, initialized symbolic state σ and call c to PATH[0](first API in a RULE R). Output of the *explore* function is σ symbolic state. Then, function

add_bindings performs binding between resulting symbolic state σ from *explore* and DECL at line 7. Then, we move to next API in a PATH, see Subsection 3.2. This process is repeated until all PATH exhausted(i.e. we reach at last API in a PATH). At line 14, boolean predicates are added by function *add_predicate* that takes σ (symbolic state) and PRED as arguments, where σ here is state obtained after reaching last API in PATH. At the end of exploration of a RULE R, satisfiability check is performed at line 15. If this check amounts to True, one is added to the feature vector and exit is performed to check next RULE R. Otherwise, zero is added to the Vector(R) and we move to the next RULE R.

Example 2. Consider again the fragment of code provided in Section 2. We show the symbolic exploration steps applied to the code fragment and the feature given in Example 1. Exploration starts from a fully symbolic state $\sigma_0 = \emptyset$. The first instruction to be symbolically executed is

```
service_db = OpenSCManagerA(NULL, NULL, GENERIC_EXECUTE)
```

Since no implementation is given of API `OpenSCManagerA`, it is treated as a purely symbolic function, i.e., such that its return value is unconstrained. Thus, after symbolic evaluations, the next state is $\sigma_1 = \sigma_0 = \emptyset$. Since the previous instruction corresponds to an invocation of the first API of the feature under analysis, the next operation is to add to σ_1 new bindings between the feature variables and the state variables. In this case we obtain $\sigma_2 = \{\texttt{service_db} = \texttt{r}\}$. The next instruction is an if statement. Clearly, when the guard is satisfied, i.e., when `service_db == NULL`, the execution jumps to another location and our exploration fails. Thus, we proceed with the new symbolic state $\sigma_3 = \sigma_2 \cup \{\texttt{service_db} \neq 0\}$. The next instruction is

```
service = OpenServiceW(service_db, (pointer+20), 0x20)
```

Again, since this API is the next in the feature under evaluation, the new symbolic state σ_4 must contains bindings between the invocation arguments and the feature variables, that is $\sigma_4 = \sigma_3 \cup \{\texttt{service} = \texttt{s}, \texttt{service_db} = \texttt{d}, \texttt{pointer} + \texttt{20} = \texttt{e}\}$. Then, the symbolic execution proceeds with another conditional statement. In this case, we want the guard to be satisfied, which implies $\sigma_5 = \sigma_4 \cup \{\texttt{service} \neq 0\}$. The next instruction is

```
err = ControlService(service,SERVICE_CONTROL_STOP,&service_state)
```

This API is last in the feature under evaluation, the new symbolic state σ_6 must contain bindings between the feature variables, the invocation arguments, and symbolic expression that is $\sigma_6 = \sigma_5 \cup \{\texttt{service} = \texttt{h}, \texttt{SERVICE_CONTROL_STOP} = \texttt{g}\} \cup \{(\texttt{r} = \texttt{d} \wedge \texttt{s} = \texttt{h} \wedge \texttt{g} = 1 \wedge (\texttt{e in [WinDefend]} \vee \texttt{e in [wuauserv]} \vee \texttt{e in [wscsvc]}))$. After reaching symbolic state σ_6, SMT solver check for which input values of parameters $\texttt{r}, \texttt{s}, \texttt{d}, \texttt{e}, \texttt{h}, \texttt{g}$ can make the symbolic expression hold true (i.e satisfiable). The solver constraints will look like a sequence of assignments `r = d, s = h, g =1, e = WinDefend, service_db,service,pointer`. If SMT solver is able to solve constraints in symbolic state σ_6, presence of a feature is marked. For instance, values that satisfy the assignment are `r=1, d=1, s=2, h=2, g=1, e=WinDefend, service_db=1, service=2, pointer= xxxxxxxxxxxxx xxxxxxxWinDefend`. Therefore, there exists a program executable path corresponding to our feature mentioned in Example 1. End result for this feature check provides us with one in our feature vector.

4. Experimental results

4.1. Dataset preparation

To evaluate the proposed method and create a symbolic feature set, we collected 32-bit Portable Executable (PE) samples for both malware and benignware. In particular, our corpus of malware dataset consists of samples from APT Malware Dataset.[2] The corpus of benignware is made up of samples presented in [7] and Malware Data Science: Attack Detection and Attribution.[3] Filtering out duplicate malware and benign samples is necessary. We used SHA256 hashing to retain unique samples in both malware and benignware groups. Also, we excluded packed binary samples and samples written in VB Script, since these samples are beyond the scope of our research. The resulting dataset consists of 1072 malware and 1112 benignware samples.

4.2. Experimental settings

We run our experiments on a x64 18.04 Ubuntu, Intel®Core™i7-9750H CPU @ 2.60GHz machine, with 16 GB RAM. The feature extraction algorithm has been implemented in Python and we adopted Angr [8] as symbolic execution engine. The set of rules used for our experiments consists of 102 features. Features are collected from various malware reports such as [3] and are encoded in our rules. They are categorized according to MITRE Techniques and Sub-Techniques [9]. We applied a two-minutes time threshold to the extraction of each feature. The ML classifier was implemented with Scikit-learn [10]. In particular, we used well-known machine learning algorithms such as Random Forest (RF), Decision Tree (DT), Naive Bayes (NB), Logistic Regression (LR), and K-NN.

4.3. Stratified K-fold cross validation

The objective of this experiment is to study the performance of the proposed symbolic feature set. Stratified K-Fold splits the dataset on K folds such that each fold contains approximately the same percentage of samples of each target class as the complete set. Stratified K-fold reduces the chance of overfitting. We conducted five separate stratified 4-fold cross-validation experiments. For each of the four partitions, 3 of the folds are used as training data. Validation of the resulting model is done on the remaining data by using it as a test set. Receiver operating characteristic (ROC) curve tracking the relationship between a classifier's detection threshold and its true and false positive rates for classifiers trained and tested on a symbolic feature set is shown in Figure 2 (a,b,c,d,e). Area under curve (AUC) values shown in Figure 2 (a,b,c,d,e) provide an aggregate measure of performance across all possible classification thresholds. It can be observed that RF and LR performance are the best among all classifiers, while NB gives lowest AUC value. Figure 2 (f) shows the box plot output of classifier's accuracy collected during stratified 4-fold cross validation trained on symbolic feature set containing 102 features. It is observed that RF and DT have high accuracy and low variation where others have more variation with low accuracy.

[2]https://github.com/cyber-research/APTMalware
[3]https://www.malwaredatascience.com/

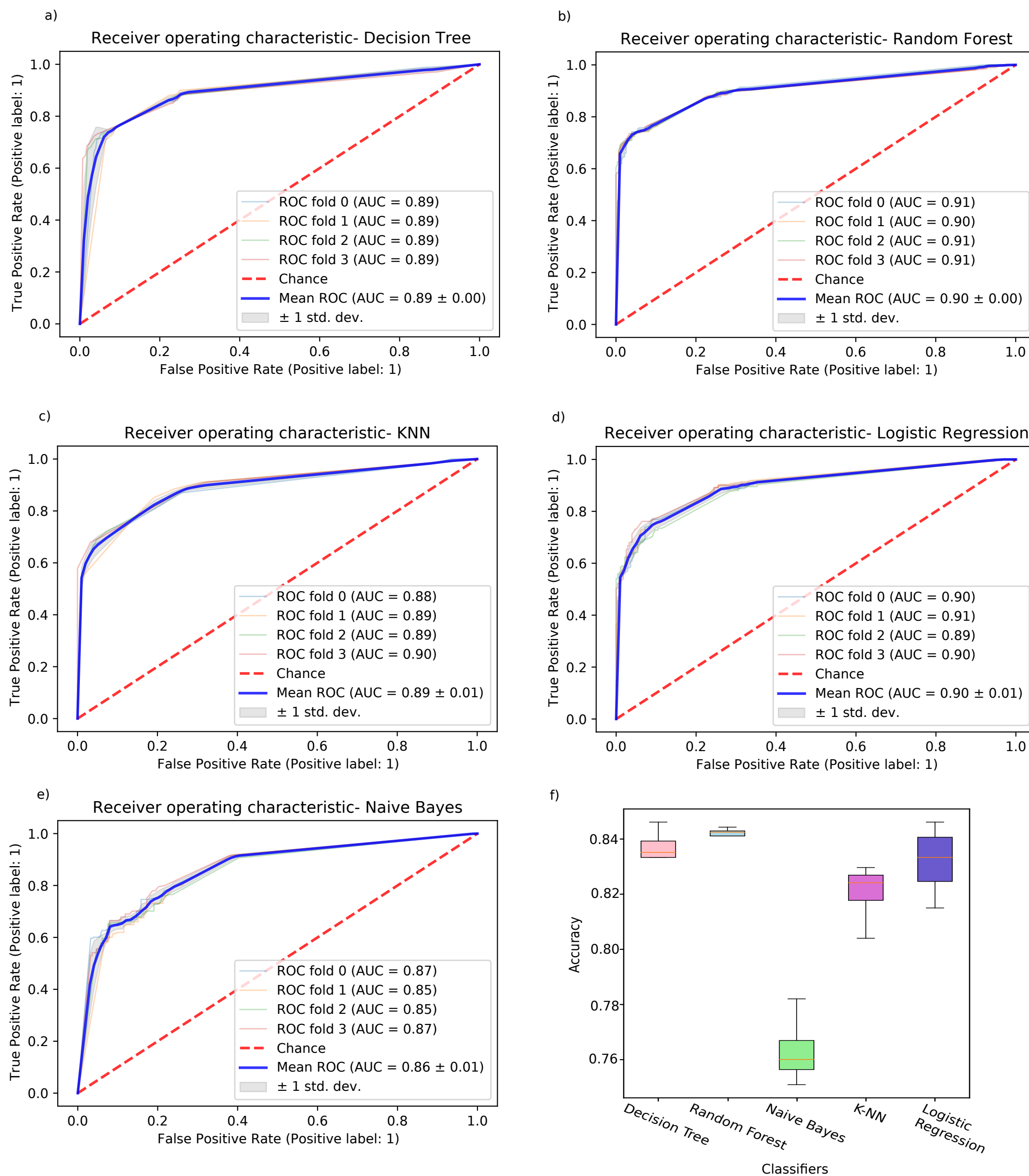


Figure 2: Five experiments (a,b,c,d,e) showing the ROC Curves using 5 different classifiers trained with symbolic features. For each experiment we show the ROC Curve of the individual cross validation folds, and a solid blue line is the averaged value of these ROC curves. Box Plot (f) depicts accuracy of 5 different classifiers with Stratified 4-fold cross validation.

These experiments are only the first attempt to use symbolic features for malware detection and classification. Nevertheless, the results presented above show that malware detection based on symbolic features is feasible. In general, we expect that the number and type of the used rules affects the overall performance of detectors and classifiers implemented in this way. However,

the study of this aspect is beyond the scope of the present work.

5. Related work

In this section, we discuss the related work on malware analysis. We categorize existing works as follows.

Static Analysis. Approaches based on static analysis scan the sample without actually executing it [11]. This technique targets syntactic signatures such as strings or instruction sequences embedded in the binary [12]. As a result, malware samples can evade detection by modifying their appearance while maintaining their functionality [13]. To tackle these limitations, a number of features to describe binary executables have been proposed. One such feature is analysing the specific sequences of bytes also known as n-gram analysis. Here each feature represents the number of times a given combination of n bytes appears in the executable [14]. Another feature relies on API calls to model the the actions of a malware ont the underlying system [15]. The list of all API calls that may be executed is obtained by disassembling the sample. Static analysis based on strings looks for suspicious text inside the binary [15, 16], e.g., known malicious URLs. More sophisticated static analysis approach such as symbolic execution rely on symbolic semantics to capture the malicious functionality of a binary. In [17], the authors show how an analyst can use symbolic execution techniques to unveil critical behavior of a remote access trojan (RAT) by deriving the list of commands and corresponding system calls sequence getting activated. In [18], the author demonstrates the use of system call dependency graphs (SCDGs) constructed using symbolic execution traces. SCDGs are then used as learning inputs in a classifier to generate signature graphs for each malware family.

Dynamic analysis. These techniques capture the behavior of the program at runtime. One method to capture such a behavior is to observe the interactions of the program with the underlying operating system in terms of API calls [19, 20]. In [21], the analysis is based on scrutinizing the genuineness of every kernel driver using flood emulation, i.e., by exploring the program code through forced executions. Approaches such as [22, 23] model program behavior in form of a graph composed of function calls and their dependencies. Dynamic analysis solely relies on extracting behavioral signature encoded in limited set of execution traces. Thus, some execution paths may remain unexplored [24]. Also, several of these techniques proceed with executing the sample in a well-protected and isolated environment, called *sandbox*. However, anti-analysis techniques, such as anti-virtualization, employed by malware samples can detect sandboxed executions [25]. The authors of [26] presented detection of malware based on conditions that trigger hidden behaviour by automatically and iteratively exploring various code paths that may be dependent on trigger inputs. Its effectiveness is demonstrated using examples such as keyloggers that only activate on targeted websites, botnets that wait for the correct command, etc.

Some approaches [24, 26] use hybrid strategies based on a combination of concrete and symbolic execution. This is usually done to benefit from the best of the two techniques, i.e., efficiency of testing and good code coverage of symbolic execution.

Discussion. The state of the art discussed above shows that malware detection based on both static and dynamic features have serious limitations. Mainly, static features struggle in capturing a precise definition of malware behavior, while dynamic features cannot effectively explore the program execution paths. While hybrid techniques can partially overcome these issue, a proper definition of "malicious" behavior is still missing. Our technique cope with this limitation by introducing a dedicated feature specification language.

6. Conclusion

This paper presented a novel symbolic execution-based feature extraction technique that is combined with machine learning to perform malware detection. The main motivation behind the choice of extracting features based on symbolic execution is its ability to provide us with behavioral features without executing the malware. Symbolic execution examines alternative execution paths by exploring the code without running it on a real machine, potentially offering a more comprehensive understanding of malware's actions.

As future work, we plan to continue the assessment of our technique. In particular, we plan to extend corpus of malware samples as well as dataset of rules. Another experiment we want to carry out is for measuring the effectiveness of our methodology against zero-day malware. For instance, this might be estimated by (*i*) only training our classifier with malware released before a certain date and (*ii*) testing the classification accuracy by using more recent malware samples. Finally, we want to consider integration of our SFSL with existing specification languages, such as YARA.

References

[1] Symantec, ISTR V24, 2019. URL: https://docs.broadcom.com/doc/istr-24-2019-en.

[2] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Computing Surveys (CSUR) 51 (2018) 1–39.

[3] Crowdstrike, Report deep panda, 2013. URL: https://paper.seebug.org/papers/APT/APT_CyberCriminal_Campagin/2013/2013.Deep.Panda/crowdstrike-deep-panda-report.pdf.

[4] Windows, Win32API Documentation, 2019. URL: https://docs.microsoft.com/en-us/windows/win32/services.

[5] Windows, Win32API Documentation, 2019. URL: https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-controlservice.

[6] Windows, Win32API Documentation, 2019. URL: https://docs.microsoft.com/en-us/windows/win32/services/service-security-and-access-rights.

[7] A. Kumar, K. Kuppusamy, G. Aghila, A learning model to detect maliciousness of portable executable using integrated feature set, Journal of King Saud University-Computer and Information Sciences 31 (2019) 252–265.

[8] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al., Sok:(state of) the art of war: Offensive techniques in binary analysis, in: 2016 IEEE Symposium on Security and Privacy (SP), IEEE, 2016, pp. 138–157.

[9] M. Corporation, Mitre Attack, 2019. URL: https://attack.mitre.org/.

[10] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.
[11] F. Biondi, T. Given-Wilson, A. Legay, C. Puodzius, J. Quilbeuf, Tutorial: An overview of malware detection and evasion techniques, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2018, pp. 565–586.
[12] P. Szor, The art of computer virus research and defence, addison-wesley professional, New York, NY, USA (2005).
[13] M. Christodorescu, S. Jha, Testing malware detectors, ACM SIGSOFT Software Engineering Notes 29 (2004) 34–44.
[14] J. Z. Kolter, M. A. Maloof, Learning to detect and classify malicious executables in the wild., Journal of Machine Learning Research 7 (2006).
[15] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, G. Giacinto, Novel feature extraction, selection and fusion for effective malware family classification, in: Proceedings of the sixth ACM conference on data and application security and privacy, 2016, pp. 183–194.
[16] J. Saxe, K. Berlin, Deep neural network based malware detection using two dimensional binary program features, in: 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), IEEE, 2015, pp. 11–20.
[17] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, Assisting malware analysis with symbolic execution: A case study, in: International conference on cyber security cryptography and machine learning, Springer, 2017, pp. 171–188.
[18] S. Sebastio, E. Baranov, F. Biondi, O. Decourbe, T. Given-Wilson, A. Legay, C. Puodzius, J. Quilbeuf, Optimizing symbolic execution for malware behavior classification, Computers & Security 93 (2020) 101775.
[19] C. Willems, T. Holz, F. Freiling, Toward automated dynamic malware analysis using cwsandbox, IEEE Security & Privacy 5 (2007) 32–39.
[20] Z. Salehi, A. Sami, M. Ghiasi, 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.
[21] J. Wilhelm, T.-c. Chiueh, A forced sampled execution approach to kernel rootkit identification, in: International Workshop on Recent Advances in Intrusion Detection, Springer, 2007, pp. 219–235.
[22] F. Karbalaie, A. Sami, M. Ahmadi, Semantic malware detection by deploying graph mining, International Journal of Computer Science Issues (IJCSI) 9 (2012) 373.
[23] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, X. Wang, Effective and efficient malware detection at the end host, in: Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09, 2009, p. 351–366.
[24] A. Moser, C. Kruegel, E. Kirda, Exploring multiple execution paths for malware analysis, in: 2007 IEEE Symposium on Security and Privacy (SP'07), IEEE, 2007, pp. 231–245.
[25] R. Wojtczuk, J. Rutkowska, Following the white rabbit: Software attacks against intel vt-d technology, ITL: http://www. invisiblethingslab. com/resources/2011/Software% 20Attacks% 20on% 20Intel% 20VT-d. pdf (2011).

[26] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, H. Yin, Automatically identifying trigger-based behavior in malware, in: Botnet Detection, Springer, 2008, pp. 65–88.