Attacking and Defending Dynamic Analysis System-Calls Based IDS

Ishai Rosenberg[1(✉)] and Ehud Gudes[1,2]

[1] The Open University of Israel, Raanana, Israel
ishai.msc@gmail.com
[2] Ben-Gurion University, Beer-Sheva, Israel

Abstract. Machine-learning augments today's IDS capability to cope with unknown malware. However, if an attacker gains partial knowledge about the IDS's classifier, he can create a modified version of his malware, which can evade detection. In this article we present an IDS based on various classifiers using system calls executed by the inspected code as features. We then present a camouflage algorithm that is used to modify malicious code to be classified as benign, while preserving the code's functionality, for decision tree and random forest classifiers. We also present transformations to the classifier's input, to prevent this camouflage - and a modified camouflage algorithm that overcomes those transformations. Our research shows that it is not enough to provide a decision tree based classifier with a large training set to counter malware. One must also be aware of the possibility that the classifier would be fooled by a camouflage algorithm, and try to counter such an attempt with techniques such as input transformation or training set updates.

Keywords: Malware detection · Malware obfuscation · Decision trees · Behavior analysis · Camouflage algorithm · Machine learning

1 Introduction

Past intrusion detection systems (IDS) generally used two methods of malware detection: (1) Signature-based detection, i.e., searching for known patterns of data within the executable code. A malware, however, can modify itself to prevent a signature match, for example by using encryption. Thus, this method can be used to identify only known malware. (2) Heuristic-based detection is composed of generic signatures, including wild-cards, which can identify a malware family. This method can identify only variants of known malware.

Machine-learning can be used in-order to extend the IDS capabilities to classify software unseen before as malicious or benign by using static or dynamic features. However, our research shows that malware code can be transformed to render machine learning classifiers almost useless, without losing the original functionality of the modified code. We call such a generic transformation, based on the classifier type and the features used, a *camouflage algorithm*.

S. Foresti and J. Lopez (Eds.): WISTP 2016, LNCS 9895, pp. 103–119, 2016.
DOI: 10.1007/978-3-319-45931-8_7

In this paper, we present a camouflage algorithm for decision tree and random forest based classifiers whose input features are sequences of system calls executed by the code at run-time. Our research has three main contributions:

1. Developing an automatic algorithm to decide which system calls to add to a malware code to make this code being classified as benign by our IDS, without losing its functionality. We then alleviate the assumption of full knowledge of the classifier by the attacker, showing that partial training set information might be enough.
2. Evaluating the algorithm against a large subset of malware samples, while previous work evaluated specific examples only.
3. Investigating possible transformations of the IDS input in-order to counter the camouflage algorithm - as-well-as a modified camouflage algorithm to evade those transformations.

While the above contributions are shown for specific classifier types (decision tree and random forest) and for specific features as input (system calls sequences), we believe the ideas are more general, and can be applied also to different classifiers with different features. The rest of the paper is structured as follows: Sect. 2 discusses the related work. Section 3 presents the problem definition and the evaluation criteria for the camouflage algorithm. Section 4 describes the IDS in detail and Sect. 5 discusses the camouflage algorithm implementation. Section 6 presents the experimental evaluation and Sect. 7 concludes the paper and outlines future research.

2 Background and Related Work

2.1 Machine Learning Binary Classifiers

The use of system calls to detect abnormal software behavior was shown in [4,15]. System call pairs (n-grams of size 2) from test traces were compared against those in the normal profile. Any system call pair not present in the normal profile is called a mismatch. If the number of system calls with mismatches within their window in any given time frame exceeded a certain threshold, an intrusion was reported.

Various machine learning classifiers, such as decision trees, SVM, boosted trees, Bayesian Networks and Artificial Neural Networks have been compared to find the most accurate classification algorithm; with varying results (e.g.: [3] chose decision trees, [8] chose boosted decision trees, etc.). The different results were affected, e.g., by the training set and the type of the feature set used.

There are two ways to extract the classifier features. They can be extracted statically (i.e., without running the inspected code), e.g.: byte-sequence (n-gram) in the inspected code [8]. The features can also be extracted dynamically (i.e., by running the inspected code), including: CPU overhead, time to execute, memory and disk consumption [12] or executed system calls sequences, either consecutive [15] or not [14]. A survey of system calls monitors and the attacks against

them was conducted in [5], stating that in-spite of their disadvantages, they are commonly used by IDS machine learning classifiers.

While using static analysis has a performance advantage, it has a main disadvantage: Since the code isn't being run, it might not reveal its "true features". For example, if one looks for byte-sequence (or signatures) in the inspected code [8], one might not be able to catch polymorphic malware, in which those signatures are either encrypted or packed and decrypted only during run-time, by a specific bootstrap code. Other limitations of static analysis and techniques to counter it appear in [11]. Obviously, a malware can still try to hide if some other application (the IDS) is monitoring its features dynamically. However, in the end, in-order to operate its malicious functionality, a malware must reveal its true features during run-time.

Since a dynamic analysis IDS must run the inspected code, it might harm the hosting computer. In-order to prevent that, it's common to run the code in a sandbox; a controlled environment, which isolates between the malicious code to the rest of the system, preventing damage to the latter. This isolation can be done: (1) At the application-level, meaning that the malicious code is running on the same operating system as the rest of the system, but its system calls affect only a quarantined area of the system, e.g., Sandboxie[1]. (2) At the operating system level (e.g., VMWare Workstation), meaning the operating system is isolated but the processor is the same. (3) At the processor level, meaning all machine instruction are generated by an emulator like Qemu (e.g. TTAnalyze). While an emulator-based sand-boxing technique might be harder to detect, it can be done, e.g., using timing attacks due-to the emulator performance degradation, as shown in [13]. Therefore, we have used the VM sandbox mechanism to implement our IDS.

2.2 The Camouflage Algorithms

Modification of the input to a decision-tree classifier based on static analysis features (binary n-grams) was presented in [7]. A simulation of the IDS classifier of the installed anti-virus program was constructed by submitting a collection of malicious and benign binaries to the classifier via a COM (Component Object Model) interface, which runs the installed anti-virus on a file path argument and returns the classifier's decision for this file. Then, a feature-set similar to the attacker' s code that would be classified as benign was found manually in the simulated decision tree. Finally, the authors appended the feature bytes to positions ignored by the system loader in the attacker's code, manually transforming its feature set to the benign one. In contrast, we encountered a dynamic analysis classifier, which is harder to fool [11].

Suggested ways to modify system call sequences were presented in [18]. It deals with *mimicry attacks*, where an attacker is able to code a malicious exploit that mimics the system calls trace of benign code, thus evading detection. [18] presents several methods: (1) Make benign system calls generate malicious

[1] http://www.sandboxie.com/.

behavior by modifying the system calls parameters. This works since most IDSs ignore the system call parameters. (2) Adding semantic *no-ops* - system calls with no effect, or whose effect is irrelevant, e.g.: opening a non-existent file. The authors showed that almost every system call can be no-op-ed and thus the attacker can add any needed no-op system call to achieve a benign system call sequence. (3) Equivalent attacks – Using a different system call sequence to achieve the same (malicious) effect.

In our work, we also use the second technique, since it's the most flexible. Using it, we can add no-op system calls that would modify the decision path of the inspected code in the decision tree, as desired. Main differences: (1) We have created an automatic algorithm and tested it on a large group of malware to verify that it can be applied to any malware, not only specific samples. (2) We verified that the modified malicious code functions properly and evade by executing it after its camouflage. (3) We refer to partial knowledge of the attacker. The authors mentioned several other limitations of their technique in [5] due-to the usage of code injection, which don't apply to our paper. One may claim that the IDS should consider only successful system calls to counter this method. However, every system call in a benign code may return either successfully or not, depending on the system's state and therefore may cause such IDS to falsely classify this code.

A similar method to ours was presented in [10]. The authors used system calls dependence graph (SCDG) with graph edit distance and Jaccard index as clustering parameters of different malware variants and used several SCDG transformations on the malware source code to "move" it to a different cluster. Our approach is different in the following ways: (1) Our classification method is different, and handles cases which are not covered by their clustering mechanism. (2) [10] showed a transformation that can cause similar malware variants to be classified at a different cluster - but not that it can cause a malware to be classified (or clustered) as a benign program, as shown in this paper. (3) Their transformations are limited to certain APIs only - and would not be effective for malware code that doesn't have them.

[16] presented an algorithm for automated mimicry attack on FSA (or overlapping graph) classifier using system call n-grams. However, this algorithm limits the malware code that can be camouflaged using it, to one that can be assembled from benign trace n-grams.

In [1,2], attacker-generated samples were added to the training set of the classifier, in-order for it to subvert the classification of malware code to benign, due-to its similarity to the added samples. However, it requires the attacker to have access to the classifier's DB, which is secured. In contrast, our method, which does not modify the classifier, is more feasible to implement.

3 Problem Description

We deal with two separated issues: (1) Classification of the inspected code, which was not encountered before, by the IDS as benign or malicious, using its system

calls sequences. (2) Developing an algorithm to transform the inspected code, in-order to change the classification of the inspected code by the IDS from malicious to benign, without losing its functionality. The general problem can be defined formally as follows:

Given the traced sequence of system calls as the array *sys_call*, where the cell: *sys_call[i]* is the i-th system call being executed by the inspected code (*sys_call[1]* is the first system call executed by the code).

Define the IDS classifier as:

classify(benign_training_set, malic_training_set, inspected_code_sys_calls),

where *inspected_code_sys_calls* is the inspected code's system calls array, *benign_training_set* is a set of system calls arrays used to train the classifier with a known benign classification and *malic_training_set* is a set of system calls arrays used to train the classifier with a known malicious classification. *classify()* returns the classification of the inspected code: either benign or malicious.

Given that an inspected code generates the array: *malic_inspected_code_sys_calls,* define the camouflage algorithm as a transformation on this array, resulting with the array: *C(malic_inspected_code_sys_calls)*. The success of the camouflage algorithm is defined as follows: Given that:

classify(benign_training_set, malic_training_set, malic_inspected_code_sys_calls)
= malicious, the camouflage algorithm result is:
classify(benign_training_set, malic_training_set, C(malic_inspected_code_sys_calls))
= benign and:
malic_behavior(C(malic_inspected_code_sys_calls))=
malic_behaviour (malic_inspected_code_sys_calls).

While in Sect. 6.3 we would show that partial knowledge of the training set is enough to generate a probabilistic camouflage, we initially assume that the attacker has access to the IDS and to the decision tree model it is based upon. Such knowledge can be gained by reverse engineering the IDS on the attacker's computer, without the need to gain access to the attacked system - just to have access to IDS. As shown in [7], an IDS decision tree can be recovered this way by exploiting public interfaces of an IDS and building the decision tree by feeding it with many samples and examining their classifications. Reconstruction attacks such as the one described in [6] for a C4.5 decision tree could also be used for this purpose. This assumption, that the IDS classifier, can be reconstructed, is common in several papers on this subject (e.g.: [2,5,10,16,18], etc.), as-well-as in cryptography (Kerckhoffs's principle). We further assume the attacker knows the system calls trace that would be produced by the malware on the inspected system. While the system calls trace might be affected by, e.g., files' existence and environment variables' values on the target system, it is highly unlikely, since the IDS should be generic enough to work effectively on all the clients' systems, making system-dependent flows rare.

The effectiveness of our IDS is determined by two factors (P is the probability):

1. We would like to minimize the false negative rate of the IDS, i.e. to minimize P(*classify(benign_training_set, malic_training_set, malic_inspected_code_sys_calls)* = benign).
2. We would like to minimize the false positive rate of the IDS, i.e. to minimize P(*classify(benign_training_set, malic_training_set, benign_inspected_code_sys_calls)* = malicious).

The overall effectiveness of the camouflage algorithm will be measured by the increased number of false negatives, i.e. we would like that:

$$\begin{aligned} &\text{P}(\textit{classify(benign\_training\_set, malic\_training\_set,} \\ &\textit{C(malic\_inspected\_code\_sys\_calls))} = \text{benign}) \geq \\ &\text{P}(\textit{classify(benign\_training\_set, malic\_training\_set,} \\ &\textit{malic\_inspected\_code\_sys\_calls}) = \text{benign}). \end{aligned}$$

Therefore, the effectiveness of the camouflage algorithm is defined as the difference between the two probabilities (which are computed by the respective frequencies). The higher the difference between those frequencies, the more effective is the camouflage algorithm.

One way to fight the camouflage algorithm is to apply transformations on the input sequences of system calls and apply the classifier on the transformed sequences. The assumption is that the transformed sequences would reduce the effectiveness of the camouflage algorithm. We define a transformation of the system calls trace of the inspected code as *T(malic_inspected_code_sys_calls)*. We define the transformation T to be *effective* iff:

1. It would not reduce the malware detection rate, i.e.:
   P(*classify(T(benign_training_set), T(malic_training_set), T(malic_inspected_code_sys_calls))*= malicious) ≥ P(*classify(benign_training_set, malic_training_set, malic_inspected_code_sys_calls)* = malicious)
2. It would not reduce the benign software detection rate, i.e.:
   P(*classify(T(benign_training_set), T(malic_training_set), T(benign_inspected_code_sys_calls))*= benign) ≥ P(*classify(benign_training_set, malic_training_set, benign_inspected_code_sys_calls)* = benign)
3. It would reduce the camouflage algorithm effectiveness:
   P(*classify(benign_training_set, malic_training_set, C(malic_inspected_code_sys_calls))*= benign) ≥ P(*classify(T(benign_training_set), T(malic_training_set), T(C(malic_inspected_code_sys_calls)))*=benign).

In the next two sections we describe in detail the IDS and camouflage algorithm implementations.

4 IDS Implementation

In-order to implement a dynamic analysis IDS that would sandbox the inspected code effects, we have used VMWare Workstation, where changes made by a

malicious code can be reverted. We used a Windows XP[2]. SP3 OS without an internet connection (to prevent the possibility of infecting other machines). The inspected executables were run for a period of 10 s (and then forcefully terminated), which resulted in about 10,000 recorded system calls per executable on average (the maximum number recorded per executable was about 60,000).[3]

The system calls recorder we have used for Windows records the Nt* system-calls. The usage of this low layer of system calls was done in-order to prevent malware from bypassing Win32API (e.g. *CreateFile()*) recording by calling those lower-level, Nt* APIs (e.g. *NtCreateFile()*). We have recorded 445 different system calls, such-as *NtClose()*, etc.

We have implemented the classifier using scikit-learn[4]. We selected the CART decision tree algorithm, similar to C4.5 (J48) decision tree, which was already proven to be a superior algorithm for malware classification [3]).

The training set for the binary classifier contains malicious and benign executables. The malicious executables were taken from VX Heaven[5]. They were selected from the 'Win32 Virus' type. Focusing on this specific mode of action of the malicious code reduce the chance of infection of other computers caused by using, e.g., worm samples. The number of malicious and benign samples in the set was similar (521 malicious samples and 661 benign samples) to prevent a bias towards classification with the same value as the majority of the training samples.

As features for the decision tree we used the position and the type of the system call, e.g.: *sys_call[3] = NtCreateFile*. Thus, the number of available feature values was very large (about 850,000). Therefore, we performed a feature selection of the 10,000 (best) features with the highest values for the χ^2 (chi-square) statistic of the training set, and created the decision tree based only on the selected features. This choice was made to ease the explanation of our algorithm in the next section. In Sect. 6.2 we would use more robust features and show that our algorithm works in this case either.

5 The Camouflage Algorithm Implementation

The goal of the camouflage algorithm is to modify the sequence of system calls of the inspected code in a way that would cause the classifier to change its classification decision from malicious to benign without harming its functionality.

[2] We used Windows XP and not newer versions, in-order to allow computer viri that use exploits found on this OS but patched afterward to run on our IDS either, thus detecting both new and old (but still used) malware.

[3] Tracing only the first seconds of a program execution might not detect certain malware types, like "logic bombs" that commence their malicious behavior only after the program has been running some time. However, this can be mitigated both by classifying the suspension mechanism as malicious or by tracing the code operation throughout the program execution life-time, not just when the program starts.

[4] http://scikit-learn.org/.

[5] http://vxheaven.org/.

This is done by finding a benign decision path (i.e., a path that starts from the tree root and ends in a leaf with benign classification) in the decision tree with the minimal edit distance [9] from the decision path of the malware (or the minimal Levenshtein distance between the paths' string representations). Then we add (not remove or modify, to prevent harming the malware functionality) system calls to change the decision path of the modified malware code to that of the benign path. Selecting the minimal edit distance means less malware code modifications.

In-order to modify the system calls sequence without affecting the code's functionality, we add the required system calls with invalid parameters. This can be done for most system calls with arguments. Others can be called and ignored. For example: opening a (non-existent) file, reading (0 bytes) from a file, closing an (invalid) handle, etc. One may claim that the IDS should consider only successful system calls. However, it is difficult for it to determine whether a system call is invoked with invalid parameters just to fool it, since even system calls of legitimate programs are sometimes being called with arguments that seem to be invalid, e.g., non-exiting registry key. In addition, IDSs that verify the arguments tend to be much slower (4–10 times slower, as mentioned in [17]).

In the basic version of our classifier, an internal node in the decision tree contains a decision condition of the form: *system_call[i]* =? *system_call_type[k]*. Assume without loss of generality that if the answer is yes (i.e., *system_call[i]* = *system_call_type[k]*), the branch is to the right (R child), and if the answer is no, the branch is to the left (L child). An example of a decision tree is presented in Fig. 1. In this decision tree, if the malware code trace contains: {*sys_call[1]=NtQueryInformationFile, sys_call[2]=NtOpenFile, sys_call[3]=NtAddAtom, sys_call[4]=NtWriteFile*} (decision path: *M'=RRRR*, classified as a malicious) and if the algorithm will insert as the fourth system call a system call with a type different than *NtWriteFile*, the classifier will declare this malware code as benign, since the decision path would change from *M'* to *P1*.

While there is no guarantee that the algorithm would converge (step 3 in Algorithm 1 exists in-order to prevent an infinite loop by switching back and forth between the same paths), it did converge successfully for all the tested samples, as shown in Sect. 6. The reason for this is the rationale behind the decision tree based on system calls: The behavior of malware (and thus the system calls sequences used by it) is inherently different from that of benign software. Because-of that, and since the decision tree is trying to reduce the entropy of its nodes, the malicious and benign software do not spread uniformly at the leaf nodes of the decision tree but tend to be clustered at certain areas. Our path modifications direct the decision path to the desired cluster.

The general algorithm is depicted in Algorithm 1. Before explaining the details of the algorithm, let's discuss the possible edit operations when modifying a malware decision path. We will demonstrate the edit operations using the decision tree depicted in Fig. 1:

1. *Substitution*: There can be two types of substitutions: Sub_L - a substitution $L \rightarrow R$ (e.g., from *P=RRRL* to *P'=RRRR*) and Sub_R - a substitution $R \rightarrow L$ (e.g., from *M=RRL* to *P3=LRL* in Fig. 1).
2. *Addition:* Add_R - an addition of *R* (e.g., from *M=RRL* to *P1=RRRL* in Fig. 1) or Add_L - an addition of *L* (e.g., from *P=RRL* to *P'=LRRL*).
3. *Deletion:* Del_L - A deletion of *L* (e.g., from *P=LRL* to *P'=RL*) or Del_R - a deletion of *R* (e.g., from *P=RRL* to *P'=RL*).

Since the only allowed modification is an insertion of a dummy system call, the algorithm handles the above 6 edit operations as follows:

– If the edit_op is Sub_L, *or* Add_R, or Del_L: Given that the condition (in the parent node of the modified\added node) is: *sys_call[i]* =? *sys_call_type[k]*, add *sys_call[i]=sys_call_type[k]*. Note that the equivalent of Del_L is Sub_L followed by a tree re-evaluation, since this is the only edit op allowing you to remove the *L* without actually deleting a system call, which might harm the code's functionality.
– If the edit_op is Sub_R, or Add_L or Del_R: Given that the condition: *sys_call[i]* =? *sys_call_type[k]*, add *sys_call[i]=sys_call_type[m] s.t. m != k*. The above note about deletion applies here too.

After each edit operation, the malware trace changes: The dummy system call addition might have affected every condition on the tree in the form of: *sys_call[j]* =? *sys_call_type[k] s.t. j*$\geq$*i*. Therefore, we need to re-evaluate the entire decision path and find again the benign paths which are closest to it. Step 2(a) exists in-order to minimize the effects of the current edit operation on the path after re-evaluating it. The system calls insertion would ideally be done automatically, e.g., by usage of tools such-as *LLVM*, as done in [10], However, as mentioned by the authors, such tools are currently lack support for dealing with the Windows CRT and Platform SDK API calls, which are used by most Windows malware. Thus we assume that the attacker would manually insert the system calls, added by the camouflage algorithm, to the malware source code. This is demonstrated for the "Beetle" virus, in the next section.

Example 1. We demonstrate Algorithm 1 using the decision tree in Fig. 1:
Given the malware code:
{*sys_call[1]=NtQueryInformationFile, sys_call[2] = NtOpenFile, sys_call[3]=NtWriteFile, sys_call[4]=NtClose*},

Its path in the IDS's decision tree is: *M=RRL* (=Right-Right-Left), and the benign paths in the decision tree are: *P1=RRRL, P2=LLL* and *P3=LRL*, the edit distances are *d(M, P1)=1, d(M, P2)=2, d(M, P3)=1*. The tuples to check are: {*(M, P1), (M, P2), (M, P3)*}. We have two paths with a minimal edit distance: *edit_sequence(M, P1)*={Add_R *(at position 3)*} and *edit_sequence(M, P3)* = {Sub_R*(at position 1)*}. The condition for-which we need to add *R in P1* is: *system_call[3] = NtAddAtom*. Thus: *i=3*. The condition for-which the edit operation applies in *P3* is: *system_call[2] = NtOpenFile*. Thus: *i=2*. Therefore, we start from *P1* and not from *P3*, since its index is larger.

Algorithm 1. System-Calls Based Decision Tree's Camouflage Algorithm

1. Given the decision tree of the IDS and a specific malware trace (i.e. its sequence of system calls as recorded) with the decision tree's path M, find all the IDS's decision tree's benign paths, *P1..Pm,* and create a list l of m tuples to check: *l={(M, P1)..(M, Pm)}*. Set *path_count[M] = 0*
2. For each tuple *(dec_path, Pj)* in l, find the minimum edit distance between *dec_path* and *Pj*, *d(dec_path, Pj)*. Select the tuple with the minimal such edit distance and find the minimal sequence of edit operations needed to change *dec_path* to *Pj*, ordered from the root of the tree to the leaf\classification node (i.e. by position in the decision path). If l is empty: Report failure.
   (a) If there is more than a single path with the same minimal edit distance, look at the first edit operation in each such path. Assuming the condition is of the form: *system_call[i] =? system_call_type[k],* select the path that maximizes *i.*
3. Set *path_count[des_path] += 1.* If *path_count[des_path]≥max_decision_path_count* : Remove all tuples that contain *dec_path* from l and go to step 2.
4. Assuming the benign path to fit is *Pj*, modify the malware code based on the first edit operation in the edit sequence, as was explained above:
   (a) If the edit_op is Sub_L, Add_R, or Del_L then: Add *sys_call[i]=sys_call_type[k].* Else: Add *sys_call[i]=sys_call_type[m] s.t. m != k.*
5. *system_call[i..n]* from before the modification now become *system_call[i+1..n+1].* Re-evaluate the new system calls sequence and generate a new decision path *M'.*
6. If M' ends with a benign leaf: Report success. Else: Remove *(dec_path, Pj) from l,* and add all the tuples with the modified malware code *{(M', P1)..(M', Pm)}* to l. Set *path_count[M'] = 0*
7. Go to step 2.

In-order to modify *M* to *P1*, we add: *sys_call[3] = NtAddAtom(NULL, 0, NULL)* (the edit_op is Add_R). Notice that we add the system call with invalid parameters. The new malware code is:

{sys_call[1]=NtQueryInformationFile, sys_call[2] = NtOpenFile, sys_call[3]= NtAddAtom, sys_call[4]= NtWriteFile, sys_call[5]=NtClose}.

Its decision path is *M'=RRRR. M'* is not classified as benign – so we remove *(M, P1)*, and add all the tuples with the modified code *M'*. Thus, we need to examine:

{{M, P2), (M, P3), (M', P1), (M', P2), (M', P3)}.

The tuple we would inspect in the next iteration is *(M', P1): d(M', P1)=1* and *i=4* (which is larger than *2* for *(M, P3)*). The algorithm would converge after the next iteration, in which we would add *sys_call[4]!= NtWriteFile*, and the modified malware code would be classified as benign (*P1*).

5.1 Random Forest Camouflage Algorithm

In Sect. 6.2, the classifier with the best performance was random forest. Since a random forest is actually a collection of decision trees, if we extend the same

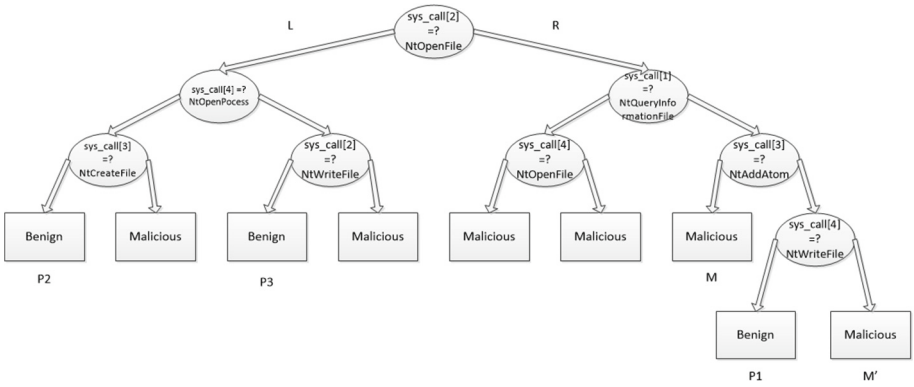

Fig. 1. A system calls based decision tree

assumptions made in Sect. 3, that-is: we know all the trees in the random forest, we can create a camouflage algorithm for random forest.

The rationale of the algorithm is simple: Since all decision trees in the random forest actually represents parts of the same code flow, we can modify each of them in turn, using Algorithm 1, and keep using the modified system calls trace, until we can fool the majority of them, thus fooling the entire random forest.

6 Experimental Evaluation

In-order to test the detection rate of our IDS, we used benign files collection from the Program Files folder of Windows XP SP3 and from our collection of third party benign programs and malware of Win32 Virus type, from VX Heaven's collection. The test set contained about 650 benign programs and 500 malware, which were different from the ones used to train the IDS in Sect. 4. The malware detection rate and the benign detection rate (as computed by the definitions specified in Sect. 3), were 84.3 % and 88.9 % respectively, as shown in the first line of Table 2.

In-order to test our camouflage algorithm, we have selected all the malware samples from our test set, which were correctly classified (i.e., as malicious) by our IDS (436 samples). We applied the camouflage algorithm on them: *None* of the camouflaged system calls sequences of those samples were identified by our IDS (effectiveness of 100 %, by the definition in Sect. 3).

We have applied the random forest camouflage algorithm on all the malware code that were detected by the random forest: 445 different samples. While there is no guarantee that the algorithm would converge, all modified section traces were classified as benign by our IDS, i.e., camouflage algorithm effectiveness of 100 %. This is due-to the same rationale mentioned in Sect. 5.

To test a complete "end-to-end" application of our system in real-life, we used the source code of the virus "Beetle"[6]. We compiled the source code and ran it through our IDS. The virus system calls trace was classified correctly as malicious by our IDS. After using our camouflage algorithm, we received the modified system calls sequence, classified as benign by our IDS. We manually matched the system calls in this sequence to the virus original source code, and applied the same modifications to it - and then recompiled the modified version. The modified version of the virus was then run in our IDS, and was falsely classified by it as benign. As expected, the malicious functionality of the code remained intact.

6.1 Comparison to Other Classification Algorithms

We've implemented and compared the effectiveness of different classification algorithms, using the same features, training set and test set. In-order to take into account true and false positives and negatives, we tried to maximize the Matthews correlation coefficient (MCC), which is used in machine learning as a measure of the quality of binary classifications [1].

The results appear in Table 1.

Table 1. Detection rate of the IDS by classifier type

Classifier type	Malware detection rate (TPR)	Benign software detection rate (TNR)	MCC
Decision tree	84.3	88.9	0.76
Random forest	86.1	89.5	0.77
k-Nearest Neighbors	89.4	86.0	0.77
Naïve Bayes (Gaussian)	87.0	54.5	0.50
Naïve Bayes (Bernoulli)	97.9	59.9	0.64
Ada-Boost	87.4	84.8	0.74
Support vector machine (Linear)	87.5	86.4	0.76
Support vector machine (RBF)	96.3	74.9	0.74
Linear discriminant analysis	82.6	82.6	0.68

The Random Forest classifier and k-Nearest Neighbors classifier were the best overall, taking into account both malware and benign software detection rate (by maximizing the MCC).

[6] The description and the source code of this virus are available at: http://vxheaven.org/lib/vpe01.html.

6.2 Countering the Camouflage: Section-Based Transformations

The basic form of decision tree node condition is: *system_call[i]= ?system_call_type[k]*. However, using this kind of input makes the IDS classification fragile: It's enough that we add a single system call in the middle of the sequence or switch the positions of two system calls, to change the entire decision path.

Therefore, we want to transform the input data (the system calls array) in a way that would make a modification of the inspected code harder to impact the decision tree path of the modified code, thus counter the camouflage algorithm. In-order to define those transformations, we first divide the system calls sequence to small sections of consecutive system calls. Each system calls section would have a fixed length, *m*. Thus, *section[i]=(sys_call[(i-1)*m+1],..,sys_call[i*m])*.

In an *order-preserving without duplicates removal section-based transformation*, we define the discrete values of the the decision nodes in the tree to be: *section[i]* =? *(sys_call[*(i-1)*m+1], sys_call[(i-1)*m+2],.., sys_call[i*m])*. However, this transformation is more specific than the basic model - so it would be easier to fool - and thus we didn't use it. This changes when adding *duplicates removal*: If there is more than a single system call of the same type in a section - only the first instance (which represent all other instances) appears in the section. This transformation prevents the possibility to split a system call into separate system calls (e.g. two *NtWriteFile()* calls, each writing 100 bytes, instead of a single call writing 200 bytes). Therefore, this was the first transformation we used.

The second transformation we examined is *non-order-preserving without duplicates removal*. This transformation is identical to the *order-preserving without duplicates removal transformation*, except for the fact that the system calls in each section are ordered in a predetermined order (lexicographically), regardless of their order of appearance in the trace. Using this transformation makes the probability of affecting the decision tree path by switching the places of two arbitrary system calls much smaller. Only the switching of two system calls from different sections might affect the decision path.

The last transformation we considered is *non-order-preserving with duplicates removal*. It is identical to the former, except for the fact that if there is more than a single system call of the same type in a section - only one instance (which represent all other instances) would appear in the section. This transformation handles both system calls switching and splitting. Notice that this transformation makes a section similar to a set of system calls: Each value can appear at most once, without position significance.

In-order to test the detection rate of our modified IDS, we used the same test set used for the basic model. A section size of *m=10* was chosen. The detection rate, computed by the definitions specified in Sect. 3, appear in Table 2.

As can be seen from this table, section-based transformations are effective, by the definition in Sect. 3.

In-order to test our camouflage algorithm effectiveness vs. the modified IDS, we have used the camouflage algorithm shown in Algorithm 1 to modify the system calls trace. Then we have applied the input transformation on the

Table 2. Detection rate of the IDS by input transformation type

Input type	Malware detection rate	Benign software detection rate
No transformation (original DB)	84.3	88.9
Non order-preserving, without duplicates removal	87.4	90.7
Non order-preserving, with duplicates removal	86.5	88.1
Order-preserving, with duplicates removal	87.6	91.3
No transformation (updated DB)	86.5	88.7

modified system calls trace - and then we fed it to the input-transformed IDS variant. Without transformation we got a false-negative rate of 100 %. With section-based transformation, non order-preserving, without duplicates removal - we got 18.8 %. With section-based transformation, non order-preserving, with duplicates removal we got 17.2 %. With section-based transformation, order-preserving, with duplicates removal - we got 17.4 %.

We see that each input transformation reduces the effectiveness of the camouflage dramatically, since the camouflage we applied was designed against individual system calls and not against input transformations.

Countering the Input Transformations with Custom-Fit Camouflage Algorithm

One might argue that camouflaging a system calls trace in our basic IDS (without the transformations suggested in Sect. 6.2) is an easy task. One needs to add only a single system call at the beginning to change all following system calls positions, thus affecting the decision path in the tree. Can we apply our camouflage algorithm on our section-based IDS with the same effectiveness?

In-order to fit our camouflage algorithm to section-based transformations, we have used Algorithm 1, except that in each iteration we added an entire system calls section, instead of a single system call. This is done in step 4: Assuming the condition is:

section[i] =? *(sys_call[(i-1)*m+1], sys_call[(i-1)*m+2], .., sys_call[i*m])*,

if the edit_op is $Sub_L, Add_R,$ or Del_L then:

Add *section[i] = (sys_call[(i-1)*m+1], sys_call[(i-1)*m+2],.., sys_call[i*m])*

(add the same section).

Else: Add *section[i]=(sys_call'[(i-1)*m+1], sys_call[(i-1)*m+2],.., sys_call[i*m])*

*s.t. sys_call'[(i-1)*m+1] != sys_call[(i-1)*m+1]*

(add a section with a different first system call).

The section is added with the same transformation type as the IDS: either *order preserving* or not, and either with or without *duplicates removal*.

We have applied this algorithm on all section-based transformations described in Sect. 6.2. Like Algorithm 1, there is no guarantee that the algorithm would converge. However, all 436 modified section traces were classified as benign by our IDS, with all input transformations, i.e., camouflage algorithm effectiveness of 100 %. This is due-to the same rationale mentioned in Sect. 5. This was also the case when modifying the random forest camouflage algorithm mentioned in Sect. 5.1 to counter input transformations by replacing Algorithm 1 used by it with this variant.

6.3 Partial Knowledge of the IDS

So far, we assumed that the attacker has full knowledge of both the classifier type, the training set used to create it and its features, in-order to generate the exact same classifier and then use it to camouflage the malicious code. We can alleviate this assumption: If the attacker can gain partial knowledge about the training set, he can construct the simulated classifier using *only* the training set he knows about and use it in Algorithm 1. Such partial knowledge is easy to gather, e.g., using the VirusTotal[7] samples closest to the IDS release date, which are very probable to be used by the IDS. We have trained the attacker classifier using a part of the training set which is used by the IDS classifier, as mentioned in Sect. 6. We then camouflaged the entire test set using Algorithm 1, based on the attacker partial knowledge based classifier.

We discovered that a knowledge of 86.4 % of the IDS training set is enough to create a camouflage that is 56.6 % effective. A knowledge of 77.7 % of the training set provides camouflage effectiveness of 31.3 % and 69.1 % of it provides effectiveness of 25.4 %. We also tested a full knowledge of the training set, with different features being selected (in case of chi-square equality). In this case, the camouflage is 64 % effective. Finally, we tested a full knowledge of the attacker on the training set and features, followed by an update of the IDS training set size by 1.7 %, without the attacker knowledge. In this case, the generated camouflage was 75.5 % effective. This means that training set updates can decrease the camouflage algorithm effectiveness, which was supported by our results, which are not shown due to space limitation.

From all the experiments, it is clear that the camouflage algorithm is useful to an attacker even with partial knowledge of the classifier.

7 Conclusions

In this article, we have shown that malware code which has been identified by a specific machine learning classifiers (decision tree or random forest) can be camouflaged in-order to be falsely classified as benign. We have done so

[7] https://www.virustotal.com/.

by modifying the actual code being executed, without harming its malicious functionality. We then applied a defense mechanism to the camouflage algorithm, called input transformations, making it more robust, and showed that it can also be evaded.

This suggests that it is not enough to use a machine learning classifier with a large DB of benign and malicious samples to detect malware - one must also be aware of the possibility that such classifier would be fooled by a camouflage algorithm - and try to counter it with techniques such as continuous updating of the classifier's training set or application of the input transformation that we discussed. However, as we have shown, even such transformations are susceptible to camouflage algorithms designed against them.

Our future work in this area would examine the effectiveness of our camouflage algorithm on other machine-learning classifiers (e.g. SVM, boosted trees, etc.) and find other algorithms to cope with such classifiers.

References

1. Baldi, P., Brunak, S., Chauvin, Y., Andersen, C.A., Nielsen, H.: Assessing the accuracy of prediction algorithms for classification: an overview. Bioinformatics **16**(5), 412–424 (2000)
2. Biggio, B., Rieck, K., Ariu, D., Wressnegger, C., Corona, I., Giacinto, G., Rol., F.: Poisoning behavioral malware clustering. In: Proceedings of the 7th ACM Workshop on Artificial Intelligence and Security (2014)
3. Firdausi, I., Lim, C., Erwin, A.: Analysis of machine learning techniques used in behavior based malware detection. In: Proceedings of 2nd International Conference on Advances in Computing, Control and Telecommunication Technologies, pp. 201–203 (2010)
4. Forrest, S., Hofmeyr, S., Somayaji, A., Longsta, T.: A sense of self for unix processes. In: IEEE Symposium on Security and Privacy, pp. 120–128. IEEE Press, USA (1996)
5. Forrest, S., Hofmeyr, S., Somayaji, A.: The evolution of system-call monitoring. In: Proceedings of the Annual Computer Security Applications Conference, pp. 418–430 (2008)
6. Gambs, S., Gmati, A., Hurfin, M.: Reconstruction attack through classifier analysis. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) DBSec 2012. LNCS, vol. 7371, pp. 274–281. Springer, Heidelberg (2012)
7. Hamlen, K.W., Mohan, V., Masud, M.M., Khan, L., Thuraisingham, B.: Exploiting an Antivirus Interface. Comput. Stand. Interfaces **31**(6), 1182–1189 (2009)
8. Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: Proceedings of the 10th International Conference on Knowledge Discovery and Data Mining, pp. 470–478 (2004)
9. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (2001)
10. Ming, J., Xin, Z., Lan, P., Wu, D., Liu, P., Mao, B.: Replacement attacks: automatically impeding behavior-based malware specifications. In: Malkin, T., Kolesnikov, V., Lewko, A.B., Polychronakis, M. (eds.) ACNS 2015. LNCS, vol. 9092, pp. 497–517. Springer, Switzerland (2015)

11. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: 23rd Annual Computer Security Applications Conference, pp. 421–430 (2007)
12. Moskovitch, R., Gus, I., Pluderman, S., Stopel, D., Fermat, Y., Shahar, Y., Elovici, Y.: Host based intrusion detection using machine learning. In: Proceedings of Intelligence and Security Informatics, pp. 107–114 (2007)
13. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting system emulators. In: Garay, J.A., Lenstra, A.K., Mambo, M., Peralta, R. (eds.) ISC 2007. LNCS, vol. 4779, pp. 1–18. Springer, Heidelberg (2007)
14. Rozenberg, B., Gudes, E., Elovici, Y., Fledel, Y.: Method for detecting unknown malicious executables. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) RAID 2009. LNCS, vol. 5758, pp. 378–379. Springer, Heidelberg (2009)
15. Somayaji, A., Forrest, S.: Automated response using system-call delays. In: Proceedings of the 9th USENIX Security Symposium, pp. 185–198 (2000)
16. Sufatrio, Yap, R.H.C.: Improving host-based IDS with argument abstraction to prevent mimicry attacks. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 146–164. Springer, Heidelberg (2006)
17. Tandon, G., Chan, P.: On the learning of system call attributes for host-based anomaly detection. Int. J. Artif. Intell. Tools **15**(6), 875–892 (2006)
18. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM Conference on Computer and Communications Security, pp. 255–264 (2002)